OS Simulation

CS 3410-Section 2

Elizabeth Herndon
Payton Mock
Zach Comstock

Abstract

The project that we will be discussing in this paper is the development, implementation, and testing of an Operating System simulation. The purpose of this project is to help students understand the fundamentals of what components make up an Operating System as well as the techniques that can improve the process speed and functionality of the OS. To help us understand what an Operating System is, we will be looking at functions such as memory management, multithreading, synchronization, and multiprocessing. Our final testing should provide us with enough information that we can conclude that certain scheduling processes and the size of the pages that we use has an impact on the overall performance of our system.

Introduction

This project focuses on the loading of a program file that will contain a list of 30 jobs that will then travel through the Operating System. This includes interactions with hard disk, long term scheduler, RAM, short term scheduler, and finally execution on the CPU(s). In this project we will learn how the different modules of an operating system interact with one another and how they share resources. In Phase 2 we will also implement the Shortest Job First scheduling method and draw conclusions. We will gather data on the effectiveness of paging and how the number of pages and frames improves or slows down our system's performance. We have decided as to implement Paging into our Phase 1 and will be comparing the effectiveness of paging as a whole.

Design

The design approach that we took when developing our OS Simulator was to break down the system into several different modules that would interact and share data with one another as the program file was processed by the system. For our design we started with a Data Flow Diagram that helped us see how the program file moves through the system and what takes place in each module. The first step was to read in the file and save it to the Hard Disk. A PCB object would be created for each job and would contain important information such as the Program Counter and Job Length that we would need to have access to throughout the OS. The Long Term Scheduler then checks RAM to make sure there is room to push the next job. The jobs are then loaded into the Ready Queue by the Short Term Scheduler. The jobs will be sorted using either First come first serve, Priority, or Shortest Job First which we added for Phase 2 of the project. From there the Dispatcher will check the CPU(s) and locate one that is idle and push the job to that CPU. The final step in the system is for the CPU to decode the job and execute the instructions. The finished job is then placed on the terminate queue. We have also designed a Stopwatch module that is responsible for gathering the metrics that we want to measure such as Average Wait time, Average Execution Time, CPU usage, and overall total time that the system ran.

