Project 5: Short-Term Process Scheduler

CST-315 Operating Systems

Angel Velazquez, Nathan Dilla

March 31, 2024

In this project, we expanded on the shell we created and developed in previous projects. We implemented a key shell component, determining whether a project should run or halt. In this project, we used threads to handle multiple processes at once. In this implementation, the scheduler evaluates the following events while determining the fate of a current process:

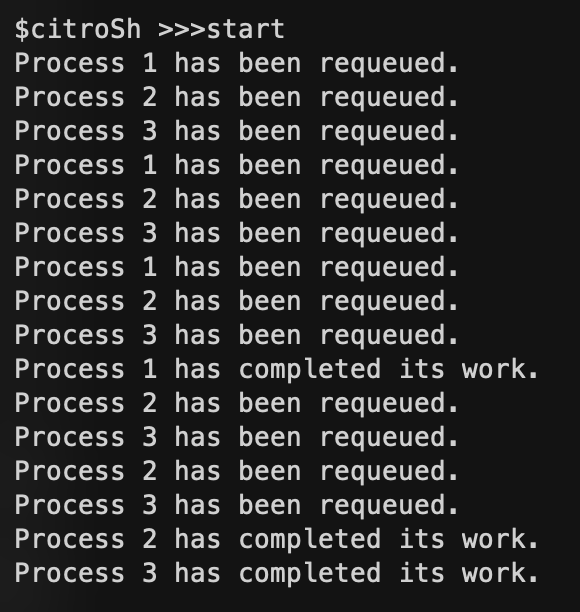
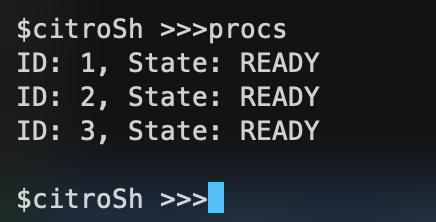
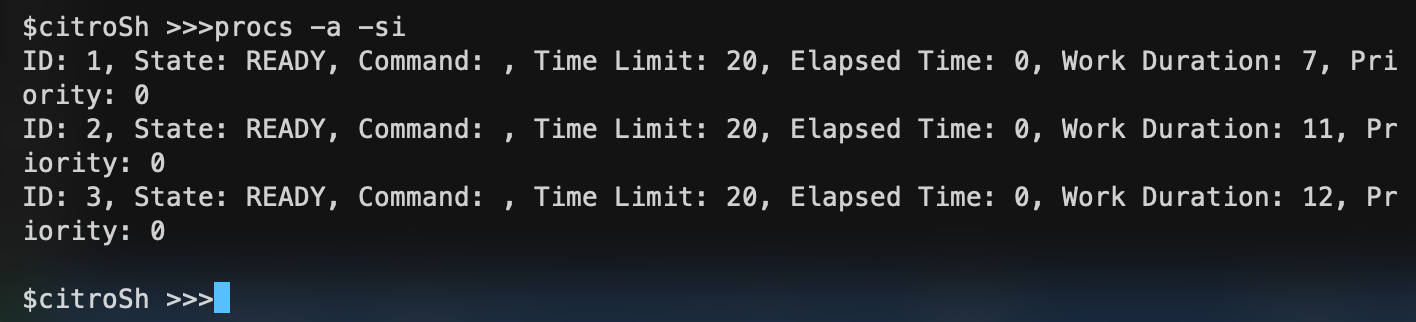
