CS 3502 PROJECT - PART 1

Fall 2024

Axel Alvarez, Matt Crowley, Justin Hammit, Abdulmajeed Kabala, Josh Menefee, Tochi Okwudire, Chris Stropoli

Operating Systems
Section W01

Table of Contents

Abstract

The following report explores the approach to designing, implementing, and testing an experimental operating system. The methodologies and approaches will be explained and detailed as well as the problems and solutions encountered. The design was based in the Java programming language. The experimental operating system contains a Kernel and Loader wrapped into a Simulator module, that instantiates the objects and pulls the program files onto the disk. Further, the operating system contains a simulated single-core Central Processing Unit (CPU) that can perform various rudimentary functions as instructed by assembly code. It also contains a simulated Memory (RAM), and Disk Storage that are 2048 bytes and 4096 bytes respectively. The operating system also contains both a Long-Term Scheduler for process optimization and wait queues. Additionally, the Central Processing Unit has several sub-parts including the Process Control Bus (PCB), the Direct Memory Access (DMA), and Registers.

Introduction

Operating systems are the defining characteristic of modern computing and are what make computers much more powerful than they were in the past. The operating system is the functional manager of the computer, and it links the two sides of hardware and software to work together and produce faster and more efficient results. It does this by managing the computational resources of the system and sending those resources to where the system deems it can best be used. Other than resource management, operating systems are also in charge of managing memory and input/output (I/O) operations for the system. Overall operating systems are made to ensure the stability and performance of a computer.

This project takes the challenge of designing, implementing, and testing an operating system in the Java programming language. The system created is meant to simulate the same functionalities found in other operating systems on all other computers. This system features a Kernel that is tasked with instantiating objects and overseeing system-wide tasks, a single-core Central Processing Unit(CPU) that takes in and executes assembly instructions, simulated memory(RAM), and Disk Storage for managing system memory, and both short and long-term schedulers in charge of handling the task prioritization. There are also the subsystems of the CPU that include the Process Control Block (PCB), Direct Memory Access (DMA), Registers, and Cache.

This report explores the steps of designing an operating system, starting with the approaches that were taken to design it. After that is an intricate look into how each of the modules mentioned earlier were implemented into the system and how they interact with each other together as a whole. Following this, is an explanation of the methods for testing and simulating the operating system. The report concludes with an analysis of the system's performance, focusing on the execution time and percentage of RAM used for a job, as well as highlighting areas for improvement. This report aims to show an understanding of operating systems and further knowledge of them by documenting the challenges and solutions that were faced while designing the intricate system.

Design Approach

Block Diagram

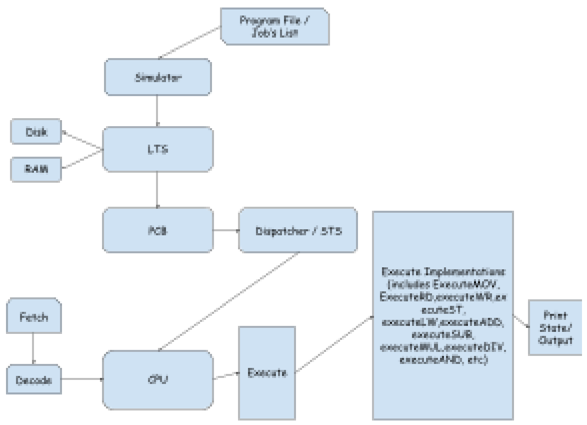

Program File / Job's List
Simulator
Disk
RAM
LTS
PCB
Dispatcher / STS
Fetch
Decode
CPU
Execute
Execute Implementations (includes ExecuteMOV, ExecuteRD, executeWR, executeST, executeLW, executeADD, executeSUB, executeMUL, executeDIV, executeAND, etc)
Print State/ Output

Diagram 1: Shows a visual representation of the simulator using nodes. It also shows the flow between nodes.

The simulation start's off with the Program file (Jobs.json). It contains all the jobs, data, and code related to them. The jobs are then sent to the Simulator node. The Simulator node creates instances of memory and CPU that the jobs get loaded into. After that the next node is the LTS. This node deals mainly with loading data from disk to RAM. and dealing with the PCB. After that, the PCB node has numerous getters and setters that help identify the jobs. The Dispatcher node is next and it deals with both the PCB and the CPU to see what actually needs to be sent over to the CPU. Now, it is time to explain the flow of the CPU node. With the data already being registered/loaded. It has to be fetched from the PCB (demonstrated with the Fetch node) and decoded to set up the register (demonstrated with the Decode node). After that, it proceeds to the Execute node, where a lot of switch cases are utilized. Next, there are multiple different implementations of the instructions. The next node in the flow would be the Print State/Output node which shows the output from the overall program file.

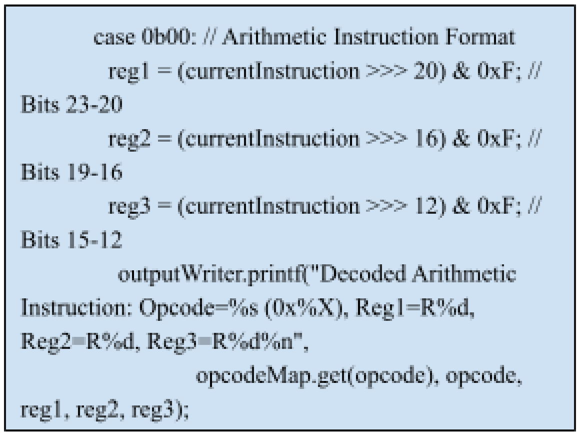
Module Implementation

The operating system simulation was designed to be split into seven different modules. Each module would function as a facsimile of its equivalent hardware component. This allowed each piece to be tested independently and for decreased coupling of the simulations. However, one of the key attributes of this design is that while the jobs are inputted as hex-based assembly code, the simulation has a translation layer that converts the code into integers, strings, and other types that are more readable and manipulatable within Java.

CPU Module Implementation

The CPU is the largest module in terms of code lines within the Operating System. It consists of a few sub modules that are as follows: the decode block, the execution block, the registers, and the ALU.

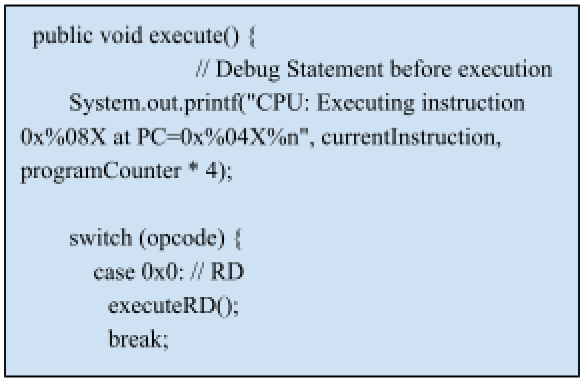
The decode block is implemented as a switch statement reads in the instruction type and opcode to determine the case. The cases are Arithmetic, Conditional Branch, Unconditional Jump and Input and Output. The Arithmetic, Conditional Branch and Input and Output each write to registers before passing the Job to the execution block. Here the Arithmetic case can be seen.



```
        case 0b00: // Arithmetic Instruction Format
          reg1 = (currentInstruction >>> 20) & 0xF; //
Bits 23-20
          reg2 = (currentInstruction >>> 16) & 0xF; //
Bits 19-16
          reg3 = (currentInstruction >>> 12) & 0xF; //
Bits 15-12
           outputWriter.printf("Decoded Arithmetic
Instruction: Opcode=%s (0x%X), Reg1=R%d,
Reg2=R%d, Reg3=R%d%n",
                      opcodeMap.get(opcode), opcode,
reg1, reg2, reg3);
```

Diagram 2: Shows a section of the Decoder in the Arithmetic Instruction case.

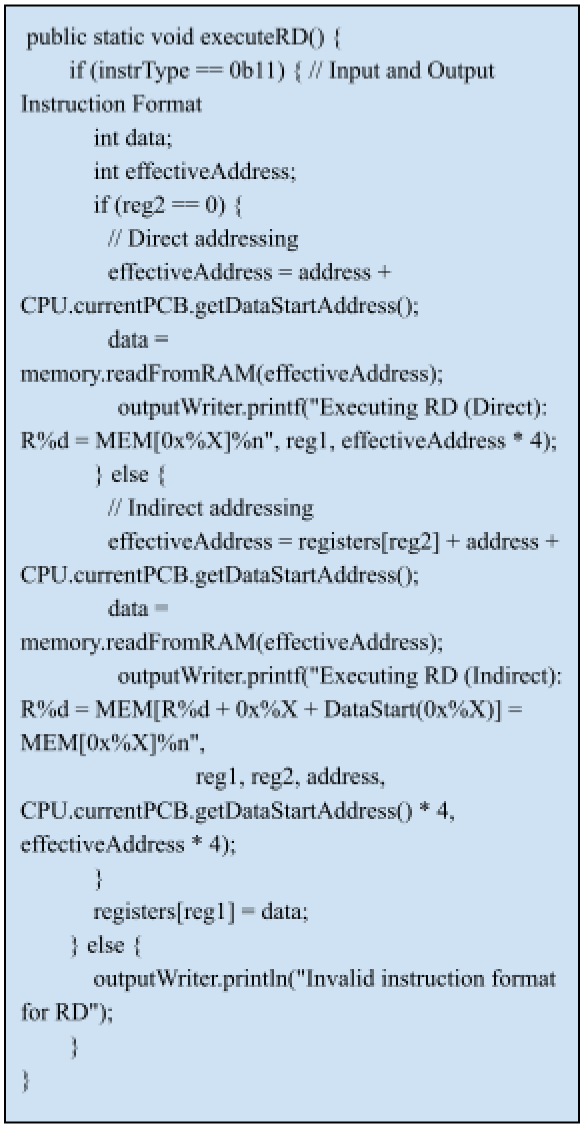
The execute block is a large switch statement containing all of the instruction types possible within the operating system. From basic math instructions to logic instructions like AND and OR to Halt and Jump.



```
  public void execute() {
                    // Debug Statement before execution
      System.out.printf("CPU: Executing instruction
0x%08X at PC=0x%04X%n", currentInstruction,
programCounter * 4);

      switch (opcode) {
        case 0x0: // RD
          executeRD();
          break;
```

Diagram 3: Shows a segment of the execute code for the Read instruction case.