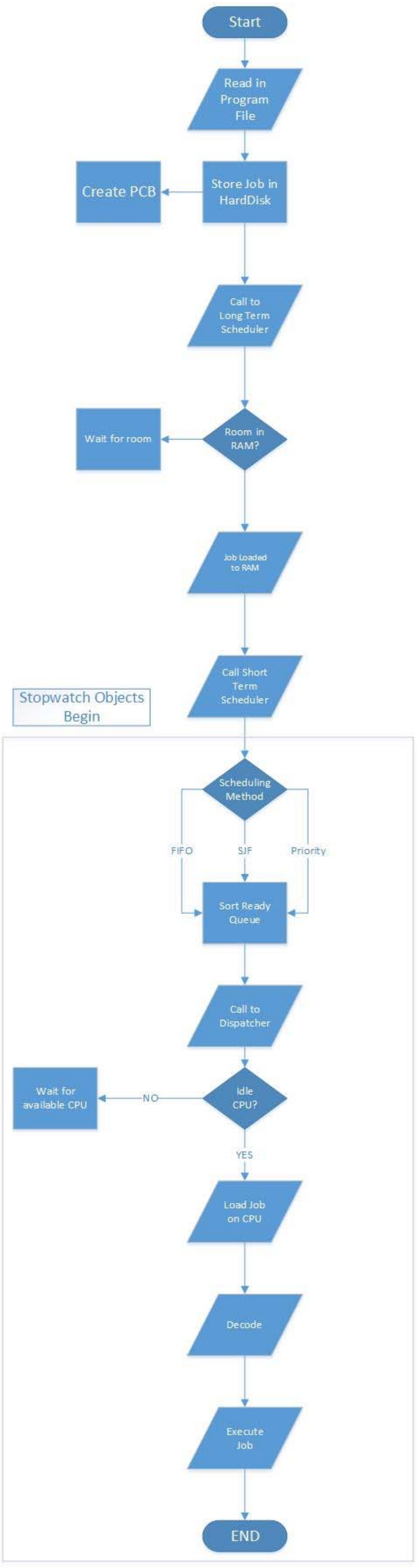


Figure 1 Operating System Flow Diagram

Implementation

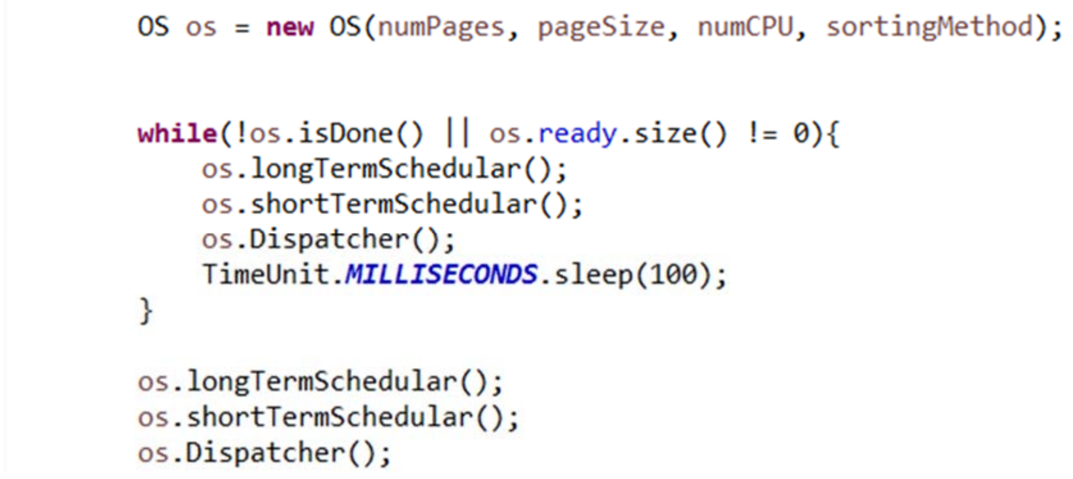
Our simulation of the operating system was done using the Java programming language and all executions for the code were done using the java virtual machine. We decided to use java as it provides a number of built in libraries and methods that could help us in programming the OS. Also, java handles pointers much better than C++ or C and our group felt that we shouldn't focus on the code as much as the implementation, so we took the easiest route.

Our OS used a number of different classes that contained methods involved in the functionality of the named class. The classes worked together and each had individual roles in the operating system. Below is a brief description of each class as well as a few screen shots associated with important code in the given class.

Classes with extensive methods do not have a screen shot to save room for the paper.

Driver

The Driver class is one of the most important classes as it contains the main method for running the program as well as several statements which allow us to gather data and save it for later use. Specifically, the driver is used to start the loop that will run the schedulers and dispatcher until each job is finished.

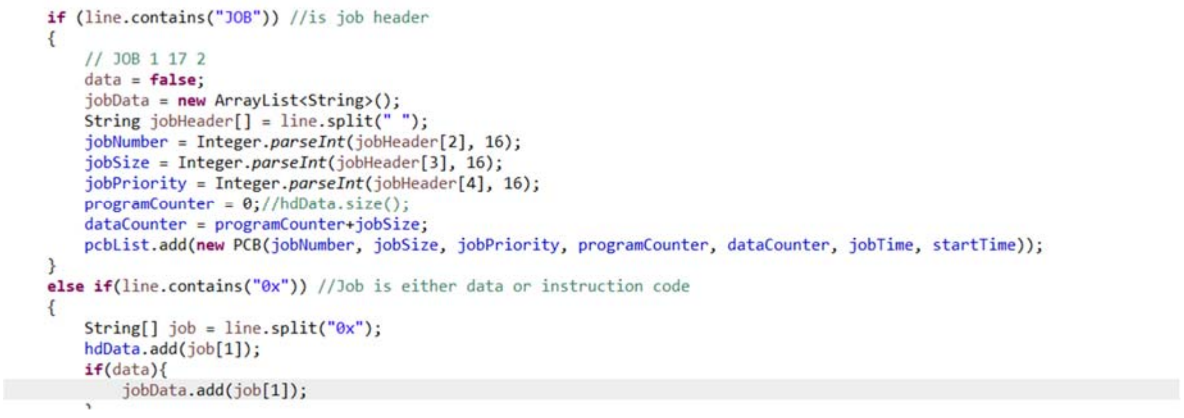


```
OS os = new OS(numPages, pageSize, numCPU, sortingMethod);


while(!os.isDone() || os.ready.size() != 0){
    os.longTermSchedular();
    os.shortTermSchedular();
    os.Dispatcher();
    TimeUnit.MILLISECONDS.sleep(100);
}

os.longTermSchedular();
os.shortTermSchedular();
os.Dispatcher();
```

After the loop is finished, we run the code in the loop once more to make sure that every job is finished. Our driver is separated into two main sections: an automated section and a manual section. The automated section runs around 50 unique tests sequentially, and gather the date for each test for us to observe after the test is over. The manual option allows the user to control the metrics and see individual data points.

