Github link: https://github.com/huhu72/Project-1

```
public class OS {
    static boolean status = false;
    static Dispatcher dispatcher = new Dispatcher();
    static CPU cpu = new CPU();
    public static void main(String[] args) {
        OS os = new OS();
        cpu.setDispatcher(dispatcher);
        Scheduler scheduler = new Scheduler();
        cpu.setScheduler(scheduler);
        Process p = new Process(cpu);
        Timer timer = new Timer();
        TimerTask tt = new TimerTask() {

            @Override
            public void run() {
                cpu.print();
            }

        };
        dispatcher.setScheduler(scheduler);
        try {
            p.createProcessesPrompt();
        } catch (FileNotFoundException e) {

            e.printStackTrace();
        }
        dispatcher.setCPU(cpu);
        dispatcher.setPCBList(cpu.getPCBList());
        dispatcher.setReadyQueue(cpu.getJobQueue());
        timer.scheduleAtFixedRate(tt,10,4000);
        cpu.runCommand();
```

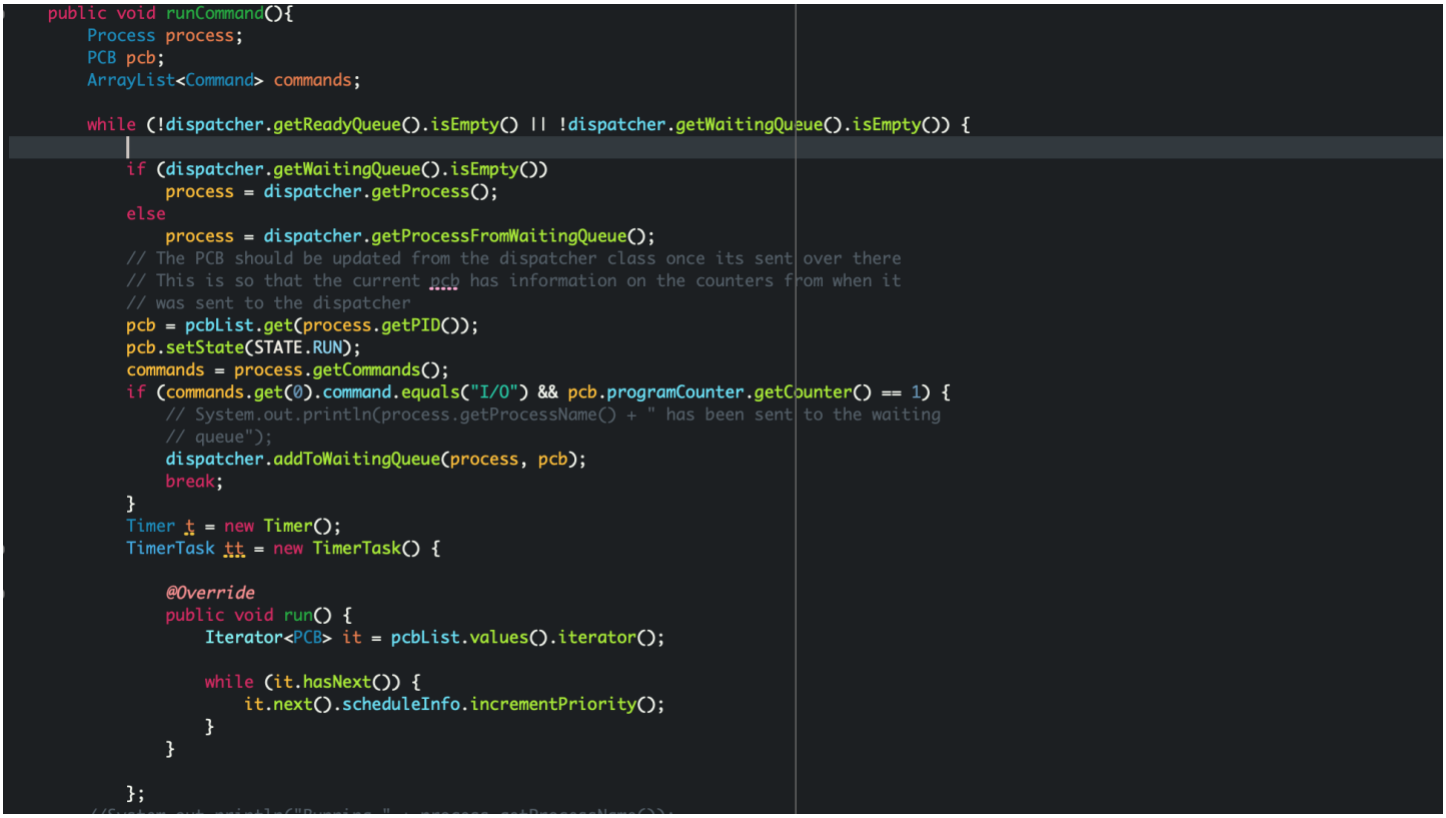
The OS first instantiates the CPU, Process, Dispatcher, and Scheduler.
After doing that, it will connect the dispatcher, CPU, and scheduler with each other. After connecting each other, it will then call on the method the process creation method, createProcesssesPrompt() to create all the processes that the user requests. It then puts all of those processes in the job queue within the cpu. Once that is done, it will grab the job queue and send it to the dispatcher class where the, unsorted, ready queue is then populated. Once the cpu starts running the processes, the OS will print out the status every 4ms after waiting 1ms after its called.

```
public void createProcessesPrompt() throws FileNotFoundException {
    Scanner input = new Scanner(System.in);
    String command;
    String argument;
    String [] arguments;
    //showTemplates();
    System.out.print("User: ");
    command = input.next();
    command = command.toLowerCase();
    if(command.equals("help")) showHelp();
    else if(command.equals("template")) {
        System.out.println();
        showTemplates();
    }else if(command.equals("create")) {
        argument = input.nextLine();
        arguments = argument.trim().split(" ");
        //getUserInput();
        for(int i = 0; i < arguments.length; i = i+2) {
            createProcesses(arguments[i] + ".txt", Long.parseLong(arguments[i+1]));

        }
    }
```