The execute block then passes the process to the specific ALU block that is desired by the program where the operation is performed. In the case below it is the RD or Read instruction that moves data from Reg2 into Reg1.



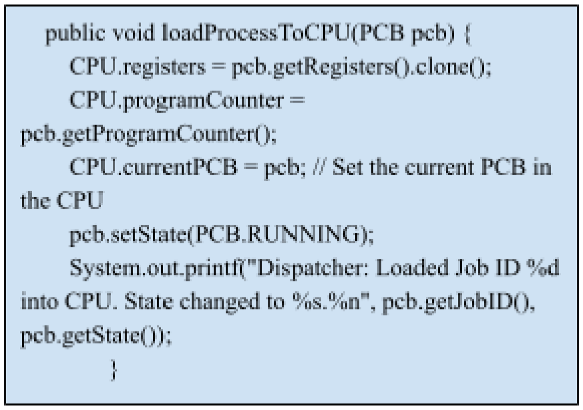
```
 public static void executeRD() {
      if (instrType == 0b11) { // Input and Output
Instruction Format
          int data;
          int effectiveAddress;
          if (reg2 == 0) {
            // Direct addressing
            effectiveAddress = address +
CPU.currentPCB.getDataStartAddress();
            data =
memory.readFromRAM(effectiveAddress);
             outputWriter.printf("Executing RD (Direct):
R%d = MEM[0x%X]%n", reg1, effectiveAddress * 4);
          } else {
            // Indirect addressing
            effectiveAddress = registers[reg2] + address +
CPU.currentPCB.getDataStartAddress();
            data =
memory.readFromRAM(effectiveAddress);
             outputWriter.printf("Executing RD (Indirect):
R%d = MEM[R%d + 0x%X + DataStart(0x%X)] =
MEM[0x%X]%n",
                      reg1, reg2, address,
CPU.currentPCB.getDataStartAddress() * 4,
effectiveAddress * 4);
          }
          registers[reg1] = data;
      } else {
          outputWriter.println("Invalid instruction format
for RD");
      }
}
```

Diagram 4: Shows a section of the ALU block code.

These ALU blocks are the vast majority of the CPU's code lines as each one has a different function. However there are still a couple of additional functions within the CPU including the resetCPU function and some of the functions used for testing such as the printState function. All of which should be self explanatory.

Dispatcher Module Implementation

The dispatcher is a very small module that contains just two functions. loadProcessToCPU and saveProcessFromCPU. These two functions change the state of the CPU from RUNNING to TERMINATED and vice versa when called. They also set the registers within the CPU for the program being loaded and reset the registers for the program that is being saved. Finally, it updates the program counter.

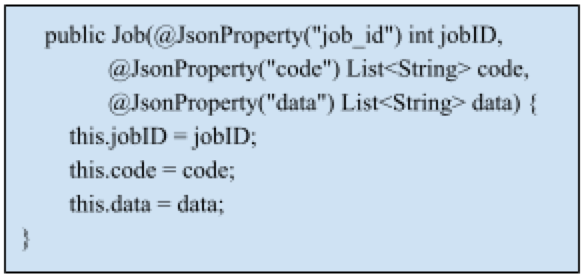


```
    public void loadProcessToCPU(PCB pcb) {
        CPU.registers = pcb.getRegisters().clone();
        CPU.programCounter =
pcb.getProgramCounter();
        CPU.currentPCB = pcb; // Set the current PCB in
the CPU
        pcb.setState(PCB.RUNNING);
        System.out.printf("Dispatcher: Loaded Job ID %d
into CPU. State changed to %s.%n", pcb.getJobID(),
pcb.getState());
            }
```

Diagram 5: Shows a segment of the Dispatcher Module

Job Module Implementation

The Job file is a simple translation layer object that takes in the data from the Jobs json file and saves them as a Job object. It scans through the json file and strips off the data that is unnecessary and the comments for user readability and then saves each of the pieces as self-evident variables that the simulator can more easily manipulate.

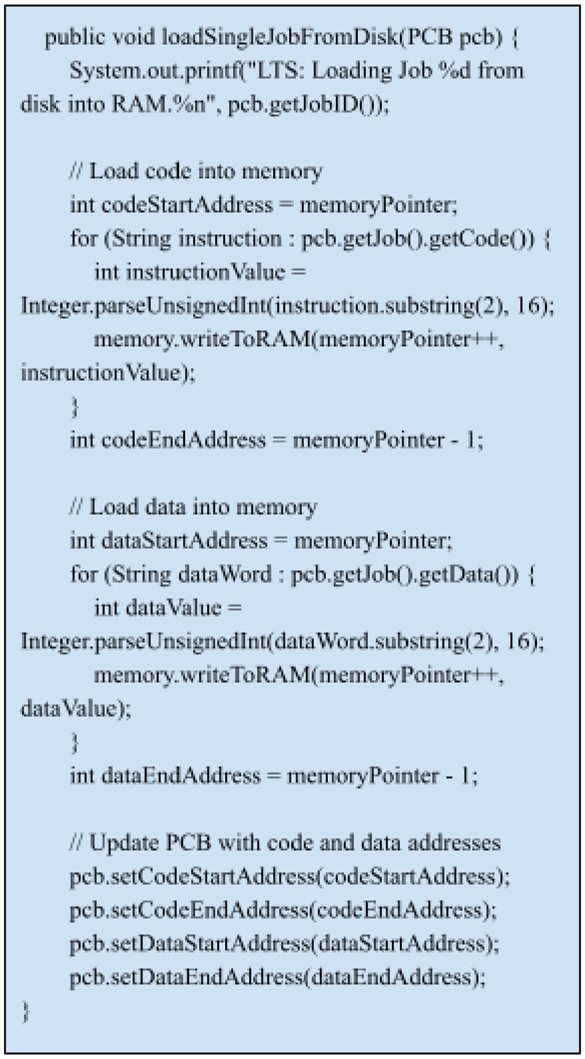


```
    public Job(@JsonProperty("job_id") int jobID,
            @JsonProperty("code") List<String> code,
            @JsonProperty("data") List<String> data) {
        this.jobID = jobID;
        this.code = code;
        this.data = data;
}
```

Diagram 6: Shows a segment of the Jobs Module.

Long Term Scheduler Implementation

The long Term Scheduler is one of the smallest modules as it only handles the manipulation of pointers for the locations of each job. It is simple a large function for loading a single job from disk and putting it into RAM with two reset functions tacked on at the end of the module. The loadSingleJobFromDisk function contains the majority of the code and starts by using a loop to scan through the job to find the instructions and skip the data where it then puts these instructions into a sequential list within memory. It then follows that up by doing the reciprocal of skipping the instructions and copying only the data from the job and putting that into a sequential list within the RAM. It then updates the Process Control Block with the starting and ending addresses of both lists.

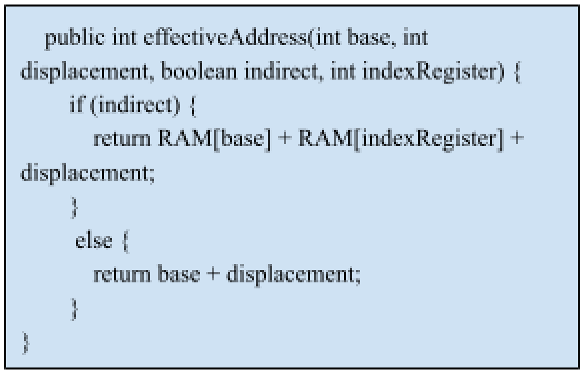


```
    public void loadSingleJobFromDisk(PCB pcb) {
        System.out.printf("LTS: Loading Job %d from
disk into RAM.%n", pcb.getJobID());

        // Load code into memory
        int codeStartAddress = memoryPointer;
        for (String instruction : pcb.getJob().getCode()) {
            int instructionValue =
Integer.parseUnsignedInt(instruction.substring(2), 16);
            memory.writeToRAM(memoryPointer++,
instructionValue);
        }
        int codeEndAddress = memoryPointer - 1;

        // Load data into memory
        int dataStartAddress = memoryPointer;
        for (String dataWord : pcb.getJob().getData()) {
            int dataValue =
Integer.parseUnsignedInt(dataWord.substring(2), 16);
            memory.writeToRAM(memoryPointer++,
dataValue);
        }
        int dataEndAddress = memoryPointer - 1;

        // Update PCB with code and data addresses
        pcb.setCodeStartAddress(codeStartAddress);
        pcb.setCodeEndAddress(codeEndAddress);
        pcb.setDataStartAddress(dataStartAddress);
        pcb.setDataEndAddress(dataEndAddress);
}
```

Diagram 7: Shows the main section of the Long Term Scheduler and how it edits PCB values.

Memory Module Implementation

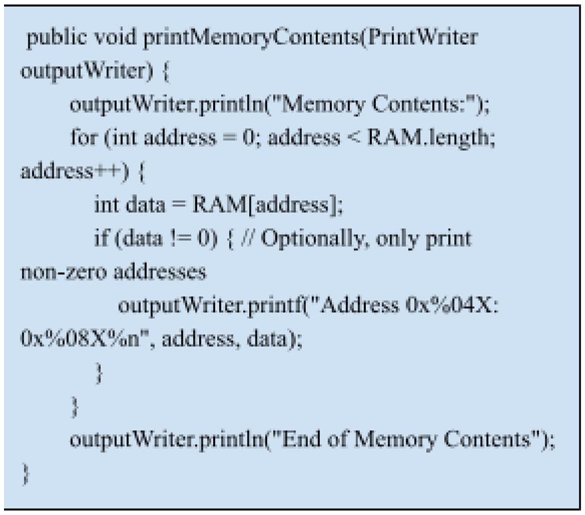
The memory module contains both the RAM and the Disk simulations. Both are implemented as large fixed arrays of 1024 and 2048 bits respectfully. There are twelve functions contained within the memory module. The most heavily used functions of this module are the respective read and write functions for both the disk and RAM. These are no more than getters and setters with extra steps and an exception catcher. The effectiveAddress function is worthy of note in that it is used to grab the index of the start of each program loaded into RAM. It uses a base variable to set the point where kernel memory ends and program memory can begin. Then adds the displacement and indexRegister to find the effectiveAddress of the program logic.



```
    public int effectiveAddress(int base, int
displacement, boolean indirect, int indexRegister) {
        if (indirect) {
            return RAM[base] + RAM[indexRegister] +
displacement;
        }
         else {
            return base + displacement;
        }
}
```

Diagram 8: Shows the function for the effectiveAddress within the Memory Module.