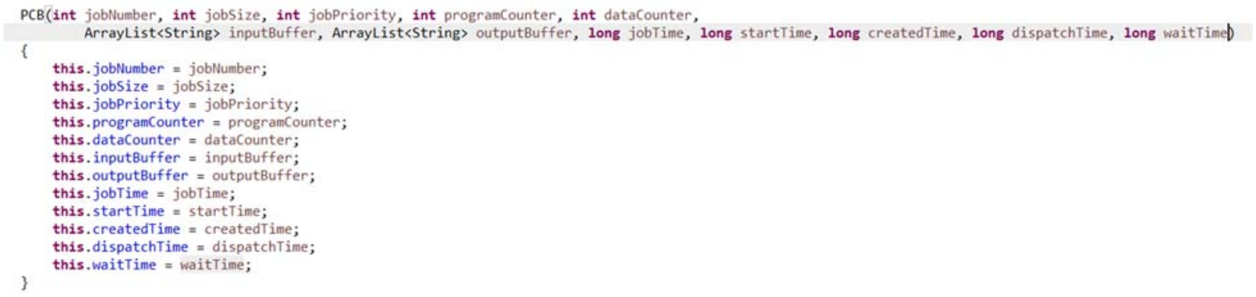
HardDisk

The HardDisk class is used to load each job into the operating system in a method called input(). The input method reads in the Program-file text document. After this it separates the document into individual lines and begins delimitating some characters from each line for later use. For example, the line "// JOB 1 17 2" has several pieces of information that are needed, so we delimitate each portion of the string that we need and save it for later use. The hard disk is also responsible for creating a PCB for each job using the information in the text document.



```
if (line.contains("JOB")) //is job header
{
    // JOB 1 17 2
    data = false;
    jobData = new ArrayList<String>();
    String jobHeader[] = line.split(" ");
    jobNumber = Integer.parseInt(jobHeader[2], 16);
    jobSize = Integer.parseInt(jobHeader[3], 16);
    jobPriority = Integer.parseInt(jobHeader[4], 16);
    programCounter = 0;//hdData.size();
    dataCounter = programCounter+jobSize;
    pcbList.add(new PCB(jobNumber, jobSize, jobPriority, programCounter, dataCounter, jobTime, startTime));
}
else if(line.contains("0x")) //Job is either data or instruction code
{
    String[] job = line.split("0x");
    hdData.add(job[1]);
    if(data){
        jobData.add(job[1]);
```

<u>PCB</u>
The PCB class is used throughout the program to initialize what is inside each PC. It contains several different variables that are unique to each job. Some example of these variables include: job size, program counter, priority, job completion time, etc. This information is passed throughout the program as it is needed. The variables are not static as they often change as they make their way through each class.

