Operating Systems

CMPT 424 • Fall 2026

– Semester Project - 150 points

Background

Our goal is to experience and come to intimately understand the fun of writing an operating system, and to get better at using AI tools to assist in your development process along the way. We'll use GitHub for this so create a private repository and add me (username *Labouseur*) as a collaborator, then e-mail me the URL.

The project is broken down into four parts, all of them due at the end of the semester. You can develop them at any pace you like, but you must do them in order, as each builds on the one before. With that in mind, Part One builds on "Project Zero", which I have written and you can download from our class web site.

Project Zero is an web app written in TypeScript. You are free to continue developing in that manner. TypeScript is all kinds of cool and an excellent language. But you don't have to use it. Feel free to suggest other languages you might like to use instead. As long as it's not Python, I'll probably be fine with it. But please ask me to be sure… and so I can talk you out of unwise choices. (For example, writing an OS in Prolog or LISP sounds like fun, and it probably is, but we only have 16 weeks together and that's not enough time.)

Examples of wonderful prior projects can be found on our class web site in the *Hall of Fame* section. Check them out for inspiration and to get an idea of what I expect from you. Those projects are all excellent, but none are perfect. Do not take them as canonical examples of OS project perfection. Rather, look upon them as inspiration. And remember, you don't need to make it a web app. (But you can. You do you… it's cool.)

Good luck, and enjoy the process!

Part One

Download Project Zero from our class web site. If you're not using HTML and TypeScript for your project, begin by re-implementing the Project Zero functionality in your language and platform. With that as a basis . . .

- ☐ Alter the ver command to display your own data.
- ☐ Add some new shell commands:
  - date - displays the current date and time
  - whereami - displays the users current location (use your imagination)
  - something else interesting and creative; surprise me
- ☐ Enhance the display with a graphic task bar that displays . . .
  - the current date and time

  - status messages as specified by the user with a new shell command: status <string>
    example: status I love Operating Systems
- ☐ Implement scrolling in the console/CLI.
- ☐ Other console/CLI enhancements:
  - Accept and display punctuation characters and symbols.
  - Handle backspace appropriately.
  - Implement command completion with the tab key.
  - Provide command history recall via the up and down arrow keys.
- ☐ Display a BSOD message (on the CLI) when the kernel traps an OS error.
  - Add a shell command to test this. Remember to include it in the help.
- ☐ Add a shell command called load to validate the user code in the code input area. Only hex digits and spaces are valid.

Part Two

Build on the functionality you built in Part One by adding the ability to load and execute one user program in 256 bytes of memory as specified by the 6502a op codes documented on our class web site.

- ☐ Modify the load command to copy the 6502a machine language op codes into main memory.
  - Put the code at location $0000 in memory
  - assign a Process ID (PID)
  - create a Process Control Block (PCB)
  - return the PID to the console and display it.
- ☐ Add a shell command, run <pid>, to run a program already in memory. Note: the user should be able to execute many load/run cycles in sequence.
- ☐ Execute the running program (including displaying memory, CPU, and PCB status as well as any output). Be sure to synchronize the CPU execution cycles with clock ticks.
- ☐ As the programs executes, update and display the memory, PCB, and CPU status (program counter, instruction reg, accumulator, X reg, Y reg, Z flag) in real time.
- ☐ Implement line-wrap in the CLI.
- ☐ Provide the ability to single-step execution.
- ☐ Allow the user to break the current program with Ctrl-C.

To do all this, you'll need to . . .

- ☐ Develop a PCB prototype and implement it in the client **OS**.
- ☐ Develop a memory **manager** and implement it in the client **OS**.
- ☐ Develop a core memory prototype and implement it in the **host** OS.
- ☐ Develop a memory **accessor** prototype and implement it in the **host** OS.
- ☐ Develop a CPU prototype and implement it in the **host** OS.

Operating Systems

CMPT 424 • Fall 2026

Part Three

Building on the functionality you built in Part Two, add the ability to execute multiple user programs at the same time.

- ☐ Allow the user to load three programs into memory at once.
- ☐ Add the following shell commands:
  - clearmem — clear all memory segments
  - runall — execute all programs at once
  - ps — display the PID and state of all processes
  - kill <pid> — kill one process
  - killall — kill all process
  - q <int> — let the user set the Round Robin quantum (measured in cpu cycles)
- ☐ Display the Process queue and its contents (including state, location, base, limit, segment, priority, and current quantum) in real time.
- ☐ Track and display turnaround time and wait time (measured in cpu cycles) for each process.

