Operating System Documentation
Gavin Alberghini
CMSC 312 -- 12/1/2018

1. Requirements

1.1 Software Requirements

Java Runtime Environment

1.2 Hardware Requirements

None

2. Running The Program

2.1 How To Launch

To run the operating system navigate to the project folder and double click the executable jar file.

2.2 Operations/Commands

2.21 Rand(x) - Creates x amount of randomly generated processes and adds them to the operating system's waiting for memory allocation queue.

2.22 Run(x) - The operating system runs x number of cyclical steps.

2.23 Out() - Displays the current Scheduler, CPU, and Graphical information. This information updates every cycle on the operating system.

2.24 MemOut() - Displays the current RAM and Virtual Memory frames in the main display window of the GUI. This information updates every cycle on the operating system.

2.25 Quit() - Terminates the operating system and GUI window.

2.26 LookUp(x) - Displays process information of process with PID x inside the OS.

2.27 SchTwo() - invoke scheduler two instead of scheduler one.

3. Design Decisions

3.1 Scheduler

The Scheduler.java class in this OS implements a multilevel feedback queue used to organise processes on their way to the CPU. On an implementation level the Scheduler.java class contains several waiting queues along with a nested MultiLevelQueue

class. This MultiLevelQueue contains three levels (foreground, intermediate, and background) and can promote/demote processes between them. The foreground queue is organized by process priority, the intermediary queue is organized by shortest job first (SJF), and the background queue is organized by shortest time remaining first(STRF). The second scheduler in the system contains different methods of sorting each level but maintains the MultiLevelQueue architecture. When the program launches, you can enter the command listed in 2.27 to invoke the second scheduler.

3.2 Memory

The memory implementation is divided into three main sections. At the lowest level the Frame.java class represents the smallest unit of memory within the system (1MB). Frames can be occupied or free, contain their physical address, and hold the pid of the process that occupies it. The Memory.java class represents a collection of frames making a more complete memory structure such as RAM where multiple frames can be allocated and freed. Finally the MMU.java class contains both the RAM and the virtual memory, along with the memory map and cache. The MMU.java class organizes the highest level operations on memory. Paging, Virtual Memory, Cache, and Registers are all implemented within this memory hierarchy.

3.3 Inter-Process Communication

Two types of inter-process communication were implemented in this project. The Mailbox.java class contains five data structures (one for each process type), containing a lock to restrict access and a shared value for read/write operations. Intuitively, when a process is created it is given the key to the mailbox of the same job type. Then a process will read the current mailbox value and write over it afterward whenever asked to perform a communication task. The other type of inter-process communication was implemented as pipes between a parent and child process. The Process.java class contains an inner class Pipe which establishes a link between two processes with a pid reference. When built, pipes also generate keys and locks for the two attached processes to insure private access. When commanded to communicate a process will capture and record its companion process' message and can call up that information later in the run.

3.4 MultiThreading/CPU

The operating system runs on a simulated dual core CPU where each core utilizes four hardware threads. This is realized with a CPU.java class which contains two Core.java objects that also in turn contain four HardwareThread objects. The CPU.java class works at the highest level of interaction between the CPU, scheduler, and remaining operating system components. It will organize processes that are running and push them down to each core in a set of four processes. Core.java will then assign each of these processes a HardwareThread object that will do the actual 'work' of the process. The cores are designed in a

way such that each HardwareThread can run its process concurrently using the Callable interface. Once all processes in a set have completed their operations the core will recognise, evaluate, and push updates back to the CPU level. From there the CPU will decide the next fate of the given processes.

3.5 Deadlock Avoidance

Deadlock avoidance is realized during the memory allocation phase of a process' life. A process will not be able to leave the waiting queue until it is able to allocate all of the needed resources to run. If a process has acquired some resources but cannot acquire another, it is obligated to relieve all resources in its possession. The three types of resources in the system are memory, generic resource one, and generic resource two.

3.6 Parent-Child Relationships

When a process is generated in the ProcessGenerator.java class for every instruction in the textblock, the corresponding process has a percentage chance to spawn a child. Children are independent in memory requirements and instructions, being able to also spawn their own children in turn. The PDT.java class tracks the child PIDs for a process, creating an implicit tree. Also, during child creating a pipe is made (as stated in *3.3*) that connect the parent and child. When a process terminates, cascading termination removes all lower level processes from the operating system if they still are active.

3.7 GUI/Operational Structure

The GUI is realized using the Java.swing API and provides an interactive control screen where various statuses can be displayed. The GUI is divided into four main output screens, four resource visualizations, a command line, and a command history. Commands as shown in *section 2.2* are entered into the command line.