CMSC 312
INTRO TO OPERATING SYSTEMS

OPERATING SYSTEM SIMULATOR

Report Submitted on: December 10th, 2019

Xander Will

Overview

The included operating system simulator is a concentrated effort to explore various design philosophies behind creating a full operating system. The simulator includes support for fake “processes” with a variety of different instructions they can execute. These are run using a dispatcher module with their fake “data” stored in a memory module. The dispatcher runs a set of 4 threads from said processes simultaneously as well as scheduling which threads will come next at any given point. This is all run by a overall simulator which generates IO events, tracks the number of cycles, and responds to commands from the built-in GUI.

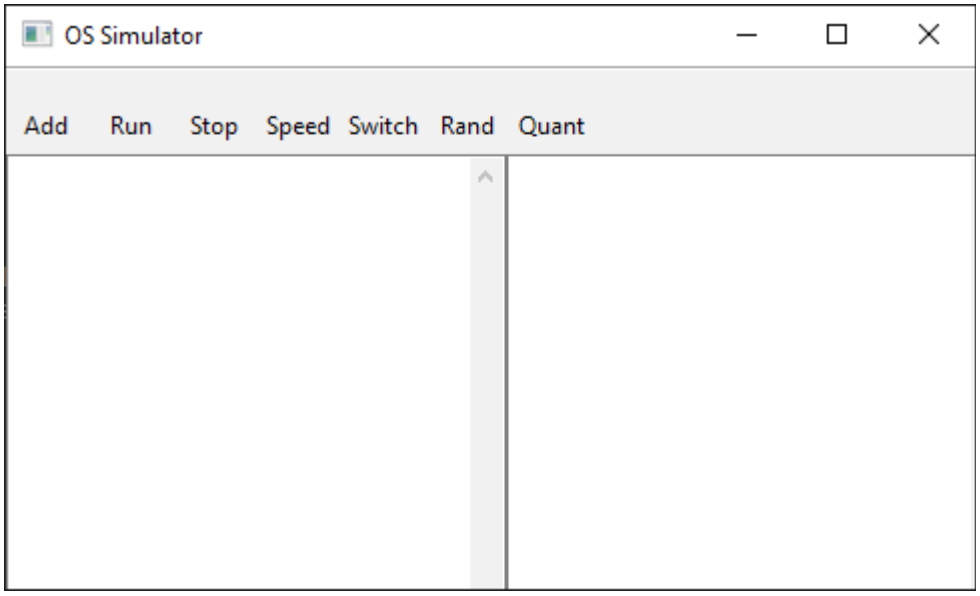
Usage

To boot, enter the following at the command line when located in the root of the provided directory:

```
python -m src
```

Please note that this program runs on Python 3 (3.7.2 was used to author this) and the wxPython package is required. This can be installed using pip.

On boot, the following screen should appear:



What follows is documentation on each of the toolbar buttons:

Add: This opens a file prompt in order to get the path for a process file (several are provided), as well as prompting the user on how many copies of the process should be open. Once confirmed, the simulator will open the specified number of copies of the process.

Run: This signals the simulator to begin running. While the GUI is up, the simulator is always polling its command mailbox, so various commands from the user will register whether the OS is running or not. Do note that the real-time statistics will only appear after running for 5 cycles. If the OS is already running, this button will do nothing.

Stop: This signals the simulator to halt. Progress and state are not lost, and can be resumed by pressing the *Run* button. If the OS is not already running, this button will do nothing.

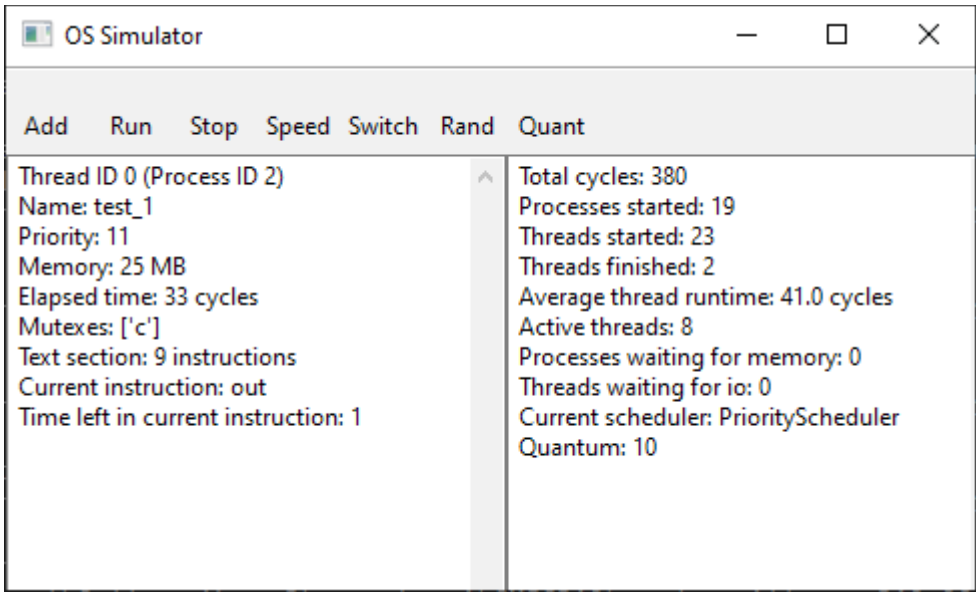
Speed: This prompts the user to provide the refresh rate of the simulator in hundredths of a second. This has a maximum of 10 seconds.