The printMemoryContents function is the largest function within this module. It serves two purposes, to print out snapshots of the RAM contents at specific points in the program as required by the report assignment, and to print out snapshots for the purposes of testing. It is implemented as a simple for loop that increments through the entire RAM array and prints it with a clean formatting for easy readability.



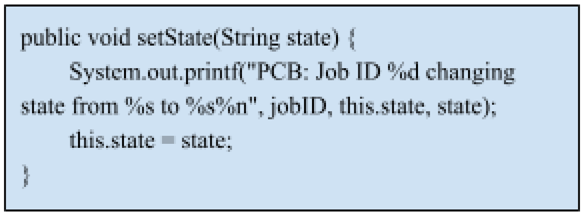
```
 public void printMemoryContents(PrintWriter
outputWriter) {
      outputWriter.println("Memory Contents:");
      for (int address = 0; address < RAM.length;
address++) {
         int data = RAM[address];
         if (data != 0) { // Optionally, only print
non-zero addresses
            outputWriter.printf("Address 0x%04X:
0x%08X%n", address, data);
         }
      }
      outputWriter.println("End of Memory Contents");
}
```

Diagram 9: Shows the printMemoryContents function of the Memory Module.

There are only a handful of functions that remain, most of which deal with resetting the memory or certain pointers. None of which should need to be explained to understand the functionality of the memory module.

PCB Module Implementation

The Process Control Block was designed to hold all of the information about a particular job other than the job itself. It contains numerous pointer variables and static variables for all five states of a program. The five states being: New, Ready, Running, Waiting and Terminated. When created the Process Control Block takes in the job object and sets all the pointers then assigns a programCounter and priority. All the functions of the Process Control Block are getters and setters for the various pointers and Job information stored that are needed by the other modules. The following example below shows one such setter with the testing print statement included.



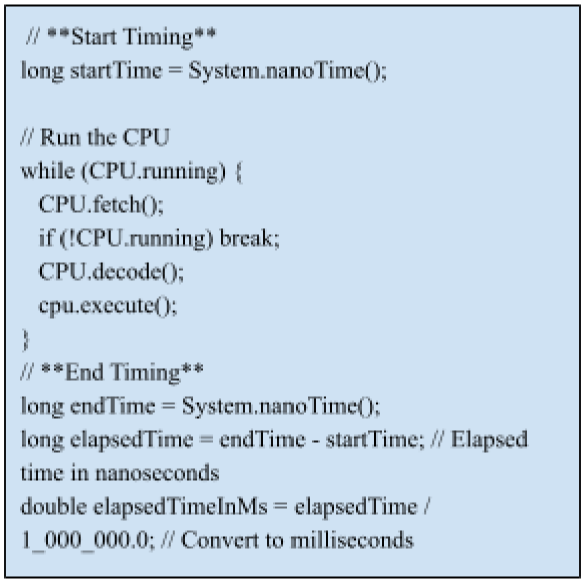
```
public void setState(String state) {
      System.out.printf("PCB: Job ID %d changing
state from %s to %s%n", jobID, this.state, state);
      this.state = state;
}
```

Diagram 10: Shows a section of the PCB module with the setState function.

Simulator Module Implementation

The Simulator file is effectively the kernel and loader combined. It starts by constructing the Memory file and the CPU file, then constructs the Dispatcher, Long Term Scheduler and Process Control Block. After this it loads up the Jobs file and starts creating Job objects for each of the programs contained within the Jobs file.

Once the jobs are stored to the disk, the simulator then creates a Process Control Block and creates a jobs list that functions as a queue. It then resets the CPU and the RAM and tells the Long Term Scheduler to load the job from disk into RAM. After this it tells the Dispatcher to load the process into the CPU and execute. Once in the CPU the job is processed and certain tracking variables are returned to the simulator for testing and analytics purposes. The following is what the simulator is communicating to the CPU, which is in a way acting like the Main() of this Operating System by calling the overarching functions of fetch, decode, and execute.



```
// **Start Timing**
long startTime = System.nanoTime();

// Run the CPU
while (CPU.running) {
  CPU.fetch();
  if (!CPU.running) break;
  CPU.decode();
  cpu.execute();
}
// **End Timing**
long endTime = System.nanoTime();
long elapsedTime = endTime - startTime; // Elapsed
time in nanoseconds
double elapsedTimeInMs = elapsedTime /
1_000_000.0; // Convert to milliseconds
```

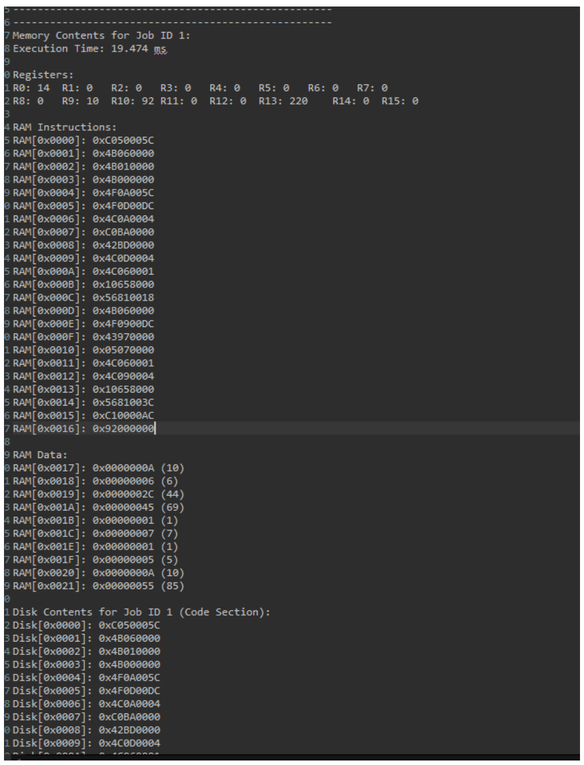
Diagram 11: Shows a section of the Simulator Module, specifically the timing on either side of the main CPU loop.

The simulator will then repeat this cycle with the next job within the jobs list until the queue is empty and the operating system terminates.

Simulation Method

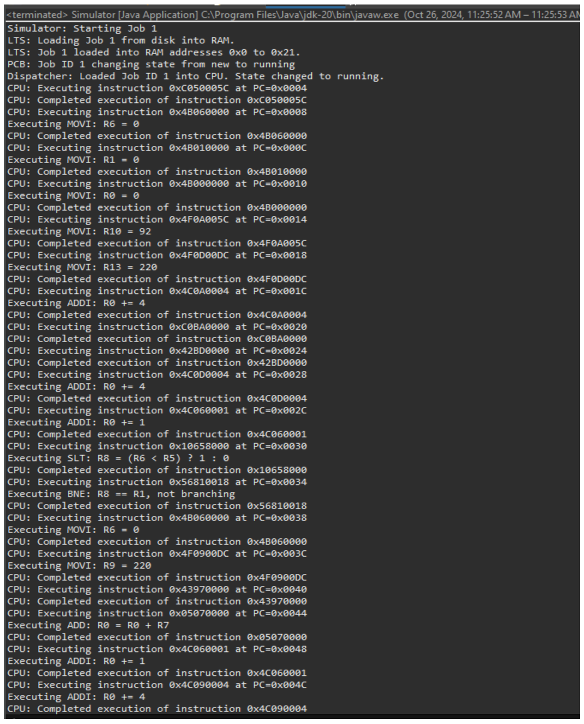
A project of this scale must be simulated and tested numerous times to ensure the completeness and satisfaction of all the requirements. It is also important to simulate this operating system because simulating is the best way for the team to conclude that the system is functional.
The program that was used to build and simulate the operating system was Eclipse. Eclipse was used because it is a reliable and stable integrated development environment with robust organizational, testing and debugging feature sets. However, during development, code was often tested directly in the terminal in addition to using the IDE tools. Testing code in the terminal allowed the team to quickly test the source code, while noting any errors to the functionalities of the operating system that needed to be addressed.
In order to feed the jobs cleanly into the operating system and allow for easy modification to the system code, it was decided to use the json format. This was easier to implement and work with than plain text files as the maven library is robust and easy to implement compared to other plain text libraries. Specifically, one JSON file of 20.9KB was used.
JSON file format is taken from JavaScript object structure. JSON files are text only and they can be used in parallel with many other programming languages, not just JavaScript. According to Jaiswal, JSON files are more versatile and flexible than CSV files and text files: they can deal with large and unstructured data sets with speed and ease [7].
JSON file format was chosen over a text file for this operating system project because along with all of the reasons stated above, the team found it easier to parse the data in JSON format. With this ease in parsing data, the team avoided having to write extra code as they were able to access job data easily without having to manipulate strings. These are just a few of the advantages utilizing JSON file format provided during the completion of this operating system project.
The team was able to time the program directly in Eclipse during the simulation runs, which made it easy for the team in data gathering purposes. The team also timed the simulation runs in Eclipse to draw conclusions on how efficient the CPU was, along with not having to manually time the simulation runs. The exact number of times

the team tested and ran this simulation is unknown, but it is likely over one hundred times. The team needed to run hundreds of tests of the operating system to test the source code. This operating system project was extremely complex at some times, and even the smallest change to the source code could have a massive effect on the performance and the outcome of the results. This is why it was of the utmost importance to simulate and test the operating system many times during its development. When a change to the source code is needed, the team would change the code and test again. Below, the team has included screenshots of simulation runs of specific jobs and their RAM contents. The screenshots display some of the jobs and their executions, along with their RAM and disk contents. By having this information, conclusions can be drawn on how the operating system handled resource allocation and the prioritization of jobs throughout the simulation. Here are some of the screenshots displaying runs of the simulation:



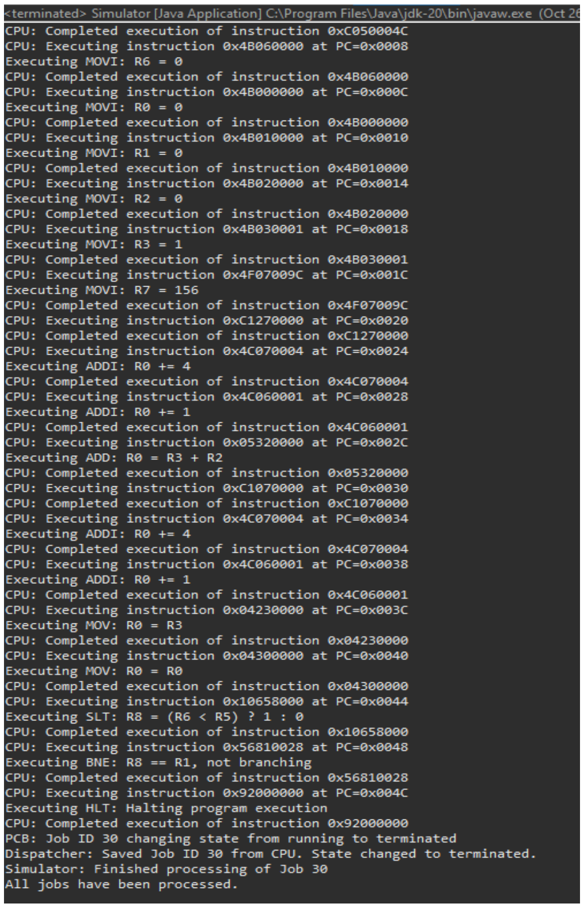
```
--------------------------------------------------
--------------------------------------------------
Memory Contents for Job ID 1:
Execution Time: 19.474 ms

Registers:
R0: 14  R1: 0   R2: 0   R3: 0   R4: 0   R5: 0   R6: 0   R7: 0
R8: 0   R9: 10  R10: 92 R11: 0  R12: 0  R13: 220    R14: 0  R15: 0

RAM Instructions:
RAM[0x0000]: 0xC050005C
RAM[0x0001]: 0x4B060000
RAM[0x0002]: 0x4B010000
RAM[0x0003]: 0x4B000000
RAM[0x0004]: 0x4F0A005C
RAM[0x0005]: 0x4F0D00DC
RAM[0x0006]: 0x4C0A0004
RAM[0x0007]: 0xC0BA0000
RAM[0x0008]: 0x42BD0000
RAM[0x0009]: 0x4C0D0004
RAM[0x000A]: 0x4C060001
RAM[0x000B]: 0x10658000
RAM[0x000C]: 0x56810018
RAM[0x000D]: 0x4B060000
RAM[0x000E]: 0x4F0900DC
RAM[0x000F]: 0x43970000
RAM[0x0010]: 0x05070000
RAM[0x0011]: 0x4C060001
RAM[0x0012]: 0x4C090004
RAM[0x0013]: 0x10658000
RAM[0x0014]: 0x5681003C
RAM[0x0015]: 0xC10000AC
RAM[0x0016]: 0x92000000

RAM Data:
RAM[0x0017]: 0x0000000A (10)
RAM[0x0018]: 0x00000006 (6)
RAM[0x0019]: 0x0000002C (44)
RAM[0x001A]: 0x00000045 (69)
RAM[0x001B]: 0x00000001 (1)
RAM[0x001C]: 0x00000007 (7)
RAM[0x001E]: 0x00000001 (1)
RAM[0x001F]: 0x00000005 (5)
RAM[0x0020]: 0x0000000A (10)
RAM[0x0021]: 0x00000055 (85)

Disk Contents for Job ID 1 (Code Section):
Disk[0x0000]: 0xC050005C
Disk[0x0001]: 0x4B060000
Disk[0x0002]: 0x4B010000
Disk[0x0003]: 0x4B000000
Disk[0x0004]: 0x4F0A005C
Disk[0x0005]: 0x4F0D00DC
Disk[0x0006]: 0x4C0A0004
Disk[0x0007]: 0xC0BA0000
Disk[0x0008]: 0x42BD0000
Disk[0x0009]: 0x4C0D0004
```

Diagram 12: Shows execution of Job 1. It shows the time to complete, register content, RAM content, and Disk contents after completion



```
<terminated> Simulator [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe  (Oct 26, 2024, 11:25:52 AM – 11:25:53 AM
Simulator: Starting Job 1
LTS: Loading Job 1 from disk into RAM.
LTS: Job 1 loaded into RAM addresses 0x0 to 0x21.
PCB: Job ID 1 changing state from new to running
Dispatcher: Loaded Job ID 1 into CPU. State changed to running.
CPU: Executing instruction 0xC050005C at PC=0x0004
CPU: Completed execution of instruction 0xC050005C
CPU: Executing instruction 0x4B060000 at PC=0x0008
Executing MOVI: R6 = 0
CPU: Completed execution of instruction 0x4B060000
CPU: Executing instruction 0x4B010000 at PC=0x000C
Executing MOVI: R1 = 0
CPU: Completed execution of instruction 0x4B010000
CPU: Executing instruction 0x4B000000 at PC=0x0010
Executing MOVI: R0 = 0
CPU: Completed execution of instruction 0x4B000000
CPU: Executing instruction 0x4F0A005C at PC=0x0014
Executing MOVI: R10 = 92
CPU: Completed execution of instruction 0x4F0A005C
CPU: Executing instruction 0x4F0D00DC at PC=0x0018
Executing MOVI: R13 = 220
CPU: Completed execution of instruction 0x4F0D00DC
CPU: Executing instruction 0x4C0A0004 at PC=0x001C
Executing ADDI: R0 += 4
CPU: Completed execution of instruction 0x4C0A0004
CPU: Executing instruction 0xC0BA0000 at PC=0x0020
CPU: Completed execution of instruction 0xC0BA0000
CPU: Executing instruction 0x42BD0000 at PC=0x0024
CPU: Completed execution of instruction 0x42BD0000
CPU: Executing instruction 0x4C0D0004 at PC=0x0028
Executing ADDI: R0 += 4
CPU: Completed execution of instruction 0x4C0D0004
CPU: Executing instruction 0x4C060001 at PC=0x002C
Executing ADDI: R0 += 1
CPU: Completed execution of instruction 0x4C060001
CPU: Executing instruction 0x10658000 at PC=0x0030
Executing SLT: R8 = (R6 < R5) ? 1 : 0
CPU: Completed execution of instruction 0x10658000
CPU: Executing instruction 0x56810018 at PC=0x0034
Executing BNE: R8 == R1, not branching
CPU: Completed execution of instruction 0x56810018
CPU: Executing instruction 0x4B060000 at PC=0x0038
Executing MOVI: R6 = 0
CPU: Completed execution of instruction 0x4B060000
CPU: Executing instruction 0x4F0900DC at PC=0x003C
Executing MOVI: R9 = 220
CPU: Completed execution of instruction 0x4F0900DC
CPU: Executing instruction 0x43970000 at PC=0x0040
CPU: Completed execution of instruction 0x43970000
CPU: Executing instruction 0x05070000 at PC=0x0044
Executing ADD: R0 = R0 + R7
CPU: Completed execution of instruction 0x05070000
CPU: Executing instruction 0x4C060001 at PC=0x0048
Executing ADDI: R0 += 1
CPU: Completed execution of instruction 0x4C060001
CPU: Executing instruction 0x4C090004 at PC=0x004C
Executing ADDI: R0 += 4
CPU: Completed execution of instruction 0x4C090004
```

Diagram 13: Shows the simulator starting Job 1, describes the current states of the LTS, PCB, Dispatcher, and CPU as it executes jobs.



```
<terminated> Simulator [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (Oct 26
CPU: Completed execution of instruction 0xC050004C
CPU: Executing instruction 0x4B060000 at PC=0x0008
Executing MOVI: R6 = 0
CPU: Completed execution of instruction 0x4B060000
CPU: Executing instruction 0x4B000000 at PC=0x000C
Executing MOVI: R0 = 0
CPU: Completed execution of instruction 0x4B000000
CPU: Executing instruction 0x4B010000 at PC=0x0010
Executing MOVI: R1 = 0
CPU: Completed execution of instruction 0x4B010000
CPU: Executing instruction 0x4B020000 at PC=0x0014
Executing MOVI: R2 = 0
CPU: Completed execution of instruction 0x4B020000
CPU: Executing instruction 0x4B030001 at PC=0x0018
Executing MOVI: R3 = 1
CPU: Completed execution of instruction 0x4B030001
CPU: Executing instruction 0x4F07009C at PC=0x001C
Executing MOVI: R7 = 156
CPU: Completed execution of instruction 0x4F07009C
CPU: Executing instruction 0xC1270000 at PC=0x0020
CPU: Completed execution of instruction 0xC1270000
CPU: Executing instruction 0x4C070004 at PC=0x0024
Executing ADDI: R0 += 4
CPU: Completed execution of instruction 0x4C070004
CPU: Executing instruction 0x4C060001 at PC=0x0028
Executing ADDI: R0 += 1
CPU: Completed execution of instruction 0x4C060001
CPU: Executing instruction 0x05320000 at PC=0x002C
Executing ADD: R0 = R3 + R2
CPU: Completed execution of instruction 0x05320000
CPU: Executing instruction 0xC1070000 at PC=0x0030
CPU: Completed execution of instruction 0xC1070000
CPU: Executing instruction 0x4C070004 at PC=0x0034
Executing ADDI: R0 += 4
CPU: Completed execution of instruction 0x4C070004
CPU: Executing instruction 0x4C060001 at PC=0x0038
Executing ADDI: R0 += 1
CPU: Completed execution of instruction 0x4C060001
CPU: Executing instruction 0x04230000 at PC=0x003C
Executing MOV: R0 = R3
CPU: Completed execution of instruction 0x04230000
CPU: Executing instruction 0x04300000 at PC=0x0040
Executing MOV: R0 = R0
CPU: Completed execution of instruction 0x04300000
CPU: Executing instruction 0x10658000 at PC=0x0044
Executing SLT: R8 = (R6 < R5) ? 1 : 0
CPU: Completed execution of instruction 0x10658000
CPU: Executing instruction 0x56810028 at PC=0x0048
Executing BNE: R8 == R1, not branching
CPU: Completed execution of instruction 0x56810028
CPU: Executing instruction 0x92000000 at PC=0x004C
Executing HLT: Halting program execution
CPU: Completed execution of instruction 0x92000000
PCB: Job ID 30 changing state from running to terminated
Dispatcher: Saved Job ID 30 from CPU. State changed to terminated.
Simulator: Finished processing of Job 30
All jobs have been processed.
```

Diagram 14: Shows contents of the CPU and its states during the execution of Job 30. This is the end of the simulation.

