COMP 310

Winter 2024

Assignment 2

Posted: Monday, February 12

Due: Friday, March 1st, 11:59 p.m.

This assignment is significantly longer than the first one, so please plan your time wisely. Don't hesitate to ask questions on Ed if you get stuck!

Please read the entire PDF before starting. It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for us to run and grade your code.

This assignment will count for 20% of your final grade. To get full marks, you must follow all directions below:

- You must write your code in the C programming language. We use C in this class because most practical contemporary OS kernels are written in C/C++ (e.g., Mac, Linux, Windows and others).
- Make sure that all filenames and any specified functions/data structures are **spelled exactly** as described in this document; otherwise, the code will likely receive a zero. You are free to modify function signatures.
- The output from your program must **exactly match** the output that we specify, except for spaces.
- Make sure that your code **compiles**. Code that does not compile will likely result in a zero.
- Write your name and student ID in a comment at the top of all files modified from the starter code.
- Your code must follow our style guidelines, which are listed on the next page. **Up to 30% can be removed for code that does not comply to our style guidelines.**

Hints & tips

- **Start early.** Programming projects always take more time than you estimate!
- You are free to include any header files from the C standard library that are available for including by default in the given Docker image.
- Do not wait until the last minute to submit your code. **Submit early and often**—a good rule of thumb is to submit every time you finish writing and testing a function.
- Write your code **incrementally**. Don't try to write everything at once. That never works well. Start off with something small and make sure that it works, then add to it gradually, making sure that it works every step of the way.
- Seek help when you get stuck! Check our discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to a TA during office hours if you are having difficulties with programming. Go to the instructor's office hours if you need extra help with understanding a part of the course material.

  - Note that small amounts of code can be posted on the discussion board as *private* posts, which only the instructors and T.A.'s can see. But if more than a few lines of code are in question, then it is better to seek help in person. Often the problem requires a different *approach* such as proper use of the debugger, and the discussion board is a poor medium for this kind of help. (You could also post a single line of code and error message (error traceback) as a public post.)

- If you come to see us in office hours, please do not ask "Here is my program. What's wrong with it?" We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. Reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.
  - However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried in order to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

Policy on Academic Integrity

See section 5 of our syllabus for our course policies on academic integrity. In short, you must write the assignment on your own (or with a partner, if you have registered them as such). Do not copy someone else's work, nor share your work with someone else. Do not copy from any online sources.

Late policy

Late assignments will be accepted up to two days late and will be penalized by 10 points out of 100 per day. Note that if you submit one minute late, it is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Revisions

Search for the keyword *updated* to find any places where the PDF has been updated.

Style guidelines

As students at the 300 and 400 level, it is expected that you code at the level of a beginner software engineer. Please follow the brief coding style below in all your OS assignments.

- Indentation: All blocks of code must be indented by a minimum of 4 spaces, or with a 1-tab. (Pretend you are writing Python code.)
- Horizontal spacing: Parts of the code that are logically different should be separated by a blank line.
- Comments: Provide useful explanations to the reader of the code. We do not expect you to comment all your code, but we do expect explanations for more complex parts. In other words, do not write comments that just restate the code; here are some situations where a comment may be appropriate:
  - describing the purpose of a function
  - explaining the high-level steps of a function
  - explaining magic numbers
  - describing side-effects of a function
- Names: Name your variables, constants, functions, objects, methods, data structures and files appropriately. The purpose of each should be obvious from the name and can save you from writing a comment.

Compilation environment

Because of the many different versions of gcc and C library implementations, we will ask you to use a standardized testing environment when compiling and testing your code. We will be using the same environment when grading your code, so it's important that we all use the same one. If your code does not compile in this environment, you will receive a very low mark, and possibly a zero.

We will use **Docker** for our testing environment. Here are the steps you should follow:

1. Install Docker by following the steps at `https://docs.docker.com/get-docker`. Docker is available for Mac, Linux and Windows.

   Note: You do not need to sign up for a Docker account. You can skip that step.

2. Open up a command line and enter the following command:

   ```
   docker pull gcc:13.2
   ```

3. `cd` to the directory containing your assignment code files (*.c, *.h and Makefile). Then enter the following command to run a Docker shell. This command is long, so you may want to alias it:

   ```
   docker run --rm -it --mount type=bind,source=.,target=/code --entrypoint /bin/bash -w /code gcc:13.2
   ```

4. Once you are in the Docker shell, you should find yourself in the `/code` directory by default. This directory will contain all the files in the folder you were in before launching the shell. You can now run `make` to compile your code.