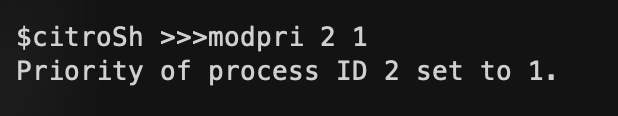
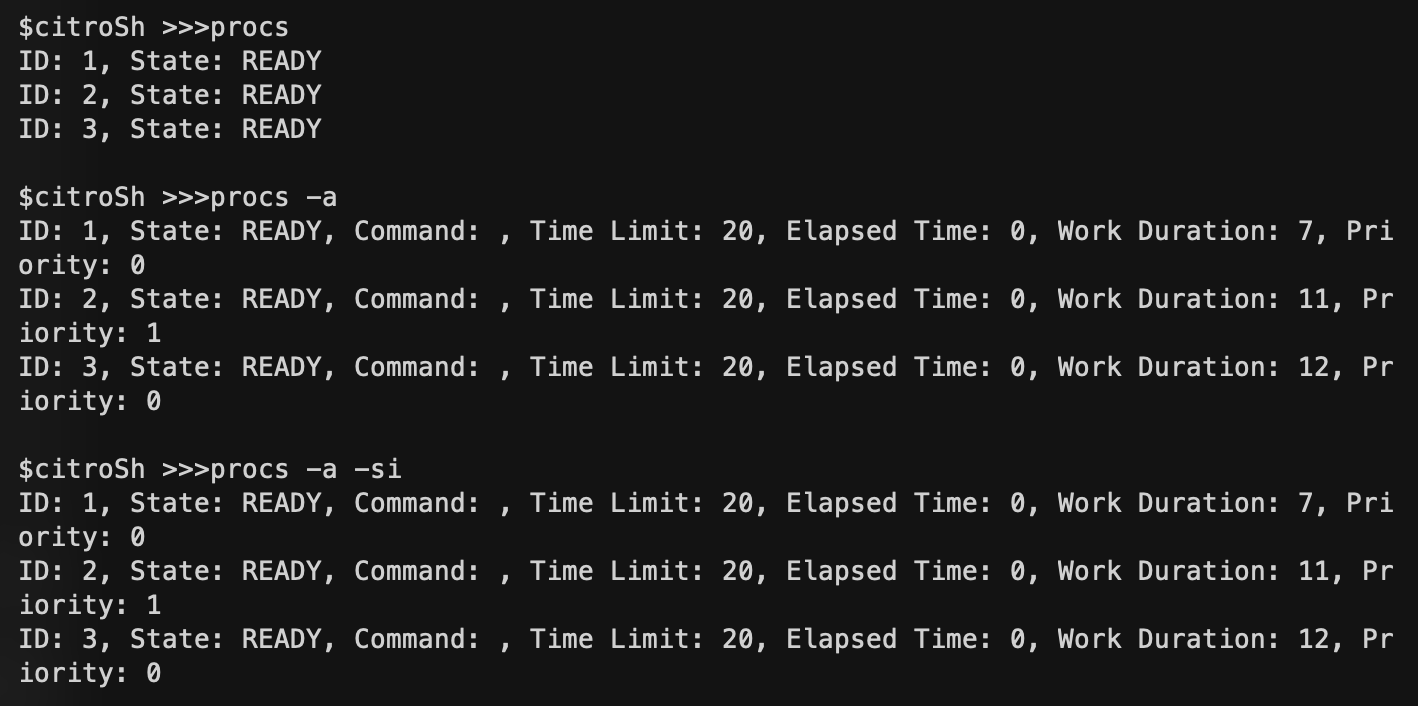
1. If the process requires I/O access or is subject to an external OS request, its state changes from **run** to **wait**.
2. If the I/O operation has been completed by the process that needed it, the process state changes from **wait** to **ready**.
3. If the process state changes from wait to ready, the state changes directly to **run**.
4. If the process terminates, it is removed from the list of managed processes.
5. If the process has reached the limit of the time set by the shell, its state changes from **run** to **ready**.