This method will take a users input as a bash command and then parse into an array of process arguments when the user types “create”. From there it will call createProcesses with the input of template name and how many processes they wish to create etc…

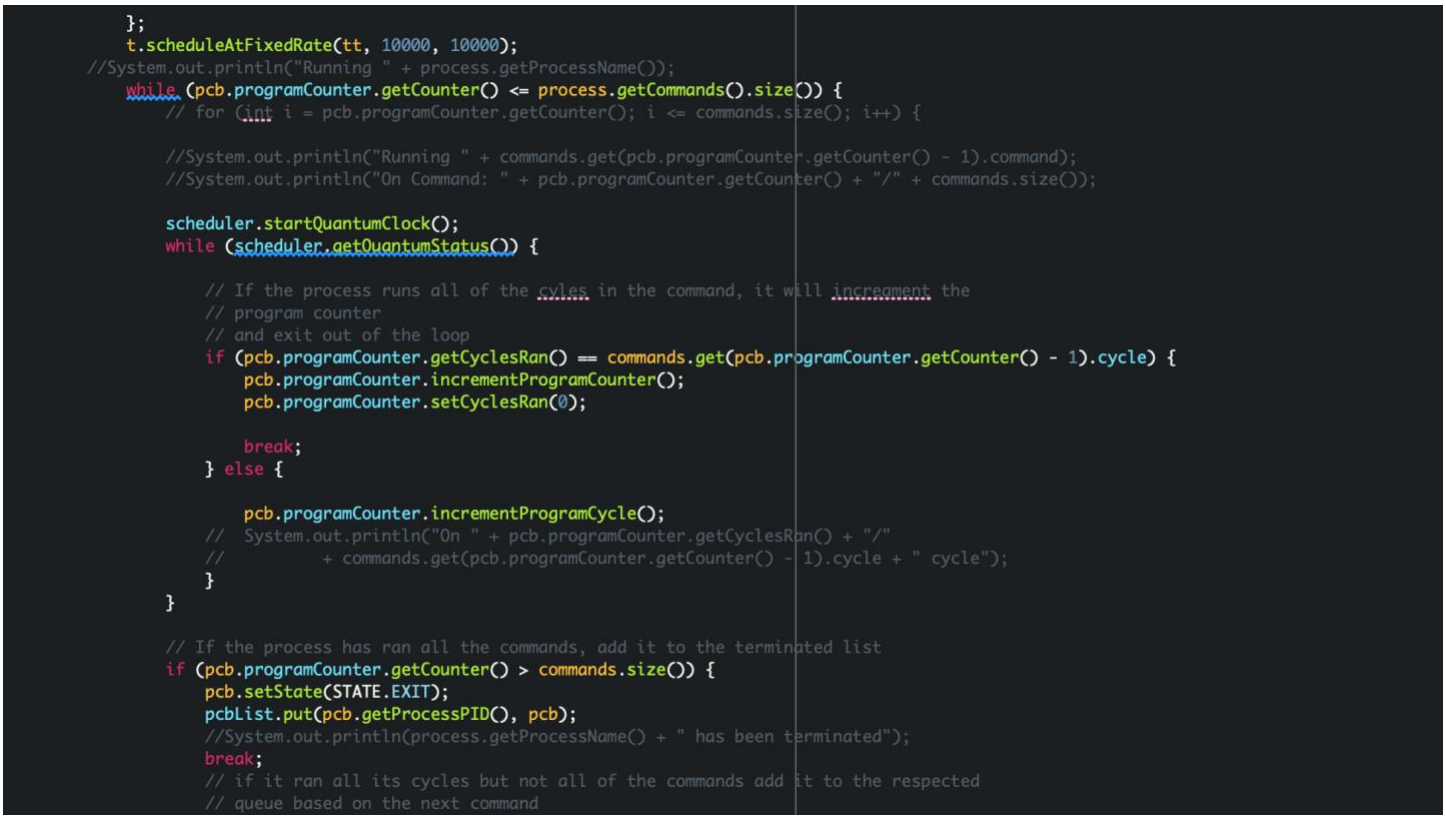
```
public void createProcesses(String templateName, long numProcessInput) throws FileNotFoundException {
    //String templateLocation = "./Templates/" + templateName;
    Process process;
    this.pid = 1;
    for (int i = 0; i < numProcessInput; i++) {
        createCommands(templateName);
        process = new Process("Process" + processCreationCounter,getCommands(), (this.pid + this.pidCounter), this.critStart, this.critEnd);
        this.pidCounter++;
        this.processCreationCounter++;
        //ncb = new PCB(process, this.cpu);
        this.cpu.addToProcessQueue(process);
        this.cpu.addPCB(new PCB(process));
        //System.out.println(this.pid);

    }

}
```