Switch: This will toggle the scheduler used by the simulator.

Rand: This will generate a random process file to be used by the OS. The file is written to the `./processes/generated` folder. This is immediately read in by the simulator and copied a user-specified number of times. If 0 is specified, the OS will not read in the process but will still write the file.

Quant: This will prompt the user to enter a time quantum for the dispatcher to use. The OS defaults to 10 cycles.

After running the OS for more than a few cycles: the real-time statistics should kick in:



On the left is the thread output window, which displays statistics from threads when they use the “out” instruction. Each “out” call will immediately overwrite the last one. On the right are constantly updated statistics on the operation of the OS. All of these are relatively self-explanatory, as well as the thread statistics. It is worth noting that “Active threads” refers to threads present in any of the ready queues. The number of threads waiting on mutexes are not shown.

Processes

Processes are implemented as JSON files, with a header consisting of the process name and priority and a body consisting of an array of instructions. Many examples are included. What follows is a list of instructions, their operation, and their required arguments:

c*alculate*: This instruction simulates general operation of a process for a given number of cycles. The number of cycles is required by the process parser as *cycles*. If “null” is used, a random cycle count will be generated from 1 to 30, allowing for templates to be written.

io: This instruction simulates IO for a process. If an IO event has not been generated, the process will block until one occurs. The number of cycles is required by the process parser as *cycles*. If “null” is used, a random cycle count will be generated from 1 to 30.

out: This instruction outputs the current thread statistics to the user interface. This requires no arguments.

yield: This instruction ends the current execution of the thread until it is scheduled again. This requires no arguments.

exe: This instruction ends thread execution entirely and removes it from all execution queues. This requires no arguments.

acquire: This instruction acquires a mutex. If the mutex is already taken, the thread will block execution until it has been has acquired. This requires the name of the mutex (either 'a', 'b', or 'c') as *monitor*.

release: This instruction releases a mutex. If the mutex is not currently held, nothing happens. This requires the name of the mutex (either 'a', 'b', or 'c') as *monitor*.

fork: This instruction creates a new process based on the enclosing process of the thread, with execution starting after the *fork* instruction. This requires no arguments.

thread: This instruction creates new threads attached to the process of the running thread, with execution starting after the *thread* instruction. This requires the number of threads to create as *num*.

send: This instruction sends a message through one of the mutexes. The message can be either true or false, and the thread does not have to be currently holding the mutex in order to change the value. This requires arguments for both the value to store (as *val*) and the name of the mutex (either 'a', 'b', or 'c') as *monitor*.

read: This instruction reads a message through one of the mutexes. If the value is true, the next instruction is executed. If not, the next instruction is skipped. This requires the name of the mutex (either 'a', 'b', or 'c') as *monitor*.

loop: This instruction resets the program counter to 0, allowing threads to be looped infinitely. This requires no arguments.

pipe: This instruction sends a message through a pipe controlled by the process that the thread draws from. The value can be either true or false. This requires the value to store as *val*.

check: This instruction checks the pipe controlled by the process that the thread draws from. If the value is true, the next instruction is executed. If not, the next instruction is skipped. This requires no arguments.

Memory is also a possible argument for any instruction. If set, it is used as the number of megabytes required for the instruction to run. If not set, the number is set to 1 internally. This is useful for instructions that are purely operational and need minimal memory.

Randomly generated processes pull exclusively from *calculate* and *io*, and always end with *exe*. More complex processes will have to be written by hand.

Daemons are supported via setting *daemon* to true in the header of each process file. This will prevent the priority from aging. Do note that the priority must be set higher than 15 to ensure that the daemon will always be placed into the background queue.

Requirements Met

This section will outline the various project requirements met by the simulator.

Process implementation and PCB

This is defined as a class in lines 145 – 213 of *process.py*.

Critical section within each process

Not all example processes have a critical section defined but one can be added with no hassle simply by adding *acquire* and *release* instructions somewhere within the process. Critical sections are implemented as monitors as a class in *monitor.py*.

Critical section resolving scheme

As previously noted, monitors are included as well as *acquire* and *release* instructions. This is shown in *monitor.py*. It should be noted that critical sections are thread-safe. Lines 162 – 171 of *dispatcher.py* handle the actual execution of the aforementioned instructions, and monitors are stripped from a thread (if they still possess them) when the thread is deallocated in lines 138 – 141.

