Kennesaw State University
College of Computing and Software Engineering

CS3502-01

Operating Systems - Dr. Patrick Bobbie

Operating System Simulation Phase 1: Final Report

Authors:
Weston Ford

May 26, 2016

Contents

1 Introduction

This document is created with the intention of providing details of the design, function, and implementation of each individual subsystem (also referred to as modules) within the associated operating system created for the course CS3502 - Operating Systems, Kennesaw State University College of Computing and Software Engineering, Spring 2016.

The core of computing is predicated upon the interdependence between functional physical components and the abstract control structures which instruct and regulate the actions of those components. These control structures interact to form the superstructure known as the operating system. While the potential complexity of an operating system is unbounded, its composition can be reduced to a handful of components who each play a role in the overarching purpose of facilitating interaction among hardware and between hardware and software.

Within this project exist multiple subsystems each with various compartmental and general roles and responsibilities. As this system is entirely virtual, many of the decisions made when designing these modules are informed by decisions made when implementing the virtual hardware with which the operating system interacts. This report will be separated into two sections. The first section of this report will be relegated to discussing the operating system while the second section will be dedicated to discussion about the virtual CPU. The operating system operations covered in this report include, with some overlap, the following items:

- Memory Management
- The Driver
- Loading
- Scheduling
- Dispatching

While discussion for each item will be relatively specific to that item, it should be expected that there will be references made in earlier sections made to concepts discussed in later sections (and vice versa). In section two this report will explain the structure and implementation of the virtual CPU with attention to specific functions of the CPU. The entire report will be written with an emphasis on implementation specifically with the Java programming language and Java API.

2 The Operating System

Any operating system must be capable of accessing instruction data and enabling its execution. At minimum this requires the ability handle input and output data, the ability to access and modify memory, and the ability to send and retrieve data to and from the CPU. First, this report will discuss how memory is handled. It is important to note that there is no explicit Memory Manager class to which all methods and virtual memory exists within. Most memory management responsibilities are implemented as methods within the RAM, DISK, PCB, and Register classes which are evoked as needed by other modules within the program.

2.1 Memory Management

The memory management system is an umbrella term which, in this report, will refer to the routines, classes, and methods which interact with memory as well as the actual virtual hardware called RAM and Disk. As this is a simulated operating system running on simulated virtual hardware effort was made to not deviate from how data is passed and stored within an actual system. In general, an attempt was made at to ensure that, at least superficially, information "looked" from the perspective of the component handling it as it is expected to look in an actual system. What this means is that information is stored directly as strings of hexadecimal or binary characters. The virtual Disk can contain 2048 words, each word as a size of 32 bits. RAM can store up to 1024 of those eight character hex strings.

One challenge with Java is determining how information can be passed between classes. As there is no way to explicitly pass objects by reference it was decided that the RAM and Disk memory would be created as static classes. These classes posses a private static String array called memory which is accessed through public methods within the RAM and Disk classes. Both classes possess methods to store and load based upon various different passed indexes and value types. This ensures that they can receive and process indexes and values as hexadecimal and binary strings, as well as decimal integers while still ensuring that data is stored as an eight character hexadecimal string and that indexes are compliant with Java's standard array indexes. Other methods allow for indirect saving and loading given base register and program counter values. Each class possess a "pointer" variable which will equal the next available open index.

Other data structures exist to store and handle various objects of certain types in different stages of processing. The PCB class contains a static ArrayList whose purpose is to store process objects. Process objects are objects which contain all of

the values of a process loaded from a program file. Process objects are discussed within the loader subsection. Other structures utilized are two queues called ready and waiting which hold process objects which are ready to execute from a new state or objects which are ready to execute after an interrupt and context switching. Lastly, the class Registers possesses a string array of size sixteen which acts as the register memory for the CPU. Each "register" within the Register string array must be a 32 character long string which represents binary information.

2.2 The Driver

This is one of the simplest components of the program and, as a result, there is little to discuss. The driver controls the primary control loop of the operating system which ensures that the scheduler, dispatcher, and cpu.execute() commands occur continuously until the entire program file has been computed and there are no more pending processes. The driver instantiates the various objects (specifically the loader, scheduler, dispatcher, and cpu(s)) to their appropriate values (when applicable) and is the location where the path of the program file to be processed is declared. Within the driver are also tests used to gather data on operating system performance.

2.3 Loading

2.3.1 The Process Class

The Loader module is the entry point for the Process class which is an important attribute of this operating system and, as it will be referenced throughout this subsection and certainly most subsequent sections, is deserving of a formal introduction. The Process class is an object which is intended to represent an individual process's PCB. Each process object has upon creation, at minimum, a record of the index of its starting location within Disk, the number of instructions specified by its control card, the size of its I/O buffers, and its process ID all stored as decimal integers. It also contains additional variables initialized to differing default values which are relevant to functionality discussed within later parts of this document (Ram address and process state within the scheduler, program counter and a Register object discussed within the dispatcher and the CPU sections).

