A Report by John Gilbert / 夏明傑 (R08922161)

Design

The main logic of this program can be found in *main.c*, which initiates the following procedures:

1) Get Policy and Inputs (*IO.c*)
2) Initialize Variables
3) Sort indices into an array (*sorted_ids[]*) based on arrival times. This was done using quicksort (found in *useful_funcs.c*)
4) Enter while loop, until all jobs are completed
5) Add jobs that are available to run into the ready queue (either a linked list or an array; I tried both)
6) Select the next job from the list based on the given priority (*scheduler.c*)
7) Start or resume the job using *process_control()* (found in *process.c*)
8) Stop previous jobs, if running (in *process_control()*)
9) Obtain and store the PID for the given job
10) Proceed forward one time_unit()
11) Check the status of the job
12) Update the elapsed time for that job
13) Compute the remaining time
14) Check if the job has finished
15) If finished, remove it from the ready queue (linked list or array)
16) Update the size of the ready queue and number of completed jobs, accordingly
17) Repeat the while loop

Step 6, the overall method employed by the scheduler, was different depending on whether or not the flag USE_LINKED_LIST was enabled. The following methods were employed for each policy using linked lists:

- **FIFO**: If linked lists were used, then jobs were added to the tail of the list when they become available, and the scheduler always selects the job at the head of the linked list. Note that when a job is complete, it is removed from the head of the list, and the next item is made the head of the list.
- **RR**: This behaves similar to FIFO, except it also checks the process elapsed time step against the current time step. If the process' elapsed step modulus 500 equals 0, then the list is shifted by moving the head of the list to the back and making the next item the head.
- **SJF**: This checks if the previous job is still running, and if so, then it selects the head of the list to let the job continue. Otherwise, it traverses the linked list to obtain the node in the list with the minimum remaining

process time steps. That node is then moved to the front of the list, and then the scheduler selects the head of the list to run.

- **PSJF**: This behaves almost identically to SJF, except that it does not check if the previous process was still running. It simply proceeds to find and select the node with the minimum remaining process time steps.

After implementing this with linked lists, I ran into issues with freeing pointers to nodes that should still exist, but somehow didn't. In response, I proceeded to throw together another solution involving arrays instead of linked lists, where the arrays were all initialized to negative values to indicate that they were not ready to run yet. The attempted methods with arrays were as follows:

- **FIFO:** Iterates through the array and selects the index of the first non-negative (ready, but incomplete) value. That index corresponds to the selected job.
- **RR:** Does the same as FIFO, except if the elapsed time modulus 500 equals 0, then it proceeds to cycle ready values. It does this by first finding the maximum value in the array, then it adds 1 to all positive values, and takes the that value mod the max value, such that the max value + 1 mod itself will reduce to zero, the lowest value in the array, thereby shifting the array.
- **SJF:** First checks if the previous process is running, and if so, selects the previous process. Otherwise, it traverses the array of remaining times, finds the minimum positive value, then returns the index for the job with that value.
- **PSJF:** Same as SJF, except without checking if the previous process was running. It automatically selects the available job with the lowest remaining run time.

After an index for a job is selected, it is passed to *process_control()*, which first checks if the previous process is running, and if so, pauses that process with *kill(previous_PID, SIGSTOP);* after which it checks the status of the job matching the index selected by the scheduler. If the status indicates the job has not started, then it starts the job using fork(), and returns the PID via the parent process back to main. If the process has the label of having already started, then it is continued with *kill(PID, SIGCONT);*. Originally, I had intended to use sched_setscheduler() or nice(), but then figured that controlling the jobs with priority to be less reliable than directly pausing or resuming them with kill().

The syscall to use dmesg and the timer was embedded in the for loop in the child process started by fork(), such that the child displays when it is finished with printk.

The status of a child process was checked from main using *waitpid(PID, &waitstatus, WNOHANG);* in which the resulting value of waitstatus would indicate whether or not the child process has completed. WNOHANG was used to make waitpid() non-blocking, so as to allow main() to continue running even if the child process had not yet finished. With that information, all other parameters could be updated.

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

There were bugs with both implementations (linked lists and arrays) in my code, primarily issues of segfaults and some processes never runnining, even though they should. This ultimately prevented me from getting any results. I spent hours and hours attempting to debug all the problems, littering the code with printf() statements that can be turned on by switching DEBUG and DIO flags to true in *definitions.h*. I believe the source of my errors lie not in the design, but in oversights during implementation. I currently do not consider C to be my best programming language, and for that reason I found debugging this particularly difficult. Feel free to offer tips and suggestions as to how my work might be improved.