Multiple inter-process communication methods

The aforementioned mutexes can be used as shared memory between different processes. This is defined in *monitor.py* with the majority of the code being handled in lines 204 – 213 of *dispatcher.py*. Pipes are also supported (though technically just within different threads of a given process) as shown in *dispatcher.py,* lines 216 – 223.

Multi-level parent-child relationship

Forking and threading are fully-supported as many times as required. Children can create their own forks and threads simply by including instructions in a process after a previous fork/thread. All threads are banded together under a single process, with all sharing the same text section. A fork will create a new process from the attributes of an existing one. Threading is supported in *process.py*, lines 196 – 202, while forking is handled in 207 – 213 of the same file.

Scheduler

Scheduling is handled in lines 232 – 293 of *dispatcher.py*.

Two schedulers and comparison

In the aforementioned bit of code, two schedulers are defined as separate callbacks within the dispatcher. Both coalesce all active threads and filter them into foreground and background queues. However, the RandomScheduler will shuffle each queue before drawing the running threads. The GUI provides an option to switch which one is currently being used.

Process priorities

These are defined in the process headers and stored when parsed. Priority is handled as smaller priorities having precedence over higher ones. Priorities are aged periodically as shown in lines 242 - 244 of *dispatcher.py*.

Multi-level queue scheduling

Two different queues are designated: foreground and background. When the scheduler chooses threads to run, it attempts a 3:1 ratio between foreground and background processes, though if the foreground queue has less than 3 threads then extra will be filled in from the background. This is demonstrated in the same scheduler code previously mentioned.

Process resources

Generally speaking, processes have the option for (and often require) memory for instructions, a slice of execution time, and shared memory.

Memory divided into hierarchy

Memory is divided up into cache (40 MB), physical RAM (4096 MB), and virtual memory (8192 MB). When an instruction executes, its first page of memory must be loaded into the cache. If it is not located, it is swapped in with an LRU algorithm, as demonstrated in lines 64 – 81 of *memory.py*. When memory is being allocated, the module attempts to find room first in physical memory, then virtual, then the cache.

Virtual memory with paging

Memory is entirely divided up into pages. When an instruction is run, all of its memory must be located in physical memory. Page swaps are handled via an LRU algorithm as shown in lines 83 – 90 of *memory.py*.

I/O interrupts and handlers

I/O events are generated in lines 78 – 88 of *simulator.py*. They are generated on 5% of cycles for a total of 25 – 50 cycles before ending. When this occurs, the schedulers will load threads from the I/O waiting queue as shown in lines 282 – 285 of *dispatcher.py*.

Multi-threading via hardware

Strictly speaking, the multi-threading for this application is not handled by the hardware due to Python's Global Interpreter Lock (or rather, due to this the hardware threads can not overlap). However, fundamentally the simulator is multi-threaded both from an overall design perspective (the main thread runs the GUI, another thread handles GUI commands, and a third drives the simulator) and from the dispatcher's perspective, where four hardware threads are spawned with each running a given simulated thread and then joining afterwards. These are synchronized on each “cycle” by the simulator. Lines 114 – 156 of *dispatcher.py* handle thread coordination and 159 – 223 are executed by each thread simultaneously.

GUI with real-time monitoring and visualizations of simulator performance / operations

The code for this is entirely located within *gui.py*, with the command classes sent and received located in *commands.py*.

Loading external processes and generating new ones on user request

This is handled by the GUI via the aforementioned *Add*/*Rand* buttons. Lines 12 – 56 of *process.py* handle individual instruction parsing while 146 – 184 handle the overall file parsing. Lines 215 – 234 provide a process generator.

Extra Credit

Daemon Support

Daemons/background processes are specifically allowed, with the *print_spooler.json* process showing what a typical one would look like. The code itself, other than the threads checking for it during file parse, lies in lines 239 – 244 of *dispatcher.py*.

Variable Quantum

The quantum can be changed within the GUI and updated on the fly during a simulation. This is handled in lines 71 – 77 of *gui.py*.

Variable Speed

The refresh speed of the simulator can be changed within the GUI and updated on the fly as with the quantum. This is handled in lines 90 – 95 of *gui.py*. Line 68 of *simulator.py* shows the actual place where the value is used to sleep.

Quirks

As may be noted, this project was completely rewritten in Python from C on short notice. This is due to repeated memory issues within the C program (memory holes within malloc likely from my MinGW build). This personally frustrates me endlessly, as Python is really not a suitable language for an OS in the slightest. However, when it came down to testing on another OS (I currently only have a Windows laptop), teaching myself how to use Valgrind (which for some reason is not taught in the ECE curriculum), or switch to Python, I decided that Python would save me a lot of time in my already limited schedule. The C source is included, though it has far fewer features than the finished product.

Some input errors in the GUI and the JSON files are still not handled. Many have been patched, but expect a few more to possibly appear.

Admittedly, some tests on long-running simulations (in the 10000 total cycles range) seem to eat a few threads here and there. A last-minute fix to the mutex system should hopefully solve this, but I'll be the first to confess I likely missed a few edge cases.