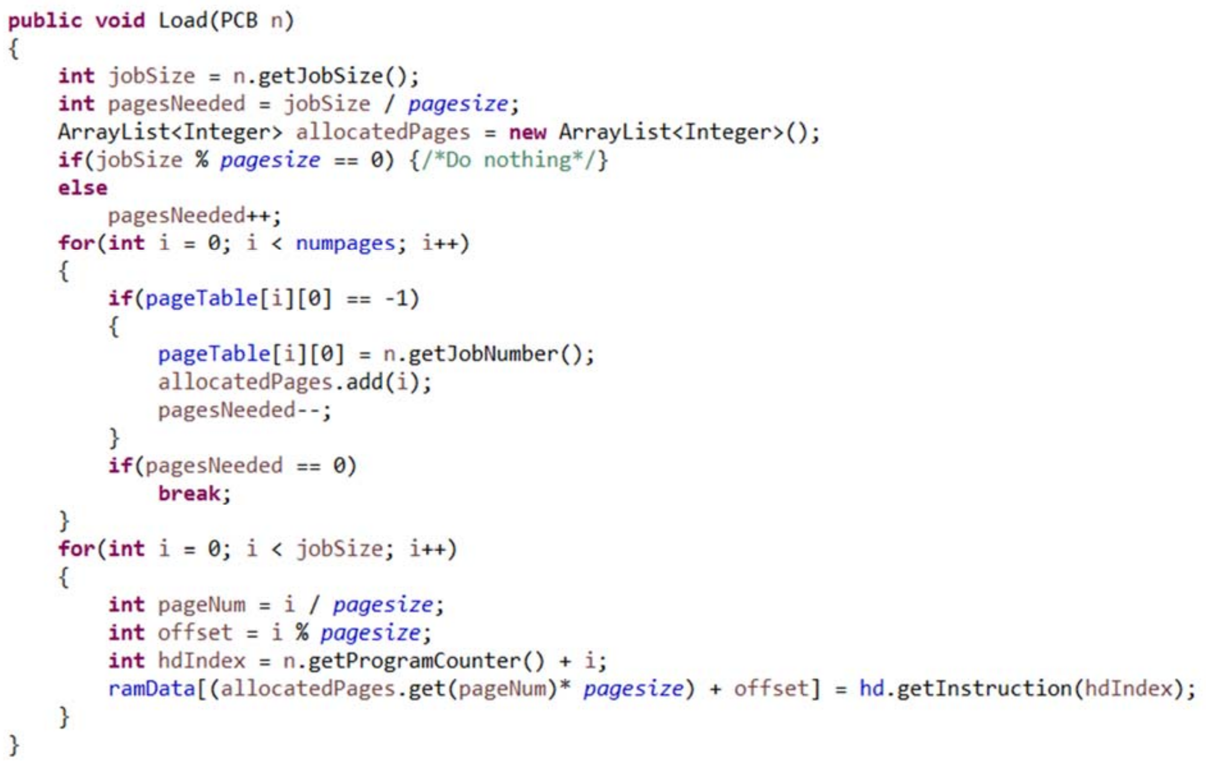


```
PCB(int jobNumber, int jobSize, int jobPriority, int programCounter, int dataCounter,
        ArrayList<String> inputBuffer, ArrayList<String> outputBuffer, long jobTime, long startTime, long createdTime, long dispatchTime, long waitTime)
{
    this.jobNumber = jobNumber;
    this.jobSize = jobSize;
    this.jobPriority = jobPriority;
    this.programCounter = programCounter;
    this.dataCounter = dataCounter;
    this.inputBuffer = inputBuffer;
    this.outputBuffer = outputBuffer;
    this.jobTime = jobTime;
    this.startTime = startTime;
    this.createdTime = createdTime;
    this.dispatchTime = dispatchTime;
    this.waitTime = waitTime;
}
```

<u>OS</u>
The OS class contains methods and objects used throughout the program, but its primary function is the long term and short term schedulers, as well as the dispatcher. The function of the short-term scheduler is to schedule the jobs based on the algorithm chosen, while the long term scheduler puts the jobs into ram in the order they were scheduled. The dispatcher is responsible for switching jobs in and out of the CPU based on the flags associated with each CPU. For example, if a CPU's "idle" flag is set to "true", then the dispatcher sends a job to the CPU to be executed. If the dispatcher sees that the CPU is currently running, then it will look at the next CPU (if available).

<u>RAM</u>
The RAM class is obviously the class that virtualizes the random access memory. It stores jobs from the hard disk and is where the jobs are held until they are ready to be executed. The RAM is loaded differently depending on a few variables: The scheduling algorithm used, the number of pages used, and the page size. The scheduling algorithm is determined by the used and is handled in the short term scheduler. The page size and number of pages is handled directly by the user. The pages are handled by a two-dimensional array, the first dimension being the number of pages and the second being the size of each page. Specifically, the RAM contains