   Note: Any changes made to your files here (including the creation of your executable) **will** be reflected in your original folder (on your hard drive). So, be careful not to delete any files from within Docker.

You are also free to modify the Docker image by adding additional packages (such as vim) to it, as long as these packages do not change the compilation environment (e.g., by adding a new compiler or new libraries). You can add packages by doing the following:

1. Create a file called `Dockerfile` with the following contents. This one installs `vim` and `valgrind`:

   ```
   FROM gcc:13.2
   RUN apt-get update && apt-get install -y --no-install-recommends vim valgrind
   ```

2. In the command line, cd to the folder containing your Dockerfile and enter the following command:

   ```
   docker build . -t 310 -f Dockerfile
   ```

3. Then run the same command as in step 3 above (first cd'ing to your code folder), but replace gcc:13.2 by 310 at the end of the command.

Submitting your code and running the public tests

For each question, we provide **examples** of how your code should behave. All examples are given as if you were to run the commands while inside the shell.

During grading, we will run these examples and verify that your code outputs the same as given in the examples. Part of your assignment mark will be determined by the results of these tests. We will also run additional, private tests on your code. Therefore, it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples. **You are free to handle corner cases as you wish, unless we explicitly mention how to handle such a case.**

To run your code on our public tests, as well as to submit your code for grading, please check post #72 (Submission instructions) on Ed. For submission of late assignments (within the two-day late period), check post #274.

Welcome to Operating Systems Assignment 2.

We have implemented a shell. Now we are in our paging era.

Starter files

We provide starter code for this assignment which contains our solution for Assignment 1 as well as additional changes listed below.

Note: If you have made improvements to the shell you made for Assignment 1, e.g., by adding additional commands (for the optional components), then you may wish to port those changes to our new starter code. You don't have to, of course. But it would be a bit sad to have done that extra work and then just discard it!

Changes in new template code

A round-robin **scheduler** has been added to the shell which allows several scripts to be executed in a round-robin fashion. To execute several scripts in this manner, a new command **`exec`** has been added. **`exec`** takes up to three files as arguments (e.g., `exec prog1 prog2 prog3`, where `progN` are text files containing script commands). Then the scheduler will execute one command from each `progN` file at a time, until all commands in those files have been executed.

Here are some notes on our implementation:

- When `exec` is executed, the contents of each specified script file will be read and fully loaded into the shell memory.
- A Process Control Block (PCB) for each script file will be created to represent the script's process. Each PCB will store the process PID (unique to each process), the location in shell memory where the file contents of the script was stored, and the current instruction of the script to execute (i.e., the program counter).
- A ready queue is used to store the PCBs of the running processes. It is implemented as a linked list. Each element in the queue has a next pointer which points to the next PCB in the ready queue. The queue also has a head pointer to point to the first PCB in the queue.
- The scheduler runs the process at the head of the ready queue by sending its current instruction to the interpreter.
- After the instruction is done, the scheduler switches to a different process according to the scheduling policy. It updates the ready queue according to said policy and then executes the current instruction of the next head process.
- When a process is done executing, its source code is removed from the shell memory.

Here is an example. Given the following three files:

prog1_pdf

```
echo helloP1
set x 10
echo $x
echo byeP1
```

prog2_pdf

```
echo helloP2
set y 20
echo $y
print y
echo byeP2
```

prog3_pdf

```
echo helloP3
set z 30
echo byeP3
```

we can run them concurrently using the following commands, and get the following output, assuming that the round-robin scheduler runs with a time slice/quantum of 2 (i.e., two instructions per process before switching):

```
$ exec prog1_pdf prog2_pdf prog3_pdf
helloP1
helloP2
helloP3
10
byeP1
20
20
byeP3
byeP2
$
```

Specifically, the following files were added to the template code:

- `pcb.c` and `pcb.h` to implement the Process Control Blocks
- `ready_queue.c` and `ready_queue.h` to implement the ready queue
- `kernel.c` and `kernel.h` to implement scheduling and process creation

And the following files were modified:

- `interpreter.c` and `interpreter.h`: to add the `exec` command and update the `run` command implementation, and miscellaneous error handling improvements
- `shellmemory.c` and `shellmemory.h`: to implement loading the script files into shell memory
- `shell.c` and `shell.h`: small miscellaneous changes
- `Makefile`: to add the new files.