2.3.2 Parsing, Loading, and Process Creation

The loader is responsible for parsing information from the program file it receives from the driver and ensuring that it is loaded onto Disk storage. Additionally, it must create a record of each of the process' unique attributes such as location in Disk, total number of instructions, input, output, and temp buffers, and various other information utilized by this operating system.

Its responsibilities can be referred to as file parsing, disk storage, and process recording. Each of these responsibilities overlap but they generally occur and are completed as a loop of parse, store, record which occurs mostly in order as the file is being processed. The entire series of events is initiated upon loader object creation and no additional methods calls are necessary from the driver to complete file processing.

In order to parse the file, the loader's constructor is passed a File object containing the program file's file path. Within the constructor a standard Java API scanner object is created and that scanner is passed along to the processFile method. Various methods are used to manipulate the program file strings in order to obtain information from the various control cards. This control card information is utilized by the createProcess method in order to obtain JOB and DATA information so that it can, firstly, assign values to variables which will be passed as parameters to a new process object and, secondly, provide the number of read/write operations to the called saveToDisk method that is responsible for saving all instruction and information data onto Disk through consecutive read/writes operations.

In order to better handle control card interpretation, a confusingly named enumerated class called PType is used. The processControlCard performs string comparisons to ascertain card type and returns the corresponding enumerated value. This value is checked within the createProcess method to determine whether the subsequent information should be interpreted using the JOB, DATA, or END formats.

The end result of one pass of the processFile method within the loader results in the scanner object being iterated through the program file to the beginning of the next JOB control card, all instructions and data being saved to the static Disk array, and a process object being created and added to the static PCB ArrayList. Loader continues through this loop until the scanner reaches the end of the file and there are no more processes to parse.

2.4 Scheduling

Within the scheduler lies the majority of intensive procedures by which the efficiency, resiliency, and dynamism of our machine is defined. Its functions are entirely relegated towards the maintenance and transference of information within and between storage components. We can deconstruct these tasks into relatively unique components which can be classified as belonging primarily to one of three categories:

- Initialization - Introduction of new instruction sets into main memory.
- Maintenance - Resource grooming and upkeep.
- Termination - Removal of completed instruction sets.

As this operating system is built to support multiple CPU's there are two categories of scheduler noted as either the long or short-term schedluer. However, it is important to note that the code only differentiates the short-term scheduler by name, with the scheduler class being labeled simply "Scheduler" possessing the ability to fulfill both roles. As the long-term scheduler is both activated first, implemented first, and performs a greater role within the OS we will discuss the details of its construction first.

2.4.1 The Long-Term Scheduler

The long-term scheduler, while never mentioned by name, can be considered to occupy the majority of the real estate within the scheduler class. Once the loader has successfully populated the PCB with process objects and loaded the corresponding instructions from programFile into Disk it returns to the driver, and continues into the primary three-method (or two-method within our multi-thread implementation) running loop of the operating system. This first function is schedule() which initiates the central control structure of the scheduler class. Schedule()'s first task is to populate RAM from Disk so that process dispatch can occur.

The default scheduling algorithm is a hybrid priority-FIFO algorithm. It will traverse the PCB list and find the process with the lowest priority number (In this case, lower priority numbers are high priority) which has not already been loaded and determines if there is space in RAM with the isSpace() method. In the event of priorities with equal priorities, lowest index entrant is loaded. The algorithm terminates at the first point in which a process cannot be loaded due to RAM constraints. This algorithm does not attempt to maximize occupied RAM by searching for smaller processes in the event of that one is too large. There is also a pure FIFO scheduling method present within the class for testing purposes.

```
public void schedule() throws MemoryException{
int priority = 1000;
int index = 0;
while (index != -1){
        index = -1;
        for (int i = 0; i < PCB.memory.size(); i++){
                if (PCB.memory.get(i).getState() == PState.NEW){
                        if (priority > PCB.memory.get(i).getPriority()){
                                priority = PCB.memory.get(i).getPriority();
                                        index = i;
                        }
                }
        }
        if (index != -1){
                if (isSpace(PCB.memory.get(index).getNumInst() + Sum_Buffers)){
                        loadProcess(index);
                else
                        index = -1;
        }
        priority = 1000;
        }
shortSchedule();
}
```

Figure 1: Primary scheduling method

Once the highest priority process is selected and determined small enough to fit into memory loadProcess(int index)

2.4.2 Resource Arbitration

Most of the methods within the class Scheduler are dedicated towards determining how to effectively allocate resources and how best to monitor and curate them in order to prevent situations which can be lethal to our operations system. Creeping problems such as memory fragmentation can slowly corrode performance and hardware capabilities while thread interference can cause memory consistency errors whose effects can range from immediately disastrous to chronically fatal.