three major methods: getRam(), canFit(), and load(). getRAM() is used to go to a specific memory location, converting the virtual address to a physical address, and return the data at the given address. confit() returns a Boolean variable and is responsible for telling if a given job can fit into RAM. The load() method is responsible for loading the PCBs into ram using the assigned page size, page count, and offset. The long term scheduler calls this method when loading jobs into memory. Along with the primary methods, the RAM class also contains methods for deallocation of memory, which is called after a job is finished executing, and a core dump



```
public void Load(PCB n)
{
    int jobSize = n.getJobSize();
    int pagesNeeded = jobSize / pagesize;
    ArrayList<Integer> allocatedPages = new ArrayList<Integer>();
    if(jobSize % pagesize == 0) {/*Do nothing*/}
    else
        pagesNeeded++;
    for(int i = 0; i < numpages; i++)
    {
        if(pageTable[i][0] == -1)
        {
            pageTable[i][0] = n.getJobNumber();
            allocatedPages.add(i);
            pagesNeeded--;
        }
        if(pagesNeeded == 0)
            break;
    }
    for(int i = 0; i < jobSize; i++)
    {
        int pageNum = i / pagesize;
        int offset = i % pagesize;
        int hdIndex = n.getProgramCounter() + i;
        ramData[(allocatedPages.get(pageNum)* pagesize) + offset] = hd.getInstruction(hdIndex);
    }
}
```

<u>CPU</u>

Finally, the CPU class is responsible for executing the jobs. It contains 3 primary methods, contextSwitchIn(), contextSwitchOut(), and run(), as well as a secondary methods which translates an instruction into binary format. Along with the methods, the class also contains the global arrays “flag”, which contains all associated flags for a given CPU, and “registers”, which stores the values of each register. The CPU begins with the dispatcher calling the contextSwitchIn() method, which passes a PCB to the CPU class. The CPU then executes the instructions associated with that class using the run() method while also setting the “idle” flag to true. Once the job has completely finished executing, the CPUs idle flag is set to “true” as well as the flag for sending the current job to the terminate queue. Once the dispatcher sees the CPU has a job ready to be terminated, it calls the contextSwitchOut() method to gather all of the PCB information for the executed job and sends it to the terminate queue as well as deallocating the job from memory. Because the CPU class uses threading, it allows for the jobs to be executed much faster, allowing for substantially better performances. However, using threading made it much harder to debug as the system had a hard time stopping at certain break points due to the speed of execution.