Data Analysis

After the simulator is run and loads the 30 jobs onto the CPU, the results that come back are shown below. The execution time of each job along with the waiting time, with the 31st bar being the average for both metrics. All the processes make 3 read/write operations. The waiting time for each process grows as expected, as the CPU runs the processes in sequential order. The waiting time could potentially be decreased substantially with the use of priority scheduling, as some of the longest execution times are loaded into memory first.

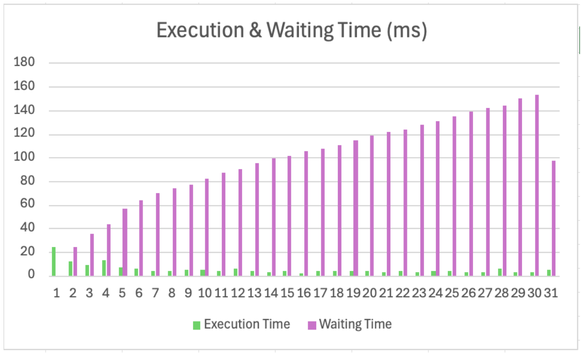


Diagram 15: Shows a visual representation of the execution time and the waiting time for each job labeled by their respective number on the x axis. The y-axis is the time in milliseconds. "Job 31" in this case is the average of all the others.

The percentage of RAM used by each job along with the average amount of RAM used can be seen in the bar graph below. No caching techniques were used.

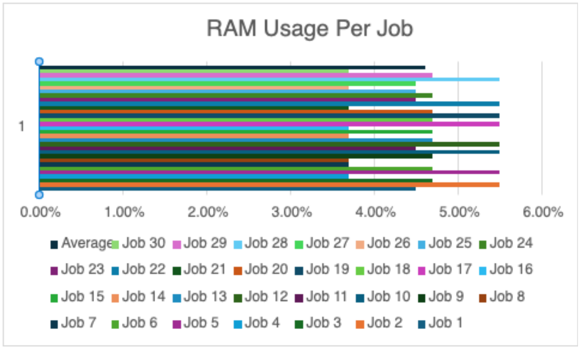


Diagram 16: Shows a visual representation of the percentage of RAM each job, (labeled by color), used during their up time on the CPU. Job 1 starts from the bottom and at the very top we have the average.

Priority scheduling would not have an effect on the RAM Usage per job because that is specific to each job, and each job starts with RAM being free.

Conclusion

After analyzing the simulations, the team concluded that the operating system is running as expected in a sequential order. The waiting time increases for each job as expected with an average of 97.3232 ms. An interesting detail is that the execution time for the first job is significantly higher than the rest of the jobs at 23.854 ms, which is around double the time of the rest of the jobs. The execution time continues to decrease until it levels out around the 10th job and then stays consistent throughout. The explanation for this was assumed to be that the system is taking time to compile in the first job and once completed, doesn't need to compile again for the rest of the jobs. The team also recorded the percentage of RAM that was utilized on each job, with an average of 4.6133%. The RAM usage stayed consistent throughout all of the jobs, with all of the jobs falling between 3.5% and 5.5% usage.

Demonstration

https://youtu.be/aesEmOlsIQI

References

[1] A. Silberschatz, P. Galvin, and G. Gagne, Operating system concepts, 10th ed. Hoboken (Nj): Wiley, Cop, 2018.

[2] FasterXML, "Jackson XML Dataformat," GitHub Repository, 2023. [Online]. Available: https://github.com/FasterXML/jackson-dataformat-xml. [Accessed: 21-October-2024].

[3] Y. D. Liang, *Introduction to Java Programming, Brief Version*, 10th ed. Boston, MA, USA: Pearson, 2015.

[4] **Gaborsch**, "Answer to: 'Max value of integer'," *StackOverflow*, Feb. 21, 2013. [Online]. Available: https://stackoverflow.com/a/15005038. [Accessed: 9-O ctober-2024].

[5] Arpaci-Dusseau, Remzi, and Arpaci-Dusseau, Andrea, Operating Systems: Three Easy Pieces, CreateSpace Independent Publishing Platform, 2018. [Accessed: 10-September-2024 thru 24-October-2024].

[6] sPyOpenSource, "OS", GitHub Repository, 2023. [Online]. Available: https://github.com/sPyOpenSource/os. [Accessed: 18-October-2024].

[7] Jaiswal, Abhishek. "JSON: Introduction, Benefits, Applications, and Drawbacks". *Turing*. https://www.turing.com/kb/what-is-json. [Accessed 26 Oct. 2024.]