To do all this, you'll need to . . .

- ☐ Store multiple programs in memory, each in their own partition /segment, allocated by the client OS (which obviously needs to keep track of available and used partitions/segments in the MMU).
- ☐ Add base and limit registers to your core memory access code in the **host** OS and to your PCB objects in the **client** OS.
- ☐ Enforce memory partition boundaries at all times.
- ☐ Create a Resident list for the loaded processes and a Ready queue for the running processes. These can be combined if you label it carefully.
- ☐ Instantiate a PCB for each loaded program and put it in the queue.
- ☐ Develop a CPU scheduler and dispatcher in the client OS using Round Robin scheduling with the user-specified quantum measured in CPU cycles (default = 6). Be sure that your client OS controls the host CPU with the CPU scheduler and dispatcher in the Kernel. Log all scheduling events.
- ☐ Implement context switches in the dispatcher with software interrupts generated by the scheduler. Only allow context switches after fully completed CPU instruction cycles. Be sure to update the mode bit (if appropriate), the PCBs, and the Process queue.
- ☐ Detect and gracefully handle errors like invalid op codes, missing operands (if you can detect that), and most importantly, memory out of bounds or access violation attempts.

Some hints:

- Do not reset PIDs as they are used. The PID value should always increase.
- The program counter for any process should never be greater than FF (255).
- Remember that scheduling is done via context switches triggered by the scheduler and implemented through software interrupts that are handled by the dispatcher.

- Updating your Ready Queue only on a scheduling events prevents the initial state from being displayed right away. This is bad.
- Display each processes' state in the Process Queue.
- What should happen if the user tries to clearmem while processes are running?
- Killing a process currently running on the CPU should work and should not create a zombie.

Part Four

Building on the functionality you built in Part Three, add a local file system and swapped virtual memory so you can execute more processes than you have partitions for in memory.

Add shell commands and implement functionality for the following disk operations, displaying a message of success or failure for each one:

- ☐ `format` — Initialize all blocks in all sectors in all tracks
- ☐ `create <filename>` — Create the file *filename*
- ☐ `read <filename>` — Read and display the contents of *filename*
- ☐ `write <filename> "data"` — Write the data inside the quotes (but not the quotes themselves) to *filename*
- ☐ `delete <filename>` — Remove *filename* from storage
- ☐ `copy <existing filename> <new filename>` — copy
- ☐ `rename <current filename> <new filename>` — rename
- ☐ `ls`— list the files currently stored on the disk.
- ☐ `format –quick` : Initialize just the first four bytes of every directory and data block
- ☐ `link file1 file2` : make both file names point to the same file data. (What happens on delete?)
- ☐ `alias command1 command2` : Alias shell commands so you can invoke with either one.

Then . . .

- ☐ Support hidden files (that do not show up in `ls` output) with filenames that begin with a period.
- ☐ Implement a command line option for `ls`:
  `ls –a` that lists all file names (even hidden ones) as well as their size and create date.
- ☐ Implement *chkdsk* and *fsck*-like utilities:
  - recover deleted files
  - reclaim data blocks that are in use but not indexed in the directory blocks
  - defragment the data blocks
- ☐ Add FCFS and non-preemptive priority scheduling algorithms to your CPU scheduler. (Keep RR as the default.) Include `getSchedule` and `setSchedule` shell commands to control this.

Implement your file system as discussed in class, including

- ☐ a disk system viewer in your OS interface
- ☐ a Disk System Device Driver (dsDD) for all of the functional requirements noted above.

- Load the dsDD in a similar manner as the keyboard device driver.
- Develop your dsDD to insulate and encapsulate the implementation of the kernel-level I/O operations (noted above) from the byte-level details of your individual blocks on the local storage.

Implement swapped virtual memory with enough physical memory for three concurrent user processes.

- ❑ Allow the OS to execute four or more concurrent user process by writing roll-out and roll-in routines to . . .
  - Take a ready process and store it to the disk via your dsDD.
  - Load a swapped-out process from disk and put it in the ready queue.
  - Your ready queue should denote which processes are where.

A final challenge: Add demand paging in place of swapping.

