Drake Tetreault
ID 35571095

CS 145B – Project 1

Final Document

Introduction

The goal of Project 1 is to simulate a simple operating system that is capable of running a number of processes simultaneously. While running, processes may request access to named system resources or IO devices, both of which kind of request may or may not be possible in the current state of the system. When a process requests access to a resource which is not immediately available, the simulated operating system will block that process without stalling the rest of the system. The system chooses which process to run based on the processes' priorities, where processes with the same priority are granted CPU time based on a round robin scheduling algorithm.

The deliverable requirements are:

- A previously submitted preliminary document.
- This revised final document.
- The source code, submitted with this final document.
- A meeting with the class TA to demonstrate functionality.

Although I worked alone, I chose to implement the additional group work, including:

- Variable resource amounts for R1-R4.
- An I/O resource.
- Command error checking.

In addition to the required shell commands, I also implemented the following commands to allow the user to see additional simulator state:

- "show-proc" shows details about one process or an overview of all processes.
- "show-res" shows details about one process or an overview of all processes.
- "debug" displays debugging information about the ready list and resource waiting queues.
- "help" lists all available commands and explains how to invoke them.

Data Structures

The data structure equivalent to a process control block in my program is the Process class. This class stores immutable data about a process in the system, namely the process' name, parent process, and priority. It also stores mutable state, which includes:

- A linked list of all direct descendants, stores as a reference to the head node of the list.
- A status, which can have the values Running, Ready, and Waiting.
- The process' position in the ready list, represented by a reference to a node in the ready list. If the process' status is Waiting, this node is null.
- The process' position in a waiting list, represented by a reference to a node in a waiting list. If the process' status is not Waiting, this node is null.
- The name of the resource on which the process is currently waiting.

- A record of the amount of each resource the process is currently holding. This allows for the simulation to release the resources held by destroyed processes.

The data structure equivalent to a resource control block in my program is the Resource class. This class stores two pieces of immutable resource data, namely the resource name and the total amount of acquirable resources of that type in the system. In addition, it stores the number of resources of that type currently available and a linked list of processes that are waiting for access to the resource, represented by a reference to the head node of the linked list. Because I implemented non-unit resources, each node in the resource's waiting list stores both the name of the resource and the amount requested.

Although I implemented the IO resource extension, I did not create a separate class for IO resources. Instead the IO resources is represented by a normal instance of the Resource class with name "IO" and total amount 1. In addition to requiring very little additional code to implement, this decision means I could trivially increase the number of IO resources available in the system if I wanted to.

The ready list is stored very simply as references to three linked list nodes which are the head nodes for the round robin queue corresponding to one of the three priorities. Processes and Resources are each stored in their own dictionary data structure, which maps from process and resource names to the actual object instances for efficient lookup.

System Architecture

When implementing my program I decided to divide the various responsibilities into the following components:

- The CommandRegistry class acts as a repository for all the available commands defined in the system, and may be queried for a command given the command name. It discovers commands using reflection at program startup. By using reflection instead of requiring explicit enumeration, I am able to easily add new commands with no changes to the rest of the system.
- The MessageBoard class lightly couples classes which emit commands from classes which receive commands.
- The Dispatcher class translates user input into system-recognizable commands. It does so by retrieving the appropriate command from the CommandRegistry, settings its parameters according to the user input, then notifying the rest of the simulation about the command using the MessageBoard.
- User interaction is the responsibility of the IInput and IOutput interfaces. The IInput interface allows the program to interact agnostically with both terminal and text file input sources, while the IOutput interface functions similarly but for output streams.
- The Simulator class implements the actual simulation logic. It stores the ready queue and process and resource lists. It responds to most shell commands by modifying the state of the simulation, and schedules the currently running process after every command.

Drake Tetreault
ID 35571095

Test Cases

I verified my implementation gave the same results as the provided test file. I then walked through several different test scenarios by hand and verified my program produced the expected result.

Code

I have included my code and the Visual Studio project which can be used to compile it in the archive file in which this final document was located. In terms of grading the correctness of my implementation, the most important file is Simulator.cs, which contains the bulk of the project’s simulation logic.