createProcesses will create a specified number of processes, based on user input and template, and create a process with a unique name and PID. After the process is created, it will then put newly created process into the job queue. Then I will create a PCB that is linked up with the process and adds it into a hashmap that is instantiated in the cpu.



```
public void runCommand(){
    Process process;
    PCB pcb;
    ArrayList<Command> commands;

    while (!dispatcher.getReadyQueue().isEmpty() || !dispatcher.getWaitingQueue().isEmpty()) {

        if (dispatcher.getWaitingQueue().isEmpty())
            process = dispatcher.getProcess();
        else
            process = dispatcher.getProcessFromWaitingQueue();
        // The PCB should be updated from the dispatcher class once its sent over there
        // This is so that the current pcb has information on the counters from when it
        // was sent to the dispatcher
        pcb = pcbList.get(process.getPID());
        pcb.setState(STATE.RUN);
        commands = process.getCommands();
        if (commands.get(0).command.equals("I/O") && pcb.programCounter.getCounter() == 1) {
            // System.out.println(process.getProcessName() + " has been sent to the waiting
            // queue");
            dispatcher.addToWaitingQueue(process, pcb);
            break;
        }
        Timer t = new Timer();
        TimerTask tt = new TimerTask() {

            @Override
            public void run() {
                Iterator<PCB> it = pcbList.values().iterator();

                while (it.hasNext()) {
                    it.next().scheduleInfo.incrementPriority();
                }
            }

        };
```

After all of the operation from the last section are completed, the OS will call this run command.
The cpu will first grab from the ready queue if the waiting queue is empty.
After grabbing the process, set the state to RUN, check if the first command is a I/O command so that it can put that process in the waiting queue, and start a timer that will increment the priority of all processes every 10 seconds after waiting 10 seconds when the timer is first called.



```
        };
        t.scheduleAtFixedRate(tt, 10000, 10000);
    //System.out.println("Running " + process.getProcessName());
        while (pcb.programCounter.getCounter() <= process.getCommands().size()) {
            // for (int i = pcb.programCounter.getCounter(); i <= commands.size(); i++) {

            //System.out.println("Running " + commands.get(pcb.programCounter.getCounter() - 1).command);
            //System.out.println("On Command: " + pcb.programCounter.getCounter() + "/" + commands.size());

            scheduler.startQuantumClock();
            while (scheduler.getQuantumStatus()) {

                // If the process runs all of the cyles in the command, it will increament the
                // program counter
                // and exit out of the loop
                if (pcb.programCounter.getCyclesRan() == commands.get(pcb.programCounter.getCounter() - 1).cycle) {
                    pcb.programCounter.incrementProgramCounter();
                    pcb.programCounter.setCyclesRan(0);

                    break;
                } else {

                    pcb.programCounter.incrementProgramCycle();
                //  System.out.println("On " + pcb.programCounter.getCyclesRan() + "/"
                //          + commands.get(pcb.programCounter.getCounter() - 1).cycle + " cycle");
                }
            }

            // If the process has ran all the commands, add it to the terminated list
            if (pcb.programCounter.getCounter() > commands.size()) {
                pcb.setState(STATE.EXIT);
                pcbList.put(pcb.getProcessPID(), pcb);
                //System.out.println(process.getProcessName() + " has been terminated");
                break;
                // if it ran all its cycles but not all of the commands add it to the respected
                // queue based on the next command
```