Appendix

Appendix A - CPU Source Code

```
package osSim;

import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

public class CPU {
  public static Memory memory;
  public static PCB currentPCB;

  public static int[] registers = new int[16]; // 16 registers (R0 to R15)
  public static int programCounter = 0;
  public int[] instructionMemory;

  public static int currentInstruction;
  public static int opcode;
  public static int reg1;
  public static int reg2;
  public static int reg3;
  public static int address;
  public static int immediate;
  public static int instrType;
  public static boolean running = true;

  public static PrintWriter outputWriter;

  public static Map<Integer, String> opcodeMap = new HashMap<>();

  static {
    opcodeMap.put(0x0, "RD");
    opcodeMap.put(0x1, "WR");
    opcodeMap.put(0x2, "ST");
    opcodeMap.put(0x3, "LW");
    opcodeMap.put(0x4, "MOV");
    opcodeMap.put(0x5, "ADD");
    opcodeMap.put(0x6, "SUB");
```

```
        opcodeMap.put(0x7, "MUL");
        opcodeMap.put(0x8, "DIV");
        opcodeMap.put(0x9, "AND");
        opcodeMap.put(0xA, "OR");
        opcodeMap.put(0xB, "MOVI");
        opcodeMap.put(0xC, "ADDI");
        opcodeMap.put(0xD, "MULI");
        opcodeMap.put(0xE, "DIVI");
        opcodeMap.put(0xF, "LDI");
        opcodeMap.put(0x10, "SLT");
        opcodeMap.put(0x11, "SLTI");
        opcodeMap.put(0x12, "HLT");
        opcodeMap.put(0x13, "NOP");
        opcodeMap.put(0x14, "JMP");
        opcodeMap.put(0x15, "BEQ");
        opcodeMap.put(0x16, "BNE");
        opcodeMap.put(0x17, "BEZ");
        opcodeMap.put(0x18, "BNZ");
        opcodeMap.put(0x19, "BGZ");
        opcodeMap.put(0x1A, "BLZ");
    }

    public CPU() {
        CPU.registers = new int[16];
        CPU.programCounter = 0;
        CPU.running = true;
    }

    public static void fetch() {
        int physicalPC = CPU.programCounter + CPU.currentPCB.getCodeStartAddress();

        if (physicalPC <= CPU.currentPCB.getCodeEndAddress()) {
            currentInstruction = memory.readFromRAM(physicalPC);
            outputWriter.printf("Fetching instruction at PC=0x%04X: 0x%08X%n", physicalPC * 4,
currentInstruction);
            programCounter++;
        } else {
            running = false;
        }
    }
```

```
  public static void decode() {
     instrType = (currentInstruction >>> 30) & 0b11; // Bits 31-30
     opcode = (currentInstruction >>> 24) & 0x3F;    // Bits 29-24 (6 bits)

     switch (instrType) {
        case 0b00: // Arithmetic Instruction Format
           reg1 = (currentInstruction >>> 20) & 0xF; // Bits 23-20
           reg2 = (currentInstruction >>> 16) & 0xF; // Bits 19-16
           reg3 = (currentInstruction >>> 12) & 0xF; // Bits 15-12
           outputWriter.printf("Decoded Arithmetic Instruction: Opcode=%s (0x%X),
Reg1=R%d, Reg2=R%d, Reg3=R%d%n",
                 opcodeMap.get(opcode), opcode, reg1, reg2, reg3);
           break;
        case 0b01: // Conditional Branch and Immediate Format
           reg1 = (currentInstruction >>> 20) & 0xF; // B-reg or Reg1
           reg2 = (currentInstruction >>> 16) & 0xF; // D-reg or Reg2
           address = currentInstruction & 0xFFFF;    // Bits 15-0
           outputWriter.printf("Decoded Conditional Branch/Immediate Instruction: Opcode=%s
(0x%X), Reg1=R%d, Reg2=R%d, Address/Data=0x%X%n",
                 opcodeMap.get(opcode), opcode, reg1, reg2, address);
           break;
        case 0b10: // Unconditional Jump Format
           address = currentInstruction & 0xFFFFFF; // Bits 23-0
           outputWriter.printf("Decoded Unconditional Jump Instruction: Opcode=%s (0x%X),
Address=0x%X%n",
                 opcodeMap.get(opcode), opcode, address);
           break;
        case 0b11: // Input and Output Instruction Format
           reg1 = (currentInstruction >>> 20) & 0xF; // Reg1
           reg2 = (currentInstruction >>> 16) & 0xF; // Reg2
           address = currentInstruction & 0xFFFF;    // Bits 15-0
           outputWriter.printf("Decoded I/O Instruction: Opcode=%s (0x%X), Reg1=R%d,
Reg2=R%d, Address=0x%X%n",
                 opcodeMap.get(opcode), opcode, reg1, reg2, address);
           break;
        default:
           outputWriter.println("Unknown instruction type");
```

```
                break;
            }
        }

        public void execute() {
            // Debug Statement before execution
            System.out.printf("CPU: Executing instruction 0x%08X at PC=0x%04X%n",
    currentInstruction, programCounter * 4);

            switch (opcode) {
                case 0x0: // RD
                    executeRD();
                    break;
                case 0x1: // WR
                    executeWR();
                    break;
                case 0x2: // ST
                    executeST();
                    break;
                case 0x3: // LW
                    executeLW();
                    break;
                case 0x4: // MOV
                    executeMOV();
                    break;
                case 0x5: // ADD
                    executeADD();
                    break;
                case 0x6: // SUB
                    executeSUB();
                    break;
                case 0x7: // MUL
                    executeMUL();
                    break;
                case 0x8: // DIV
                    executeDIV();
                    break;
                case 0x9: // AND
                    executeAND();
                    break;
```

```
case 0xA: // OR
    executeOR();
    break;
case 0xB: // MOVI
    executeMOVI();
    break;
case 0xC: // ADDI
    executeADDI();
    break;
case 0xD: // MULI
    executeMULI();
    break;
case 0xE: // DIVI
    executeDIVI();
    break;
case 0xF: // LDI
    executeLDI();
    break;
case 0x10: // SLT
    executeSLT();
    break;
case 0x11: // SLTI
    executeSLTI();
    break;
case 0x12: // HLT
    executeHLT();
    break;
case 0x13: // NOP
    executeNOP();
    break;
case 0x14: // JMP
    executeJMP();
    break;
case 0x15: // BEQ
    executeBEQ();
    break;
case 0x16: // BNE
    executeBNE();
    break;
case 0x17: // BEZ
```

```
                executeBEZ();
                break;
            case 0x18: // BNZ
                executeBNZ();
                break;
            case 0x19: // BGZ
                executeBGZ();
                break;
            case 0x1A: // BLZ
                executeBLZ();
                break;
            default:
                outputWriter.printf("Unknown opcode: 0x%X%n", opcode);
                break;
        }

        // Debug Statement after execution
        System.out.printf("CPU: Completed execution of instruction 0x%08X%n",
currentInstruction);
        printState();
    }

    // Implementations for each instruction

// RD: Reads data at address or data pointed to by Reg2 into Reg1
    public static void executeRD() {
        if (instrType == 0b11) { // Input and Output Instruction Format
            int data;
            int effectiveAddress;
            if (reg2 == 0) {
                // Direct addressing
                effectiveAddress = address + CPU.currentPCB.getDataStartAddress();
                data = memory.readFromRAM(effectiveAddress);
                outputWriter.printf("Executing RD (Direct): R%d = MEM[0x%X]%n", reg1,
effectiveAddress * 4);
            } else {
                // Indirect addressing
                effectiveAddress = registers[reg2] + address + CPU.currentPCB.getDataStartAddress();
                data = memory.readFromRAM(effectiveAddress);
            outputWriter.printf("Executing RD (Indirect): R%d = MEM[R%d + 0x%X + 
DataStart(0x%X)] = MEM[0x%X]%n",
                reg1, reg2, address, CPU.currentPCB.getDataStartAddress() * 4, effectiveAddress * 
4);
        }
        registers[reg1] = data;
      } else {
        outputWriter.println("Invalid instruction format for RD");
      }
    }

// WR: Write data in Reg1 to address or location pointed to by Reg2
  public static void executeWR() {
      if (instrType == 0b11) { // Input and Output Instruction Format
        int effectiveAddress;
        if (reg2 == 0) {
            // Direct addressing
            effectiveAddress = address + CPU.currentPCB.getDataStartAddress();
            memory.writeToRAM(effectiveAddress, registers[reg1]);
            outputWriter.printf("Executing WR (Direct): MEM[0x%X] = R%d%n", 
effectiveAddress * 4, reg1);
        } else {
            // Indirect addressing
            effectiveAddress = registers[reg2] + address + CPU.currentPCB.getDataStartAddress();
            memory.writeToRAM(effectiveAddress, registers[reg1]);
            outputWriter.printf("Executing WR (Indirect): MEM[R%d + 0x%X + 
DataStart(0x%X)] = MEM[0x%X] = R%d%n",
                reg2, address, CPU.currentPCB.getDataStartAddress() * 4, effectiveAddress * 4, 
reg1);
        }
      } else {
        outputWriter.println("Invalid instruction format for WR");
      }
    }

// ST: Store contents of Reg1 into address pointed to by Reg2 or specified address
  public static void executeST() {
      if (instrType == 0b01) { // Conditional Branch and Immediate Format
        int data = registers[reg1];
        int effectiveAddress;
        if (reg2 == 0) {
          // Direct addressing
          effectiveAddress = address + CPU.currentPCB.getDataStartAddress();
          outputWriter.printf("Executing ST (Direct): MEM[0x%X] = R%d%n",
effectiveAddress * 4, reg1);
        } else {
          // Indirect addressing
          effectiveAddress = registers[reg2] + address + CPU.currentPCB.getDataStartAddress();
          outputWriter.printf("Executing ST (Indirect): MEM[R%d + 0x%X + DataStart(0x%X)]
= MEM[0x%X] = R%d%n",
              reg2, address, CPU.currentPCB.getDataStartAddress() * 4, effectiveAddress * 4,
reg1);
        }
        memory.writeToRAM(effectiveAddress, data);
      } else {
        outputWriter.println("Invalid instruction format for ST");
      }
  }

// LW: Load data into Reg1 from address or location pointed to by Reg2
  public static void executeLW() {
      if (instrType == 0b01) { // Conditional Branch and Immediate Format
        int data;
        int effectiveAddress;
        if (reg2 == 0) {
          // Direct addressing
          effectiveAddress = address + CPU.currentPCB.getDataStartAddress();
        } else {
          // Indirect addressing
          effectiveAddress = registers[reg2] + address + CPU.currentPCB.getDataStartAddress();
        }
        data = memory.readFromRAM(effectiveAddress);
        outputWriter.printf("Executing LW: R%d = MEM[0x%X]%n", reg1, effectiveAddress);
        registers[reg1] = data;
      } else {
        outputWriter.println("Invalid instruction format for LW");
      }
  }
```

```
// MOV: Transfers contents of Reg2 into Reg1
public static void executeMOV() {
   if (instrType == 0b00) { // Arithmetic Instruction Format
      registers[reg3] = registers[reg2]; // Note: D-reg is reg3
      System.out.printf("Executing MOV: R%d = R%d%n", reg3, reg2);
   } else {
      System.out.println("Invalid instruction format for MOV");
   }
}

// ADD: Add Reg2 and Reg3, store in Reg1
public static void executeADD() {
   if (instrType == 0b00) { // Arithmetic Instruction Format
      registers[reg3] = registers[reg1] + registers[reg2]; // D-reg is reg3
      System.out.printf("Executing ADD: R%d = R%d + R%d%n", reg3, reg1, reg2);
   } else {
      System.out.println("Invalid instruction format for ADD");
   }
}

// SUB: Subtract Reg3 from Reg2, store in Reg1
public static void executeSUB() {
   if (instrType == 0b00) { // Arithmetic Instruction Format
      registers[reg3] = registers[reg2] - registers[reg1]; // D-reg is reg3
      System.out.printf("Executing SUB: R%d = R%d - R%d%n", reg3, reg2, reg1);
   } else {
      System.out.println("Invalid instruction format for SUB");
   }
}

// MUL: Multiply Reg2 and Reg3, store in Reg1
public static void executeMUL() {
   if (instrType == 0b00) { // Arithmetic Instruction Format
      registers[reg3] = registers[reg1] * registers[reg2]; // D-reg is reg3
      System.out.printf("Executing MUL: R%d = R%d * R%d%n", reg3, reg1, reg2);
   } else {
      System.out.println("Invalid instruction format for MUL");
   }
}
```

```
// DIV: Divide Reg2 by Reg3, store in Reg1
public static void executeDIV() {
   if (instrType == 0b00) { // Arithmetic Instruction Format
      if (registers[reg2] != 0) {
         registers[reg3] = registers[reg1] / registers[reg2]; // D-reg is reg3
         System.out.printf("Executing DIV: R%d = R%d / R%d%n", reg3, reg1, reg2);
      } else {
         System.out.println("Error: Division by zero");
      }
   } else {
      System.out.println("Invalid instruction format for DIV");
   }
}

// AND: Logical AND of Registers 1 and 2, store in Register 3
public static void executeAND() {
   if (instrType == 0b00) { // Arithmetic Instruction Format
      registers[reg3] = registers[reg1] & registers[reg2];
      System.out.printf("Executing AND: R%d = R%d & R%d%n", reg3, reg1, reg2);
   } else {
      System.out.println("Invalid instruction format for AND");
   }
}

// OR: Logical OR of Registers 1 and 2, store in Register 3
public static void executeOR() {
   if (instrType == 0b00) { // Arithmetic Instruction Format
      registers[reg3] = registers[reg1] | registers[reg2];
      System.out.printf("Executing OR: R%d = R%d | R%d%n", reg3, reg1, reg2);
   } else {
      System.out.println("Invalid instruction format for OR");
   }
}

// MOVI: Copy immediate data into Reg1
public static void executeMOVI() {
   if (instrType == 0b01) { // Conditional Branch and Immediate Format
      registers[reg2] = address; // D-reg is reg2
      System.out.printf("Executing MOVI: R%d = %d%n", reg2, address);
   } else {
        System.out.println("Invalid instruction format for MOVI");
    }
}

// LDI: Same as MOVI
public static void executeLDI() {
    executeMOVI();
}

// ADDI: Add immediate value to Reg1, store in Reg1
public static void executeADDI() {
    if (instrType == 0b01) { // Conditional Branch and Immediate Format
        registers[reg1] += address; // B-reg is reg1
        System.out.printf("Executing ADDI: R%d += %d%n", reg1, address);
    } else {
        System.out.println("Invalid instruction format for ADDI");
    }
}

// MULI: Multiply immediate value with Reg1, store in Reg1
public static void executeMULI() {
    if (instrType == 0b01) { // Conditional Branch and Immediate Format
        registers[reg1] *= address; // B-reg is reg1
        System.out.printf("Executing MULI: R%d *= %d%n", reg1, address);
    } else {
        System.out.println("Invalid instruction format for MULI");
    }
}

// DIVI: Divide Reg1 by immediate value, store in Reg1
public static void executeDIVI() {
    if (instrType == 0b01) { // Conditional Branch and Immediate Format
        if (address != 0) {
            registers[reg1] /= address; // B-reg is reg1
            System.out.printf("Executing DIVI: R%d /= %d%n", reg1, address);
        } else {
            System.out.println("Error: Division by zero");
        }
    } else {
        System.out.println("Invalid instruction format for DIVI");
    }
}

// SLT: Set Reg3 to 1 if Reg1 < Reg2; else 0
public static void executeSLT() {
    if (instrType == 0b00) { // Arithmetic Instruction Format
        registers[reg3] = (registers[reg1] < registers[reg2]) ? 1 : 0;
        System.out.printf("Executing SLT: R%d = (R%d < R%d) ? 1 : 0%n", reg3, reg1, reg2);
    } else {
        System.out.println("Invalid instruction format for SLT");
    }
}

// SLTI: Set Reg2 to 1 if Reg1 < immediate value; else 0
public static void executeSLTI() {
    if (instrType == 0b01) { // Conditional Branch and Immediate Format
        registers[reg2] = (registers[reg1] < address) ? 1 : 0;
        System.out.printf("Executing SLTI: R%d = (R%d < %d) ? 1 : 0%n", reg2, reg1, address);
    } else {
        System.out.println("Invalid instruction format for SLTI");
    }
}

// HLT: Logical end of program
public static void executeHLT() {
    System.out.println("Executing HLT: Halting program execution");
    running = false;
}

// NOP: Do nothing and move to next instruction
public static void executeNOP() {
    System.out.println("Executing NOP: No operation");
}

// JMP: Jump program counter to specified address
public static void executeJMP() {
    if (instrType == 0b10) { // Unconditional Jump Format
        programCounter = address / 4;
        System.out.printf("Executing JMP: Jumping to address 0x%X%n", address);
    } else {
            System.out.println("Invalid instruction format for JMP");
        }
    }


    // BEQ: Branch when Reg1 equals Reg2
    public static void executeBEQ() {
        if (instrType == 0b01) { // Conditional Branch and Immediate Format
            if (registers[reg1] == registers[reg2]) {
                programCounter = address / 4;
                System.out.printf("Executing BEQ: R%d == R%d, branching to address 0x%X%n",
reg1, reg2, address);
            } else {
                System.out.printf("Executing BEQ: R%d != R%d, not branching%n", reg1, reg2);
            }
        } else {
            System.out.println("Invalid instruction format for BEQ");
        }
    }


    // BNE: Branch when Reg1 does not equal Reg2
    public static void executeBNE() {
        if (instrType == 0b01) { // Conditional Branch and Immediate Format
            if (registers[reg1] != registers[reg2]) {
                programCounter = address / 4;
                System.out.printf("Executing BNE: R%d != R%d, branching to address 0x%X%n",
reg1, reg2, address);
            } else {
                System.out.printf("Executing BNE: R%d == R%d, not branching%n", reg1, reg2);
            }
        } else {
            System.out.println("Invalid instruction format for BNE");
        }
    }

    // BEZ: Branch when Reg1 equals zero
    public static void executeBEZ() {
        if (instrType == 0b01) { // Conditional Branch and Immediate Format
            if (registers[reg1] == 0) {
                programCounter = address / 4;
                System.out.printf("Executing BEZ: R%d == 0, branching to address 0x%X%n", reg1,
address);
            } else {
                System.out.printf("Executing BEZ: R%d != 0, not branching%n", reg1);
            }
        } else {
            System.out.println("Invalid instruction format for BEZ");
        }
    }

    // BNZ: Branch when Reg1 does not equal zero
    public static void executeBNZ() {
        if (instrType == 0b01) { // Conditional Branch and Immediate Format
            if (registers[reg1] != 0) {
                programCounter = address / 4;
                System.out.printf("Executing BNZ: R%d != 0, branching to address 0x%X%n", reg1,
address);
            } else {
                System.out.printf("Executing BNZ: R%d == 0, not branching%n", reg1);
            }
        } else {
            System.out.println("Invalid instruction format for BNZ");
        }
    }

    // BGZ: Branch when Reg1 is greater than zero
    public static void executeBGZ() {
        if (instrType == 0b01) { // Conditional Branch and Immediate Format
            if (registers[reg1] > 0) {
                programCounter = address / 4;
                System.out.printf("Executing BGZ: R%d > 0, branching to address 0x%X%n", reg1,
address);
            } else {
                System.out.printf("Executing BGZ: R%d <= 0, not branching%n", reg1);
            }
        } else {
            System.out.println("Invalid instruction format for BGZ");
        }
    }
    // BLZ: Branch when Reg1 is less than zero
    public static void executeBLZ() {
        if (instrType == 0b01) { // Conditional Branch and Immediate Format
            if (registers[reg1] < 0) {
                programCounter = address / 4;
                System.out.printf("Executing BLZ: R%d < 0, branching to address 0x%X%n", reg1,
address);
            } else {
                System.out.printf("Executing BLZ: R%d >= 0, not branching%n", reg1);
            }
        } else {
            System.out.println("Invalid instruction format for BLZ");
        }
    }

    // Method to print the state of registers and relevant memory
    public void printState() {
        outputWriter.println("Registers:");
        for (int i = 0; i < registers.length; i++) {
            outputWriter.printf("R%d: %d\t", i, registers[i]);
            if ((i + 1) % 8 == 0) outputWriter.println();
        }

        // Only print the Output Buffer
        outputWriter.println("\nData Memory (Relevant Addresses):");

        int ouptAddress = 0x00AC;
        outputWriter.printf("Output Buffer (Address 0x%X):%n", ouptAddress);
        outputWriter.printf("MEM[0x%X]: %d%n", ouptAddress,
memory.readFromRAM(ouptAddress));
        outputWriter.println("--------------------------------------------------");
    }

    public static void resetCPU() {
        registers = new int[16];
        programCounter = 0;
        running = true;
        currentInstruction = 0;
        opcode = 0;
    reg1 = 0;
    reg2 = 0;
    reg3 = 0;
    address = 0;
    immediate = 0;
    instrType = 0;
  }
}
```

