|  |
| --- |
| University of Windsor |
| 60-330 Project |
| A Standalone Operating System Simulator |
|  |
| **Jason Bennett (bennettj@uwindsor.ca), George Kyrtsakas (kyrtsakg@uwindsor.ca), Halen Whiston (whistonh@uwindsor.ca), Ping Yong (yongp@uwindsor.ca)** |
| **8/20/2013** |

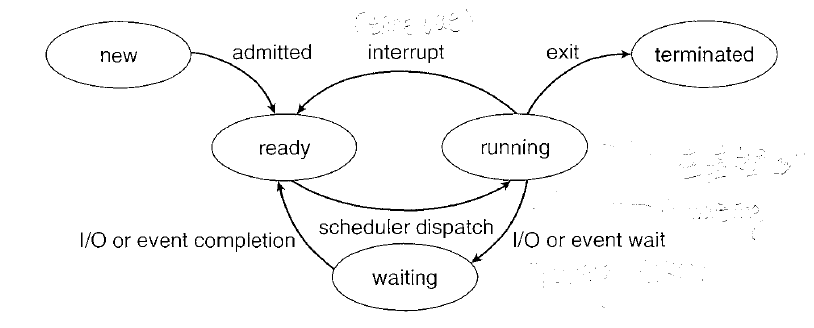
|  |
| --- |
| This documentation is to be submitted with our operating system. It contains information explaining the functions and concepts taught in the course and how they are implemented into our application. |

60-330 Project Documentation

# Introduction

For our project, we have developed a standalone java application that simulates the basic functionality of a computer operating system. The simulation used within the application applies concepts taught in the class 60-330 “Operating System Fundamentals,” as well as the concepts described in the original project requirements. Please keep in mind that this application is designed to simulate the functionality of an operating system. This means that all resources and memory that is used by the application is artificial and only exists to portray operating system concepts. We have given our homemade operating system the name, “2P2E,” which stands for “Two Printers to Eat.”

# 5-State Process Model

The application utilizes a standard 5-State Process Model and follows the basic flowchart accordingly. This model is used to describe the states of process within our operating system. Basically, at any given point in time, all currently active processes have a state. Processes that are created are placed in their “new” state, where they are then changed to a “ready” state when they have access to the required resources/devices. Processes may begin “running” once this condition is met, however, they will be changed to their “waiting” state should a certain condition not be met (such as an I/O event or required resource that is locked by another process). Once a process has completed its task, it becomes “terminated”, thus ending the process’ activity.

The processes are created using the “Process.java” class within our application. This class contains all attributes or event information related to processes. To create a new process, a new pid value must be passed into the “Process” constructor, as well as a process name, priority level, submission time, burst cycle, and list of devices/resources required by the process. Once these values have been passed, a process is created and put into the “new” state. Depending on the resources that are required and available, the process will be able to move to further states. These states are determined using one of the 5-State Queues.

# Job Submission

Jobs are submitted through the command line interface. Details of the job such as burst count, RAM required, priority and others are passed as arguments to the Process constructor. There are some preset processes that are hard coded into the simulator for the user’s convenience.

# Machine Architecture

## RAM Model for User Processes

RAM is modelled using an array of integers where each integer represents an 8KB page in memory. When a process requests a page, the next available array element is reserved by changing its value from 0 to 1.

## CPU Functionality

Potato.

## I/O Devices

I/O devices are modeled in the device class as objects that can be locked to a process. There are 10 built in I/O devices numbered 1 through 10.

## System Clock

The system clock is more of a burst clock, which is our chosen unit of time. Every time it increments a process can move between queues based on the scheduling algorithm.

# Process and Resource Tables, Queues and Management

## Resource Management Tables

Resources are stored in Resource objects created by the main system. Each unique resource is stored in a static array, and assigned a unique port ID in the format of “R-X”, where “X” is the index of the resource in the array. These Resources can be accessed using the “findResource()” function, which is used by the process to pull all the available resources. The process can then lock the resources using the “setLock()” function, and release them when completed using the “releaseLock()” function. In addition to this, the resources and their information can be printed to the screen in a nicely formatted table using “getTableInfo()”.

## Device Management Tables

Devices work in a way exactly the same as the Resources. Devices are stored in Device objects created by the main system. Each unique Device is stored in a static array, and assigned a unique port ID in the format of “R-X”, where “X” is the index of the Device in the array. These Devices can be accessed using the “findDevice()” function, which is used by the process to pull all the available Devices. The process can then lock the Devices using the “setLock()” function, and release them when completed using the “releaseLock()” function. In addition to this, the Devices and their information can be printed to the screen in a nicely formatted table using “getTableInfo()”.