As before, to compile your code, you will run `make clean; make myshell` in the Docker container as detailed earlier in this PDF. Also like the last assignment, to run your shell, you can either run in interactive mode

(by typing `./myshell` at the prompt), or in batch mode, by making a file of commands and then typing `./myshell < commands.txt`. We will be using the latter mode to test your code.

We encourage you to compile the starter code now and play around a bit with the `exec` command so that you understand how it works before continuing to read this PDF. You are going to need to have to modify our implementation, so part of this assignment is to understand the existing code given to you. As always, if you have troubles with this part after spending some time on it, feel free to come to see us in office hours or post on Ed to obtain clarifications.

Your tasks

Your tasks for this assignment are as follows:

1. Implement a paging system.
2. Implement a demand paging system.
3. Implement the LRU page 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 `run` and `exec` commands will need to use paging.

Details on the behavior of your memory manager follow in the rest of this section. We will make recommendations, but you have full freedom for your implementation, as long as it complies with all instructions and edge cases that we specify below.

For each sub-part below, we also note how (relatively) lengthly it may take you to accomplish it, by indicating (Short), (Medium) or (Long). If you find that you are taking a very long time on a short part, then it may be helpful to rethink your implementation and possibly seek advice in office hours/Ed as there could be a much easier way to solve it. Note however that these notes are only our best approximations and we do not attach fixed amounts, but only specify them for a relative comparison.

Also, a reminder that we will be having a lab on this assignment on Wednesday, February 14 (see our Ed post for details). You are strongly encouraged to attend the lab or watch the recording afterwards as it will go through some helpful tips for getting started with the assignment.

Part 1: Implement the paging infrastructure

We will start by building the basic paging infrastructure. For this intermediate step, you will modify the `run` 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 `run`/`exec` commands in the provided template code. However, this step is crucial, as it sets up the scaffolding for demand paging in the following section.

As a reminder, the run command is specified as follows:

`run SCRIPT.txt`: Executes the commands in the file `SCRIPT.txt` in the current directory

`run` assumes that a file with the given name exists in the current directory. It opens that file and 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.

And the exec command specification is as follows:

`exec prog1 prog2 prog3` (where `prog3` or `prog2 prog3` may be omitted).

Note that the same script may be passed twice (e.g., `exec prog1 prog1` is valid). Also, note that we will not test recursive `exec` calls.

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

1. (Short) **Set up the (empty) backing store for paging.** A backing store is the part of the hard drive that is used by a paging system to store information not currently in main memory. In this assignment, we will represent the backing store as a directory.

   - An empty backing store directory should be created when the shell is initialized. It should be created inside the current directory. In case the backing store directory is already present (e.g., because of an abnormal shell termination), then the initialization should remove all contents in the backing store.
   - The backing store should be deleted upon exiting the shell with the `quit` command. (Note: If the shell terminates abnormally without using `quit` then the backing store may still be present.)

2. (Medium) **Partition the shell memory.** The shell memory should be split into two parts:

   - Frame store. A part that is reserved to load pages into the shell memory. The total lines in the frame store will be a multiple of the size of a single frame. In this assignment, each page (and thus frame) will have three lines, so the total lines in the frame store will be a multiple of three.
   - Variable store. A part that is reserved to store variables.

   To accomplish this you will need to modify the current shell memory implementation. Note that 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).

   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. (Short) **Add a shell command `resetmem`.** This command will take no arguments. It will clear the entirety of the variable store, setting all values to default (the string `"none"`). Note that it will not modify the frame store.

4. (Medium) **Load scripts into the backing store and frame store** when `run` or `exec` is called. The shell will load script(s) as follows.

   - The script(s) are copied as files into the backing store. The original script files (in the current directory) are closed, and the files in the backing store are opened. (Thus, if exec has identical arguments, then the same program will be copied into the backing store multiple times, and each copy will be its own process with its own PCB.)
   - Then, all the pages for each program should be loaded from the backing store into the frame store. (This will change in the next section.) Unlike in the starter code, where scripts were loaded contiguously into the shell memory, in your implementation the pages do not need to be contiguously loaded. (See example on next page.)
   - **When a page is loaded into the frame store, it must be placed in the first free spot (i.e., the first available hole).**

For example, assume you have two programs that each have six lines. Since each frame will store three lines, you will thus have two pages per program, which do not have to be contiguous in the 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	

5. (Long) **Create the page table.** You must extend the PCB data structure to include a page table. Each process will have its own page table. The page table will keep track of the loaded pages for that process, and their corresponding frames in memory.