Appendix B - Dispatcher Source Code

```
package osSim;

public class Dispatcher {
  public void loadProcessToCPU(PCB pcb) {
     CPU.registers = pcb.getRegisters().clone();
     CPU.programCounter = pcb.getProgramCounter();
     CPU.currentPCB = pcb; // Set the current PCB in the CPU
     pcb.setState(PCB.RUNNING);
     System.out.printf("Dispatcher: Loaded Job ID %d into CPU. State changed to %s.%n",
pcb.getJobID(), pcb.getState());
  }

  public void saveProcessFromCPU(CPU cpu, PCB pcb) {
     pcb.setRegisters(CPU.registers.clone());
     pcb.setProgramCounter(CPU.programCounter);
     pcb.setState(PCB.TERMINATED);
     System.out.printf("Dispatcher: Saved Job ID %d from CPU. State changed to %s.%n",
pcb.getJobID(), pcb.getState());
  }
}
```

Appendix C - Memory Source Code

```
package osSim;

import java.io.PrintWriter;
import java.util.Arrays;
// memory for our simulator

public class Memory {
  private int[] RAM;
  private int[] disk;
  private int diskPointer = 0; // Points to the next free disk location

  public Memory() {
     RAM = new int[1024];
     disk = new int[2048];
     Arrays.fill(RAM, 00000000);
     Arrays.fill(disk, 00000000);
  }

  public int readFromRAM(int address) {
     if (address < 0 || address >= RAM.length) {
        throw new IllegalArgumentException("Invalid address");
     }
     return RAM[address];
  }

  public void writeToRAM(int address, int data) {
     if (address < 0 || address >= RAM.length) {
        throw new IllegalArgumentException("Invalid address");
     }
     RAM[address] = data;
  }

  public int readFromDisk(int address) {
     if (address < 0 || address >= disk.length) {
        throw new IllegalArgumentException("Invalid address");
     }
     return disk[address];
  }
```

```
public int getRAMSize() {
   return RAM.length;
}

public void writeToDisk(int address, int data) {
   if (address < 0 || address >= disk.length) {
      throw new IllegalArgumentException("Invalid address");
   }
   disk[address] = data;
}

public int getDiskSize() {
   return disk.length;
}

public int effectiveAddress(int base, int displacement, boolean indirect, int indexRegister) {
   if (indirect) {
      return RAM[base] + RAM[indexRegister] + displacement;
   } else {
      return base + displacement;
   }
}

public void resetMemory() {
   Arrays.fill(RAM, 0);
}

public void resetDiskPointer() {
   diskPointer = 0;
}

public int getDiskPointer() {
   return diskPointer;
}

public void incrementDiskPointer(int value) {
   diskPointer += value;
}

public void printMemoryContents(PrintWriter outputWriter) {
        outputWriter.println("Memory Contents:");
        for (int address = 0; address < RAM.length; address++) {
            int data = RAM[address];
            if (data != 0) { // Optionally, only print non-zero addresses
                outputWriter.printf("Address 0x%04X: 0x%08X%n", address, data);
            }
        }
        outputWriter.println("End of Memory Contents");
    }


    public void dumpEntireMemoryContents(PrintWriter outputWriter) {
        outputWriter.println("--------------------------------------------------");
        outputWriter.println("Complete Memory Dump");

        // Dump RAM Contents
        outputWriter.println("\nRAM Contents:");
        for (int address = 0; address < RAM.length; address++) {
            int data = RAM[address];
            outputWriter.printf("RAM[0x%04X]: 0x%08X%n", address, data);
        }

        // Dump Disk Contents
        outputWriter.println("\nDisk Contents:");
        for (int address = 0; address < disk.length; address++) {
            int data = disk[address];
            outputWriter.printf("Disk[0x%04X]: 0x%08X%n", address, data);
        }

        outputWriter.println("End of Memory Dump");
        outputWriter.println("--------------------------------------------------");
    }
}
```

