Implementation

The implementation was originally done in C and then migrated to C++ in order to expand the flexibility of our code. All code was compiled on a Linux system using GCC. The data structures in use are as follows:

- A struct for Processes containing PID, Parent PID, Program Counter, the process image file path, register value, priority, state, start time, total run time, and the remaining time slice.
    - Make_proc
        - Handles process creation
    - Fork_proc
        - Forks the current proc and handles incrementing the program counter
- A class for instructions (instruction_t) as a container for op codes and arguments. This parses the argument into the proper type for the opcode.
- A map from a string to vector of instruction_t to contain the programs themselves.
- A class for the CPU itself, this handles the actual processing of the processes as well as management. Data stored includes current time, PCB Table, a deque of ready and blocked processes, and the currently running process. Member functions of this class perform the bulk of process management tasks.
    - get_proc_by_id
        - A central procedure for retrieving a process object’s reference. Funnelling all calls through one procedure improves manageability.
    - manage_proc
        - Handles runtime tracking and whether the process should be parked.

    - tick
        - Increments the program counter and handles the next instruction.
    - load_proc
        - Handles Fetching and loading of the next ready process.
    - park_proc
        - Handles storing the process back into the ready queue if it has reached the end of its timeslice. Forces a call to load_proc on completion.
    - block
        - Handles storing the process into the blocked queue if it has executed a block command. Forces a call to load_proc on completion.
    - unblock
        - Handles unblocking a process and moving it to the ready queue.
    - print_proc
        - Outputs standard information about the specified process.

      print_current_state
        - Outputs the current state of the entire simulation.
- The PCB Table itself is implemented as a vector of process structs.
- Additional global functions
    - mgrHandleInput
        - Handles input from the console

A second program was also developed which simply generates random programs for testing and demonstration purposes. The source code for the project is hosted on github at https://github.com/zpolygon95/CSC-377-Project

Findings

Development of the program forced us to consider several issues for the flow and accessibility of information within the simulation. Among these was the issue of which data structures to create and what data to include within them. The CPU class is a prime example of this. Contained within the CPU class are all of the core management functions for processes in our simulation. Despite the fact that some of these (primarily the process ID lookup function) would be mainly functions of the memory and not the CPU, these were grouped into the CPU class as a matter of convenient program management. By centralizing these functions within the CPU, we could cleanly control the flow of data in and out of the class in a single location.

The PCBTable structure greatly aided our ability to manage the processes at work. By centralizing the actual processes into an index (the PCBTable) and creating a single lookup function, we were able to quickly handle all instances of needing to access process data without passing extraneous copies of the structures around. This also allowed us to ensure that no matter when a particular process was blocked or pushed back into the queue, that we could always guarantee it had the most current data. This resulted from the fact that our system only ever instantiates a single copy of any given process. While not necessarily how this works in reality (where parts of the process are loaded into the cache, worked on, then returned to main memory) it achieves the same end result.

In order to replicate the consistency of being able to push processes back into memory and switch to a new process, we had to design the process structures themselves to be entirely self-contained. Every process structure contains its PID, PPID, registers, and other relevant data. This simplified the loading/unloading process significantly because we did not need to copy data from the CPU, only ensure that the process was added back to a queue. All subsequent data could then safely be overwritten without endangering the integrity of the previous process.

In conclusion, this experimental simulation allowed us to get hands-on so to speak with the management of processes. We were able to functionally investigate, analyze, and determine our preferred flow of the processes between the CPU and associated memory. Doing this helped to familiarize us with the inner workings of the process management subsystem in an operating system, and the rationale behind certain design decisions of such systems. The simulation also provides a way to get an intuitive feel for how processes move within the process management system, as they rise and fall through the queues.