A Report by John Gilbert / 夏明傑 (R08922161)

UPDATE: I fixed a lot of issues with my original project by switching from linked lists to arrays. I also simplified my code and made it more readable with the use of structs. Being that looping through an array of structs did not take up as much time as originally thought, I further simplified my code by removing the sorted *ready queue* which originally only contained processes that were available. Instead, I just stored the status of a process inside its corresponding struct and looped through all of the structs to check available processes. I also added a global variable *global_steps*, which indicates the number of time steps all processes have taken, and shared the state of the variable with forked processes using *mmap*. Thus, the *main()* loop no longer needed to be in sync with running processes to know when new processes should start.

Design

The main logic of this program can be found in *main.c*, and general definitions and imports can be found in *definitions.h*. The overall logic of the program is as follows:

1) Get policy (defined in *get_policy.c*)
2) Get inputs (process names, ready times, and execution times)
3) Initialize parameters
4) Initiate loop that runs until all jobs are complete
5) Add jobs that are ready (by comparing *ready time* with *global step*)
6) Select job based on policy (defined in *select_job.c*)
7) Run selected job, and pause any previous jobs (defined in *run_job.c*)
8) Check if any jobs have finished and update their status (*update_status.c*)
9) Print debug info if DEBUG flag enabled (in *definitions.h*)
10) Print output (job names and PID)

New processes were started with *fork()*, paused with *kill(PID, SIGSTOP)*, resumed with *kill(PID, SIGCONT)*, and checked to see if finished with *waitpid(PID, &waitstatus, WNOHANG)*, where *waitstatus* would equal 0 if completed, and *WNOHANG* to make *waitpid()* non-blocking. The time step *global_step* was updated (+1) inside the for-loop of running processes to keep the ready times and execution times of all processes in sync.

Jobs were selected according to the given policy as follows:

FIFO: The status of all processes (stored in the struct *jobs*) was checked. Of jobs with a status of READY, the process with the lowest *ready time* was selected to run next. If the previous process was already running (indicated by a boolean variable *running*), then it would simply return the previous process id.

RR: Same as FIFO, except if the previous process was already running, the elapsed time (*global_step* minus the previous process *ready time*) mod 500

was used to check if the job should be rotated, and if so, then it would select the next job id with a READY status, and if no other jobs were ready, it would return the previous job id.

SJF: The setup was similar to FIFO, except that instead of selecting a READY job with the lowest *ready time*, it would select the one with the lowest *execution time*.

PSJF: Same as SJF, but preemptive, meaning it would select the job (with a status of READY) with the lowest *execution time*, regardless of whether or not a previous job was running.

Kernel Version

The Linux kernel used for compiling syscalls was Linux Kernel 5.4.35, and the kernel was accessed via an AWS instance. Information regarding setup can be found on the README.md of my github page.

Results, Expectations, Discussion

I was surprised at how much easier this project was once I decided to rewrite everything without linked lists. I feel like the biggest issue with my linked list implementation was my code would segfault when trying to access a pointer for a node that may no longer have been in scope. Storing all information for all processes in an array of structs and constantly looping through that array to check everything eliminated this issue, and made the program much easier to organize.