Appendix D - Job Source Code

```
import com.fasterxml.jackson.annotation.JsonProperty;
import java.util.List;

public class Job {
  private int jobID;
  private int priority; // Added priority field
  private List<String> code;
  private List<String> data;

  // Default constructor required for Jackson
  public Job() {}

  public Job(@JsonProperty("job_id") int jobID,
         @JsonProperty("priority") String priority, // Add priority to constructor
         @JsonProperty("code") List<String> code,
         @JsonProperty("data") List<String> data) {
    this.jobID = jobID;
    this.priority = Integer.parseInt(priority, 16); // Initialize priority
    this.code = code;
    this.data = data;
  }

  // Getters and setters
  public int getJobID() {
    return jobID;
  }

  public void setJobID(int jobID) {
    this.jobID = jobID;
  }

  public int getPriority() { // Getter for priority
    return priority;
  }

  public void setPriority(int priority) { // Setter for priority
    this.priority = priority;
  }
```

```
    public List<String> getCode() {
        return code;
    }

    public void setCode(List<String> code) {
        this.code = code;
    }

    public List<String> getData() {
        return data;
    }

    public void setData(List<String> data) {
        this.data = data;
    }
}
```

Appendix E - Long Term Scheduler Source Code

```
package osSim;

public class LTS {
  private Memory memory;
  private int memoryPointer;

  public LTS(Memory memory) {
     this.memory = memory;
     this.memoryPointer = 0; // Start of RAM
  }

  public void loadSingleJobFromDisk(PCB pcb) {
     System.out.printf("LTS: Loading Job %d from disk into RAM.%n", pcb.getJobID());

     // Load code into memory
     int codeStartAddress = memoryPointer;
     for (String instruction : pcb.getJob().getCode()) {
        int instructionValue = Integer.parseUnsignedInt(instruction.substring(2), 16);
        memory.writeToRAM(memoryPointer++, instructionValue);
     }
     int codeEndAddress = memoryPointer - 1;

     // Load data into memory
     int dataStartAddress = memoryPointer;
     for (String dataWord : pcb.getJob().getData()) {
        int dataValue = Integer.parseUnsignedInt(dataWord.substring(2), 16);
        memory.writeToRAM(memoryPointer++, dataValue);
     }
     int dataEndAddress = memoryPointer - 1;

     // Update PCB with code and data addresses
     pcb.setCodeStartAddress(codeStartAddress);
     pcb.setCodeEndAddress(codeEndAddress);
     pcb.setDataStartAddress(dataStartAddress);
     pcb.setDataEndAddress(dataEndAddress);

     System.out.printf("LTS: Job %d loaded into RAM addresses 0x%X to 0x%X.%n",
           pcb.getJobID(), codeStartAddress, dataEndAddress);
  }
  public void resetMemoryPointer() {
     memoryPointer = 0;
  }

  public void clearJobQueue() {
    // TODO Auto-generated method stub

  }
}
```

Appendix F - PCB Source Code

```
package osSim;

public class PCB {
  public static final String NEW = "new";
  public static final String READY = "ready";
  public static final String RUNNING = "running";
  public static final String WAITING = "waiting";
  public static final String TERMINATED = "terminated";

  private int jobID;
  private int codeStartAddress;
  private int codeEndAddress;
  private int dataStartAddress;
  private int dataEndAddress;
  private int programCounter;
  private int[] registers;
  private String state;
  private Job job; // Reference to the Job object

  // **New Fields for Disk Addresses**
  private int diskStartAddress;
  private int diskEndAddress;

  public PCB(Job job) {
    this.job = job;
    this.jobID = job.getJobID();
    this.programCounter = 0;
    this.registers = new int[16];
    this.state = NEW;
  }

  // Getters and setters for code/data addresses
  public int getJobID() {
    return jobID;
  }

  public int getCodeStartAddress() {
    return codeStartAddress;
  }
```

```
public void setCodeStartAddress(int codeStartAddress) {
   this.codeStartAddress = codeStartAddress;
}

public int getCodeEndAddress() {
   return codeEndAddress;
}

public void setCodeEndAddress(int codeEndAddress) {
   this.codeEndAddress = codeEndAddress;
}

public int getDataStartAddress() {
   return dataStartAddress;
}

public void setDataStartAddress(int dataStartAddress) {
   this.dataStartAddress = dataStartAddress;
}

public int getDataEndAddress() {
   return dataEndAddress;
}

public void setDataEndAddress(int dataEndAddress) {
   this.dataEndAddress = dataEndAddress;
}

public int getProgramCounter() {
   return programCounter;
}

public void setProgramCounter(int programCounter) {
   this.programCounter = programCounter;
}

public int[] getRegisters() {
   return registers;
}
```

```
    public void setRegisters(int[] registers) {
        this.registers = registers;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        System.out.printf("PCB: Job ID %d changing state from %s to %s%n", jobID, this.state,
state);
        this.state = state;
    }

    public Job getJob() {
        return job;
    }

    // **New Getters and Setters for Disk Addresses**

    public int getDiskStartAddress() {
        return diskStartAddress;
    }

    public void setDiskStartAddress(int diskStartAddress) {
        this.diskStartAddress = diskStartAddress;
    }

    public int getDiskEndAddress() {
        return diskEndAddress;
    }

    public void setDiskEndAddress(int diskEndAddress) {
        this.diskEndAddress = diskEndAddress;
    }
}
```

Appendix G - Simulator Source Code

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Comparator;
import java.util.List;
import java.io.File;
import java.util.ArrayList;
import java.util.PriorityQueue;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.core.type.TypeReference;

public class Simulator {
  public static void main(String[] args) {
      // Initialize Memory and CPU
      Memory memory = new Memory();
      CPU cpu = new CPU();
      CPU.memory = memory;

      // Initialize Dispatcher and LTS
      Dispatcher dispatcher = new Dispatcher();
      LTS lts = new LTS(memory);

      // Load jobs from JSON file
      List<Job> jobs = loadJobsFromJson("jobs.json");

      // Create a priority queue for PCBs, prioritized by job priority
      PriorityQueue<PCB> jobQueue = new PriorityQueue<>(Comparator.comparingInt(pcb ->
pcb.getJob().getPriority()));

      try {
         CPU.outputWriter = new PrintWriter(new FileWriter("output.txt"));
      } catch (IOException e) {
         e.printStackTrace();
         return;
      }

      // Add PCBs to the job queue
      for (Job job : jobs) {
```

```
    jobQueue.offer(new PCB(job));
}

// Process jobs from the priority queue
while (!jobQueue.isEmpty()) {
    PCB pcb = jobQueue.poll(); // Get the highest priority PCB
    int currentJobID = pcb.getJobID();
    System.out.printf("Simulator: Starting Job %d%n", currentJobID);

    // Reset CPU and memory for the job
    CPU.resetCPU();
    memory.resetMemory();
    lts.resetMemoryPointer(); // Reset RAM pointer before loading the job

    // LTS loads the current job from disk into RAM using PCB
    lts.loadSingleJobFromDisk(pcb);

    // Load process into CPU
    dispatcher.loadProcessToCPU(pcb);

    // **Start Timing**
    long startTime = System.nanoTime();

    // Run the CPU
    while (CPU.running) {
        CPU.fetch();
        if (!CPU.running) break;
        CPU.decode();
        cpu.execute();
    }

    // **End Timing**
    long endTime = System.nanoTime();
    long elapsedTime = endTime - startTime; // Elapsed time in nanoseconds
    double elapsedTimeInMs = elapsedTime / 1_000_000.0; // Convert to milliseconds

    // Save process state
    dispatcher.saveProcessFromCPU(cpu, pcb);

    // Print memory contents before resetting memory
```

```
            printMemoryContents(memory, CPU.outputWriter, pcb, elapsedTimeInMs);

            // Clear job queue for the next job
            lts.clearJobQueue();

            System.out.printf("Simulator: Finished processing of Job %d%n", currentJobID);
        }

        // Close the output writer
        cpu.outputWriter.close();

        System.out.println("All jobs have been processed.");
    }

    // Method to read jobs from JSON file
    public static List<Job> loadJobsFromJson(String filename) {
        List<Job> jobs = new ArrayList<>();
        try {
            ObjectMapper objectMapper = new ObjectMapper();
            File file = new File(filename);

            // Read JSON file and convert to List<Job>
            jobs = objectMapper.readValue(file, new TypeReference<List<Job>>() {});

        } catch (Exception e) {
            e.printStackTrace();
        }
        return jobs;
    }

    public static void printMemoryContents(Memory memory, PrintWriter outputWriter, PCB pcb,
double elapsedTimeInMs) {
        outputWriter.println("--------------------------------------------------");
        outputWriter.printf("Memory Contents for Job ID %d:%n", pcb.getJobID());

        // Print Execution Time
        outputWriter.printf("Execution Time: %.3f ms%n", elapsedTimeInMs);

        // Print CPU registers
        outputWriter.println("\nRegisters:");
        for (int i = 0; i < CPU.registers.length; i++) {
            outputWriter.printf("R%d: %d\t", i, CPU.registers[i]);
            if ((i + 1) % 8 == 0) outputWriter.println();
        }

        // Print RAM Instructions
        outputWriter.println("\nRAM Instructions:");
        for (int address = pcb.getCodeStartAddress(); address <= pcb.getCodeEndAddress();
address++) {
            int data = memory.readFromRAM(address);
            outputWriter.printf("RAM[0x%04X]: 0x%08X%n", address, data);
        }

        // Print RAM Data
        outputWriter.println("\nRAM Data:");
        for (int address = pcb.getDataStartAddress(); address <= pcb.getDataEndAddress();
address++) {
            int data = memory.readFromRAM(address);
            if (data != 0) { // Skip zeros if desired
                outputWriter.printf("RAM[0x%04X]: 0x%08X (%d)%n", address, data, data);
            }
        }

        // Print Disk Contents for this job (Code Section)
        outputWriter.println("\nDisk Contents for Job ID " + pcb.getJobID() + " (Code Section):");
        List<String> code = pcb.getJob().getCode();
        for (int i = 0; i < code.size(); i++) {
            outputWriter.printf("Disk[0x%04X]: %s%n", i, code.get(i));
        }

        // Print Disk Contents for this job (Data Section)
        outputWriter.println("\nDisk Contents for Job ID " + pcb.getJobID() + " (Data Section):");
        List<String> dataSection = pcb.getJob().getData();
        for (int i = 0; i < dataSection.size(); i++) {
            String dataString = dataSection.get(i);
            if (!dataString.equals("0x00000000")) {
                outputWriter.printf("Disk[0x%04X]: %s%n", code.size() + i, dataString);
            }
        }
        outputWriter.println("End of Memory Contents");
        outputWriter.println("-----------------------------------------------------");
    }
}
```

Appendix H - Core Dump