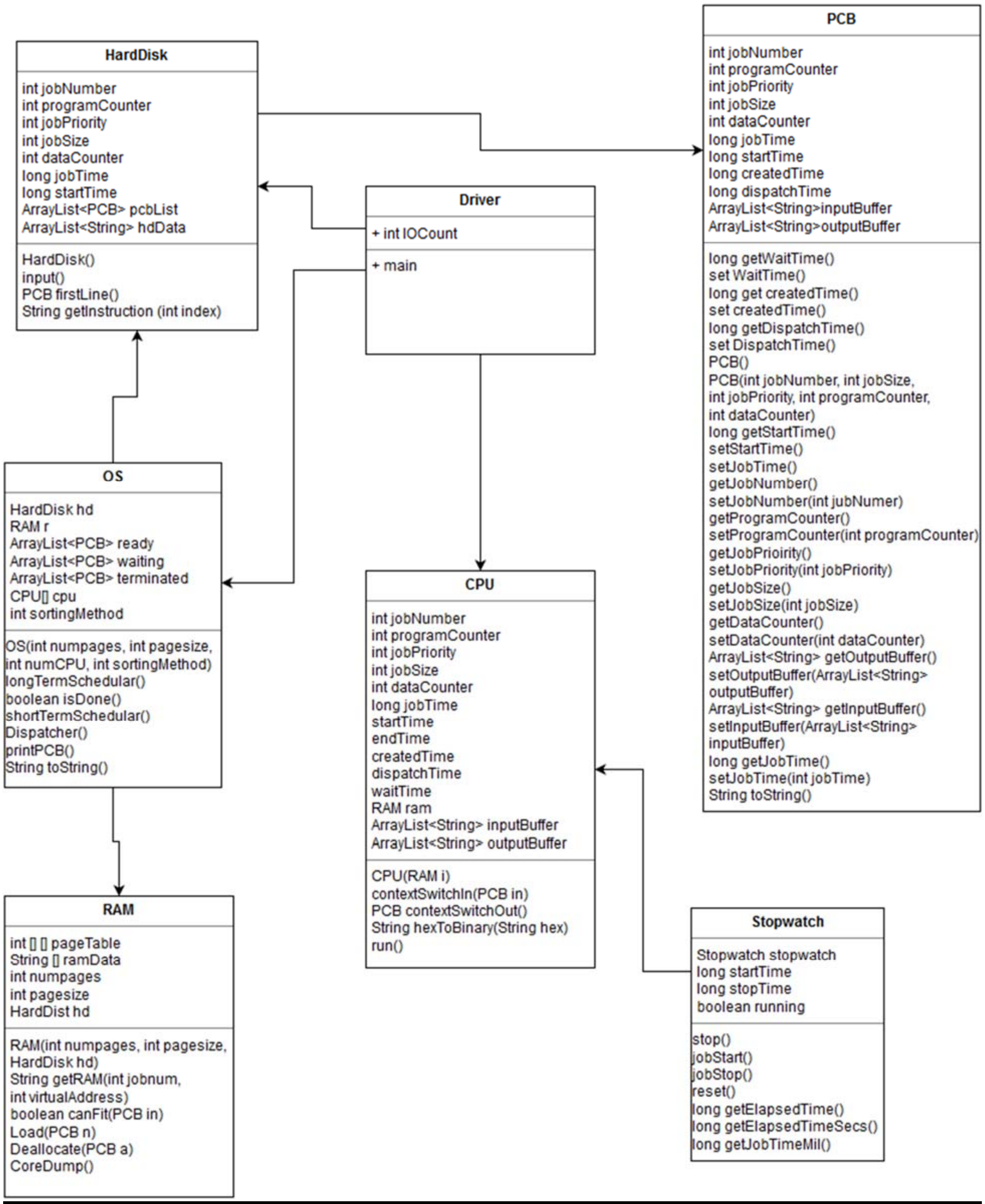


Figure 2Operating System Class Diagram

Simulation

Our group took the project not only as a learning experience for the fundamentals of an OS, but also as an experiment to see what variables within a computer cause bottlenecks and speed performance issues during job execution. The OS was designed in such a way that we could test different variables like page count and the number of CPUs to determine what some of the best case and worst case scenarios could be, as well as to determine what is physically going on in the system. To check these variables, we made several timers that allowed us to track and store times for each component in our system. Specifically, we timed the overall system time (from start to finish), the time of each jobs' completion, the time each jobs' was spent in ram, and the time that a given CPU was idle. We were able to take this data and store it to compare for different OS specifications. To make this easier, we set up 54 unique test cases where the number of pages, page size, number of CPUs, and scheduling algorithm varied. We then were able to make a chart showing the overall system time for each case. This allowed us to see best case and worse case scenarios for each instance so we could further investigate the cause for such outcomes. We were also able to see very interesting results pertaining to the CPU idle time compared to the average waiting time per job. These results will be discussed in the "Results" section below.

Data & Results

Due to the nature of pdf formatting, the graphs and data associated with this section are included separately in individual pdf documents. Please refer to these documents by their title given in this paper.

<u>Total Time</u>

This graph represents 54 unique data points that show the difference in total execution time for 4 differing variables: Number of pages, page size, number of CPUs, and the scheduling algorithm used. The table below shows the data points associated with each variable.

Variable	**Data Points**
Number of pages	4, 8, 16
Page Size	8, 16
Number of CPUs	1, 2, 4
Scheduling algorithm	1 (FCFS), 2 (Priority), 3 (SJF)

As you can see from the pdf document, there is a wide range of execution times. The worst case scenarios all include cases where there is only a single CPU, indicating a primary bottleneck. The next bottleneck seemed to be the combination of page count and page size. The best case scenarios for the OS happened when there were 4 CPUs with at least 8 pages with a page size of at least 16. For this particular case, the scheduling algorithm did not play a particularly large role in overall execution time.

