Alyssa Ayala - Writing sample

The following writing sample is from a previous project that involved using C code, which can be viewed here. Specifically, I had to create a ready queue that held the addresses of several processes, which I admit was a challenge since I was still learning about pointers and memory management. This writing sample demonstrates my software engineering abilities by going over how I face difficult problems by discussing the problems that I've encountered and how I fixed them.

When it comes to figuring out how to work with the addresses of processes, I first needed to figure out how I could use the ready queue and FIFO (first-in-first-out) without any memory leaks. I knew that I wanted to use arrays for both because of how I could allocate memory and hence not need any additional methods to modify them. I had originally planned to create double pointer arrays to hold the addresses of the MiniPCBs (process control blocks), but that either pointed to the last element of them or gave a segmentation fault. It later occurred to me that I should be allocating memory to the indexes instead of the arrays themselves. I still used pointer arrays at the end, but with a set array size of 15 each, the defined maximum size for this project.

For the ready queue, as I was reading the input file, I first used `malloc` to allocate the size of a `MiniPCB`, and then `memcpy` to copy the address of the MiniPCB to the ready queue. For the FIFO, I found that memory had to be allocated in the scheduling algorithms for there to be no segmentation faults. So I created a global pointer that I could use to allocate the size of a MiniPCB so that the FIFO would have enough room for the MiniPCBs. I also found the output buffer in the logger function to be an issue, so I designed it so that it would be allocated the size of a MiniPCB and freed during each run of the logger thread. When the output file is finished, each index of the ready queue and global pointer are freed.

To program the schedulers, I designed them to return one process at a time. Hence I designed my pthreads to run the same amount of times as the size of the ready queue, which is how they know when to stop. I loop through the ready queue to find the smallest variable of concern depending on what scheduling algorithm is being used. Once the scheduling algorithm has decided which MiniPCB to send to the logger thread, the process is dispatched by calculating the return value of the input parameters in an operation.

For both the operation and scheduling algorithm functional arrays, I created two extra methods called `funcIndex` and `algorithmIndex`, where I put the string value of the argument or operation, and they return the appropriate array index. I figured that it would be more convenient to use them rather than have the scheduling algorithm run multiple conditional statements in it. I then set the variable of concern of the chosen MiniPCB to 1000 so that it acts as an empty place in the array. Since the for-loop looks for the smallest variable of concern, 1000 is large enough for the MiniPCB to not be considered for the rest of the program as if it were not there anymore.

The creation and joining of the scheduler/dispatcher and logger threads act as the `send()` and `recv()` operations. When joining the scheduler/dispatcher thread, `send()` occurs since the FIFO is being initialized at a specific index. Then `recv()` occurs in the creation of the logger thread since the logger function takes the return value of the scheduler/dispatcher thread by taking the FIFO at the same index as input. At first, I had an issue with this because a line would replace another and be printed twice in the output file. It turns out that it was how I created and joined the threads. Created threads are supposed to be joined immediately before focusing on other threads, but I had initially created and joined them in separate for-loops. Hence there was a memory leak since the threads are not being properly joined.