   You are free to implement the page table however you wish. One possible implementation is creating an array for the page table, where the values stored in each cell of the array represent the frame number in the frame store. For instance, in the example above with the two programs of six lines each, 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

   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:

- The frame store is large enough to hold all the pages for all the programs. (We will modify this assumption in part 2 below.)
- 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 (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 multiple commands.

If everything is correct so far, your `run`/`exec` commands should have the same behavior as in the template code. We include some examples with the template code that you can use to make sure your code works properly.

Part 2: Extend the shell with demand paging

We are now ready to add demand paging to our shell. In the section above, we assumed that all pages of all programs fit in the shell memory's frame store. Now, we will remove this assumption, as follows:

1. (Short) **Set the sizes for the frame store and variable store as macros.** As you may know, when compiling with GCC, we can pass the `-D` flag followed by `name=value`. This flag will have the effect of replacing all occurrences of `name` in your compiled executable with the value `value`. We will use this flag to set the size of our frame store and variable store as follows.

   - Update your `Makefile` to add `-D name=value` flags to the gcc command for the two sizes. Use `"framesize"` and `"varmemsize"` for the two names.
   - Do not specify the values inside the `Makefile`. Instead, we will specify the values as an argument to `make`. By running `make xval=42`, the variable `xval` can be accessed inside the `Makefile` by writing `$(xval)`. Use this approach to specify the values for the two sizes.

     For example, you might call `make` like this: `make xval=42`

     And in your `Makefile` you might have: `gcc -D XVAL=$(xval) -c test.c`

     Then, in your C code, the term `XVAL` will be replaced by 42.
     E.g.: for the line `int x = XVAL;`, `x` will be set to 42.
   - Using the technique described above, your shell will be compiled by running

     `make myshell 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 `myshell`.
   - Include an extra printout in your shell welcome message as follows (on the line right after `"Shell v2"`:

     `"Frame Store Size = X; Variable Store Size = Y"`

     where `X` and `Y` are the values passed to `make` from the command line.

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

2. (Medium) **Load only part of each script into memory** when `run` or `exec` is called. In the previous section, we loaded the entirety of each script into the frame store. Here, we will modify our approach to load pages into the frame store dynamically, as they become necessary. We will begin by modifying the shell behaviour when a `run` or `exec` is called.

   - In the beginning of the `run`/`exec` commands, only the first two pages of each program should be 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 into the first available hole.

     The programs should then start executing normally, according to the round-robin scheduling policy with time slice of two lines of code.

3. (Long) **Handle page faults.** Since not all pages of a program may be in memory, there may come a point where a program needs to execute the next line of code that resides in a page which is not yet in memory. This situation is known as a page fault, and you will deal with it as follows:

   - 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.

- The missing page for process P is brought into the frame store from the file in the backing store. 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 improve on this policy in part 3 of the assignment.) Do not forget to update P's page table.

  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."
  ```

- 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 into the shell memory in order (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, 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".

Part 3: Adding a 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. Note that, with this policy, a page fault generated by process P1 may cause the eviction of a page belonging to process P2.

(Optional) Part 4

The following sub-parts are not worth any marks and will not be graded. However, if you pass the tests for this part, you will be awarded the specified number of `COMP310COIN`.

Note: For those working with a partner, you will split the coins with your partner. However, your leaderboard score will appear as if you had each obtained the full amount.

500 coins In the given template code, there is a bug that occurs for certain inputs. If you fix the bug and pass the corresponding (optional) test on submission of your code, you will be automatically awarded 500 coins.

1000 coins Compare the number of page faults of selected programs under the Random and LRU page replacement policies. Analyze how the page fault rate varies based on program characteristics. Include your analysis in the `README.md` file (see below).

1000 coins Add another page replacement policy. Do not use random (since that what we implemented in Part 3). Instead, you could try FIFO (First In First Out) or another of your choice. Also, add a new shell command `pagepolicy policyname`. It will switch the current page replacement policy to the given policy.

2500 coins For the additional policy mentioned in the above bullet point, choose the EELRU page replacement policy, which is a paper that students are presenting the week of February 12. You can find the paper link in the presentations spreadsheet on myCourses and/or come to one of our presentation sessions that week to learn more about it.

README

Finally, add a **`README.md`** file into the same folder as your code files. In this file, state your name(s) and McGill ID(s), any comments you would like the TA to see, and whether you have implemented any optional parts, and if so, which parts, so that the TA will know to check for this.