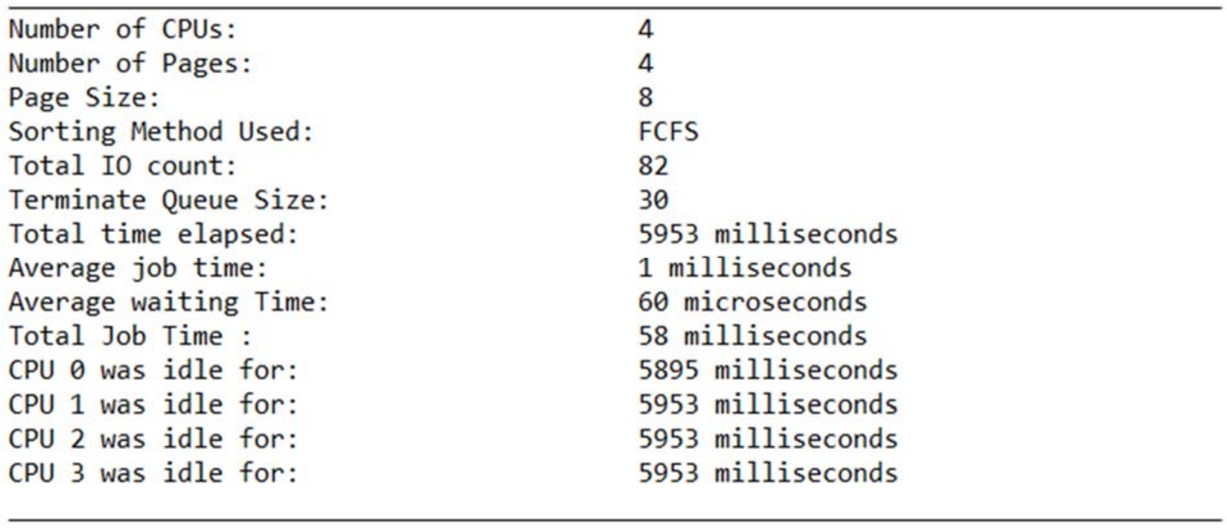
<u>Average Wait Time</u>

This graph represents the average time spent on the ready queue for each job using the same data points as mentioned above. As you can see from the chart, there are several cases where the average waiting time is very close to 0. We speculate that this is because there are several cases where the CPU(s) are finishing jobs so quickly that they are simply waiting for jobs to be loaded into memory. This results in the jobs hardly ever waiting in the ready queue. However, in cases where there was only one CPU, the average wait times were significantly higher because the CPU could only work on a single job at a time, leaving several jobs waiting in ram. The worst case for this instance was when we had the highest ram sizes and smallest CPU counts.

CPU Benchmark (Both Files)
Seeing as how the CPU was the primary bottleneck in both of the above cases, I decided to normalize all variables other than the CPU to see how much the CPU count actually affected the system. From the charts you can see the huge performance increases with multiple CPUs, but the system time levels off at around the eighth CPU. This leveling off is most likely due to inherent time restraints such as job execution times, print statements, and the java runtime environment itself.

Other Graphs
Other graphs included contain the RAM benchmarks as well as the scheduling algorithm benchmarks. The two RAM benchmarks variables included the page count and the page size. The chart indicates that, with 4 CPUs, the time continuously decreases as page count and size were increased. With the scheduling algorithm chart, we see that there is a difference in speed for each scheduling algorithm, but it is minor compared to the rest of the variables. The SJF algorithm is shown to be the most efficient with the given normalized variables.

Other Interesting Data
Another interesting data point not represented graphically was the idle times for a given CPU. This allowed us to see which CPUs were being utilized and which were mostly idle. We did this by comparing the CPU idle time to the total system time. For example, if the total system time was 1300 milliseconds and the CPU idle time was 1300 milliseconds, the given CPU was never used. Below is a more specific example.



```
Number of CPUs:                   4
Number of Pages:                  4
Page Size:                        8
Sorting Method Used:              FCFS
Total IO count:                   82
Terminate Queue Size:             30
Total time elapsed:               5953 milliseconds
Average job time:                 1 milliseconds
Average waiting Time:             60 microseconds
Total Job Time :                  58 milliseconds
CPU 0 was idle for:               5895 milliseconds
CPU 1 was idle for:               5953 milliseconds
CPU 2 was idle for:               5953 milliseconds
CPU 3 was idle for:               5953 milliseconds
```

The case above shows a system with a very small amount of RAM with four CPUs. As you can see, only the first CPU was ever used as only one job could fit in memory at any given time. So, when the first CPU is given a job to execute, the other CPUs have nothing to do as there are no jobs waiting in memory.

Conclusion

In conclusion, we have discovered that there are several components that make up an Operating System and that several of them impact the way the overall system runs. We have discovered that paging is a very important factor when it comes to RAM and increasing the number of pages and size of the pages can in fact enhance the performance of the system however the margin of improvement does not seem to be as large as we initially would have thought. We can also conclude that the scheduling algorithm also plays an important role in the overall speed with which the system performs but just like with paging, there doesn't seem to be a very significant change. From our data we can conclude that there are ways to change the performance of the operating system in a way that we maximize the throughput however there exists a point where such enhancements can actually lead to decreases in the performance of the system. In closing, we have learned that there are several components that go into an operating system and that all of those parts must be able to work together in order for the system to reach is maximum potential.