It will then enter a while loop that can only be ended using break;. Before running any cycles, it will start the time quantum clock that is handled by the scheduler. It will then enter another while loop that will increment the cycle of the current command the process is on until the scheduler has reached the specified round robin or the process has ran all of the cycles in that command and then will increment the program counter and set the cycle counter back to 0 and then break out of the while loop.

```
                pcb.programCounter.incrementProgramCounter();
                pcb.programCounter.setCyclesRan(0);

                break;
            } else {

                pcb.programCounter.incrementProgramCycle();
            //  System.out.println("On " + pcb.programCounter.getCyclesRan() + "/"
            //          + commands.get(pcb.programCounter.getCounter() - 1).cycle + " cycle");
            }
        }


        if (pcb.programCounter.getCounter() > commands.size()) {
            pcb.setState(STATE.EXIT);
            pcbList.put(pcb.getProcessPID(), pcb);
            //System.out.println(process.getProcessName() + " has been terminated");
            break;
            // if it ran all its cycles but not all of the commands add it to the respected
            // queue based on the next command
        } else if (pcb.programCounter.getCyclesRan() < commands.get(pcb.programCounter.getCounter() - 1).cycle
                && commands.get(pcb.programCounter.getCounter() - 1).command.equals("I/O")) {
            //System.out.println(process.getProcessName() + " has been sent to the waiting queue");
            dispatcher.addToWaitingQueue(process, pcb);
            break;

            // if there are still cycles to be ran in the current command and the command is
            // Calculate
            // Dont need to increment the program counter if the process still have cycles
            // to run but the pcb needs to be updated
            // in the dispatcher class
        } else if (pcb.programCounter.getCyclesRan() < commands.get(pcb.programCounter.getCounter() - 1).cycle
                && commands.get(pcb.programCounter.getCounter() - 1).command.equals("CALCULATE")) {
        //  System.out.println(process.getProcessName() + " has been sent to the ready queue");
            dispatcher.addToReadyQueue(process, pcb);
            break;
        } else {
```

The CPU will then decided if the process has anymore commands to run or if there are any more cycles that need to be ran. If there are no more commands to be ran, the cpu will then set the PCB associated to the process, change its state and update its map of PCB it currently holds. If it determines that a process needs to run more cycles, it will then add it to its respected queue in the dispatcher.

```
            // if there are still cycles to be ran in the current command and the command is
            // Calculate
            // Dont need to increment the program counter if the process still have cycles
            // to run but the pcb needs to be updated
            // in the dispatcher class
        } else if (pcb.programCounter.getCyclesRan() < commands.get(pcb.programCounter.getCounter() - 1).cycle
                && commands.get(pcb.programCounter.getCounter() - 1).command.equals("CALCULATE")) {
        //  System.out.println(process.getProcessName() + " has been sent to the ready queue");
            dispatcher.addToReadyQueue(process, pcb);
            break;
        } else {
            if (commands.get(pcb.programCounter.getCounter()) != null) {

                Command nextCommand = commands.get(pcb.programCounter.getCounter() - 1);
                pcb.programCounter.setCyclesRan(0);
                if (nextCommand.command.equals("CALCULATE")) {
                    //System.out.println(process.getProcessName()
                    //      + " has been sent to the ready queue based on the next command");
                    dispatcher.addToReadyQueue(process, pcb);
                    // break;
                } else {
                //  System.out.println(process.getProcessName() + " has been sent to the waiting  queue");
                    dispatcher.addToWaitingQueue(process, pcb);
                    // break;
                }
            }
            break;
        }

    }

    // }
    // in a couple of cycles, increment the priorities of all processes
    // break;
}
// print();
```