Here is what our scheduler does:

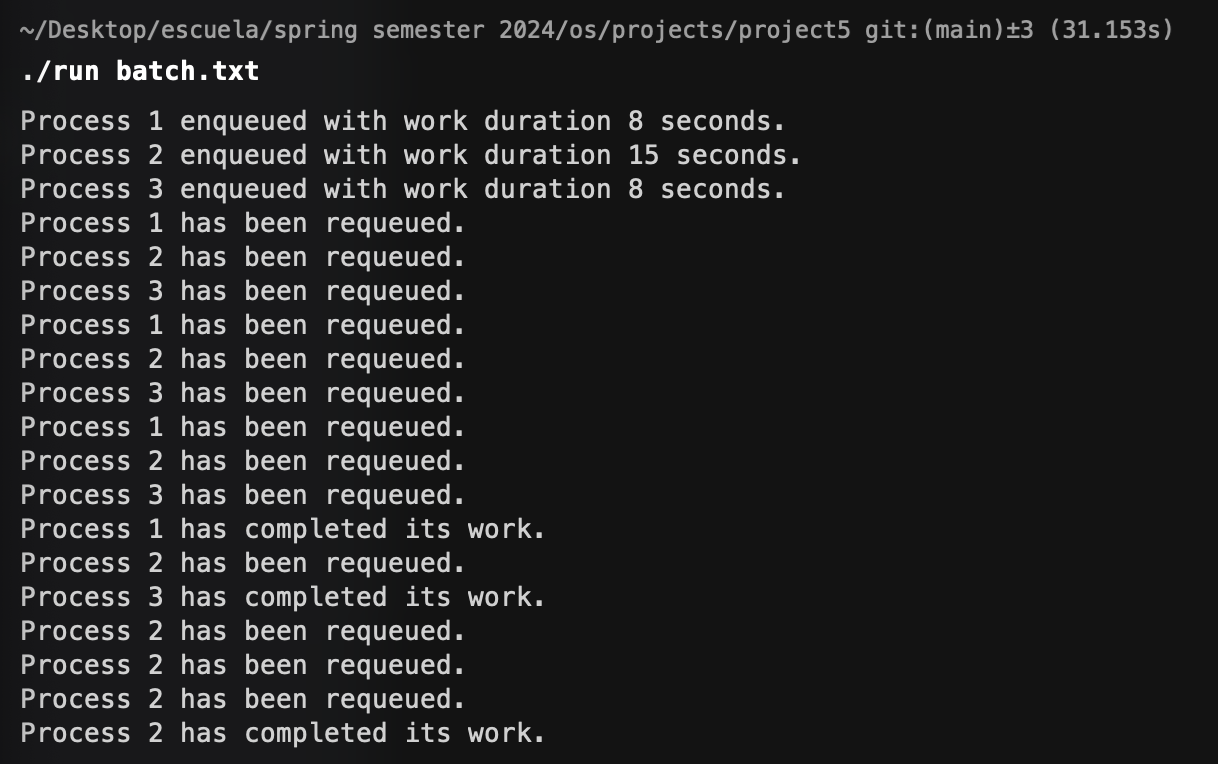
1. **How are you implementing the queue of processes?**
   1. The queue of processes is implemented using the **std::queue<Process\*>** data structure. Each process is represented by a pointer to a **Process** struct, allowing for efficient insertion and removal of processes from the queue. Processes are added to the queue when they are ready to be executed, and they are dequeued when it’s their turn to run.
2. **How is the scheduler assigning a reasonable share of the CPU to each process?**
   1. The scheduler employs a round-robin scheduling algorithm, which assigns a fixed time quantum to each process for execution. After a process consumes its time quantum, it is deleted, and the next process in the queue can run. This ensures that each process receives a fair share of CPU time, preventing any single process from abusing the CPU.
3. **How do you maximize efficiency?**
   1. Efficiency is maximized by utilizing the round-robin scheduling algorithm combined with thread-based execution. When one process is waiting for I/O or its time expires, the scheduler immediately switches to the next process in the queue, minimizing CPU idle time.
4. **How do you ensure a quick reaction to user input?**
   1. User input is handled asynchronously in the main loop using **getline(cin, input)**. This allows the program to promptly respond to user commands without blocking or delaying the execution of other tasks.
5. **How do you ensure that the same job takes approximately the same time when repeated?**
   1. The work duration of each process is randomly generated between 5 and 15 seconds. This randomness ensures that the same job may take different amounts of time when repeated, adding variability to the system and preventing predictability.
6. **How do you minimize the overhead?**
   1. Officially updating process dates only when necessary minimizes overhead. States are updated primarily in the **updateProcessState()** function based on each process's current state and progress. Additionally, pthreads are used for concurrency, reducing the overhead associated with thread management.
7. **How do you account for efficient resource utilization?**
   1. Resource utilization is optimized by efficiently managing process execution using the queue-based scheduling approach. Each process is executed for a fixed time, ensuring Fair CPU allocation will prevent excessive context switching and overhead.
8. **How do you avoid starving a process?**
   1. Starvation is avoided by using a round-robin scheduler, where each process gets an equal chance to execute within a fixed time. Even if a process before priority is repeatedly queued, it will eventually get a chance to run, preventing starvation.
9. How do you ensure that the shell does not collapse under increased load, but its **performance is gradually degraded?**
   1. The shell’s performance is maintained under increased load by using preemptive scheduling and limiting the time for each process. As the load increases, the time may be adjusted to ensure responsiveness while preventing excessive system load. Additional optimizations, such as process prioritization or load balancing, could be implemented to enhance performance further.
10. **Is your scheduler preemptive or cooperative?**
    1. The scheduler is preemptive, employing a round-robin scheduling algorithm with a fixed time. Processes are preempted after consuming their allocated time quantum, allowing the scheduler to switch to the next process in the queue.
11. **Provide the process queuing diagram.**
    1. | Process 1 |
    2. | Process 2 |
    3. | Process 3 |

This diagram represents a simple queue of processes, where each process waits for its turn to execute. The scheduler selects processes from the front of the queue, ensuring fair CPU allocation and execution.

**Interactive Mode**

1. Use the command interpreter to launch several processes.
   1. 
2. Display all the processes and their states
   1. 
3. Display all the information about the process
   1. 
4. Modify a process priority
   1. 
5. For step 2, 3, and r above, implement two modes, minimal and detailed information. The modes will be invoked by adding appropriate arguments to the command.
   1. 

**Batch Mode**



GITHUB REPO: [HERE](https://github.com/angel-vlzqz/Operating-Systems/tree/main/projects/project5)