## 5-State Queuing Model

The 5-State Queuing Model is the representation of the 5-State Processing Model in code form. This is demonstrated by creating five objects of the LinkedList class, one for each of the Process States, then storing and moving processes from list to list as necessary. This is demonstrated in Queue Management below.

## Queue Management

The Queue Manager class is used to move processes from queue to queue. It stores five static lists, one for each of the five required states, and runs scheduling algorithms from the Non-Pre-emptive Scheduler class to move processes efficiently to the needed queue. The Queue Manager class is also used to ensure processes have the required devices and resources before running, and ensuring processes will release locks if they cannot run, or are terminated. When needed, the Queue Manager can display a table of each queue, as well as which processes exist in which state queue at the time.

# Scheduling

## Short Term and Long Term

All processes in our operating system are scheduled for both the short-term and long-term. When processes move into the ready queue, for instance, they are sorted by a long-term schedule so that the process to leave the queue first will be at the front. Short-term scheduling is used here to move processes out of the ready queue, as soon as they are able to run. Both of the scheduling methods are addressed within the NonPreemptiveScheduler class, whose functions not only place objects into the queues, but also utilize sorting algorithms to do so.

## FCFS, SJF, Priority

Our operating system implements three different scheduling algorithms, which the user can use to run programs. The scheduling method (FCFS, SJF, and Priority) which the user would like to use must be selected prior to running the simulation, although the default is set to FCFS. The input given by the user is then passed to the NonPreemptiveScheduler class, which will store the information, and then change the way that the processes are sorted into the queues based on that information.

## Deadlock Avoidance/Prevention

In order to avoid causing deadlocks in the simulation, we used a version of the reader-writer problem when implementing our wait queue, and handling the manner in which processes access devices and resources. Once per cycle, each process in the wait queue will attempt to obtain the resources and devices that it requires in order to run. Should it succeed in doing so, it will lock all of them for itself, and proceed into the ready queue again. However, if the process fails to obtain exclusive access to all of its required devices and resource, it will not try to lock any of them for itself in that cycle. This will prevent a process from withholding resources and/or devices from other processes in the wait queue, and makes it so that processes never deadlock.

# User Process Synchronization and Simulation

## Simulation of Single Process Execution

Processes in our application may enter the “running” state, in which case, are then transferred to the running queue. Processes in the running queue have access to all of the necessary resources and are executing. The structure of the running queue can be examined within the “QueueManager.java” class. Processes continue in this running state until they are finished a job or are terminated. In our application, only one process may be actively running at a time.

## Multiprogramming Support and Management

Support for basic multiprogramming is implemented in our application. Based on the textbook, this support ensures that the CPU is utilized during most of the operating system’s total runtime. For example, if a process is busy waiting on resources to become available, it is sent to the waiting queue until the required resource becomes available. During this time, another process is scheduled to be used with the CPU. This technique is intended to maximize CPU usage and prevent wasted time and resources. The class that manages this form of multiprogramming is the “QueueManager.java” class. All processes in the waiting queue will not use up as much CPU as a process in the running queue, so any activity that is done by a process moved to the waiting queue will be replaced with a process that has access to all of its resources (a process in the “running” state).

## Synchronization and Critical Sections

The application uses a form of synchronization to ensure processes have permission to access resources. Basically, a reader-writer lock is set on resources that are accessed by processes during run time. Any resource that has a write-lock on it means that it cannot be accessed by any other processes. Processes attempting to access a locked resource are moved to the waiting queue until the resource becomes available. When a process is finished utilizing a resource, it unsets the lock and the resource may be used by another process. This form of synchronization ensures processes do not attempt to modify the same resource at the same time.

## Peterson Solution to Critical Sections

The “Process.java” class has Peterson’s solution utilized in a method; however, the overall nature in which the application was designed prevents proper implementation of the algorithm. The algorithm is intended to allow shared resources to be accessible by multiple processes. Our application prevents us from getting the most use out of this algorithm because it only permits one process to access any one resource at a time. A resource that is accessed by a process becomes locked and therefore, inaccessible to other processes. Assuming that our application does support shared resources, the Peterson solution would have made it so that any two processes could access the same resource. It does this by utilizing a “flag” Boolean variable in the process class, as well as a “turn” variable to mark the turn of the process to access the resource. If the “flag” is true and the “turn” of the process is set, then the process may access the resource without any conflict.

# User Interface and Reporting

## Command Line Interface

The command line interface is implemented in the numbered-list style, giving the user easy access to all our functions while only using a small amount of space on-screen.

## Reporting Summary Statistics

After every burst cycle, all five queues are printed very nicely in a table on the screen along with the burst cycle count. Most of this work is taken care of in the Device and Simulator classes.

## Comparative Statistics for Scheduling

At the end of the simulation run the average wait time is printed on screen so the user can compare the different scheduling algorithms for that given set of processes.

# Conclusion