If there are more cycles to run, it will look at its next command and places the process in the respected queue and resets the processes cycle counter.

```
public void addToReadyQueue(Process process, PCB pcb) {
    pcbList.put(process.getPID(), pcb);
    this.cpu.updatePCBList(process,pcb);
    this.readyQueue.add(process);
    sortReadyQueue(this.readyQueue);
    this.readyQueue = scheduler.getReadyQueue();
}

public Queue<Process> getReadyQueue() {
    return this.readyQueue;
}

public Process getProcess() {
    Process processReturn = this.readyQueue.poll();
    //System.out.println(processReturn.getProcessName() + " is being sent to the cpu to run");
    return processReturn;
}

public void addToWaitingQueue(Process process, PCB pcb) {
    pcbList.put(process.getPID(), pcb);
    this.cpu.updatePCBList(process,pcb);
    this.waitingQueue.add(process);
    sortWaitingQueue(this.waitingQueue);
    this.waitingQueue = scheduler.getWaitingQueue();
}

public void sortWaitingQueue(Queue<Process> wq) {
    scheduler.setWaitingQueue(wq, this.pcbList);
    this.waitingQueue = scheduler.getWaitingQueue();
}

public Queue<Process> getWaitingQueue() {
    return this.waitingQueue;
}
```

The dispatchers addToWaiting/ReadyQueue will take the process that needs to be added to its respected queue, and calls apon the sort method which will pass that queue into the scheduler to be sorted.

```
public Process getProcess() {
    Process processReturn = this.readyQueue.poll();
    //System.out.println(processReturn.getProcessName() + " is being sent to the cpu to run");
    return processReturn;
}

public void addToWaitingQueue(Process process, PCB pcb) {
    pcbList.put(process.getPID(), pcb);
    this.cpu.updatePCBList(process,pcb);
    this.waitingQueue.add(process);
    sortWaitingQueue(this.waitingQueue);
    this.waitingQueue = scheduler.getWaitingQueue();
}

public void sortWaitingQueue(Queue<Process> wq) {
    scheduler.setWaitingQueue(wq, this.pcbList);
    this.waitingQueue = scheduler.getWaitingQueue();
}

public Queue<Process> getWaitingQueue() {
    return this.waitingQueue;
}

public Process getProcessFromWaitingQueue() {
    Process processReturn = this.waitingQueue.poll();
//  System.out.println(processReturn.getProcessName() + " is being sent to the cpu to run from the waiting queue");
    return processReturn;
}

public Scheduler getScheduler() {
    return this.scheduler;
}

public void setScheduler(Scheduler s) {
    this.scheduler = s;
}
```

When getting the process from either queue, it will grab the head of the queue and send that process back to the CPU and will also update the hashmap of the PCBs just incase there are anything changed within the PCB

```
Scheduler(){
    this.readyQueue = new PriorityQueue<>((p1,p2)->{
        return pcbInfo.get(p2.getPID()).getPriority() - pcbInfo.get(p1.getPID()).getPriority();
    });
    this.waitingQueue = new PriorityQueue<>((p1,p2)->{
        return pcbInfo.get(p2.getPID()).getPriority() - pcbInfo.get(p1.getPID()).getPriority();
    });
}
public void addToPQ(Process p) {
    this.readyQueue.add(p);
}
public void addPCBInfo(PCB pcb) {
    this.pcbInfo.put(pcb.getProcessPID(), pcb);
}
public Process getProcess() {
    return this.readyQueue.poll();
}
public void setReadyQueue(Queue<Process> rq, HashMap<Long,PCB> pcbList) {
    this.readyQueue = rq;
    this.pcbInfo = pcbList;
}
public Queue<Process> getReadyQueue(){
    return this.readyQueue;
}

public void setWaitingQueue(Queue<Process> waitingQueue, HashMap<Long,PCB> pcbList) {
    this.waitingQueue= waitingQueue;
    this.pcbInfo = pcbList;

}
public Queue<Process> getWaitingQueue(){
    return this.waitingQueue;
}
public Boolean getQuantumStatus() {
    return this.quantumStatus;
}
```

The scheduler will sort the queues by priority ,with the highest number being the most important.

```
    public void startQuantumClock() {
        quantumStatus = true;
        Timer timer = new Timer();
        TimerTask tt = new TimerTask() {

            @Override
            public void run() {
                quantumStatus = true;
                timer.cancel();
            }

        };
        timer.schedule(tt, 4);


    }

}
```

The timeQuantum clock will run for 4ms at max every time this method is called. Onces its called it will stop its self so that a new timer can run the next time it gets called

In this project, the round robin with priority scheduling algoritm was used. The quantum time was accuired by running 4 processes within the os and getting its average run time.

Commands:

- Help
- Template
- Create <template name> <number of desired processes> …