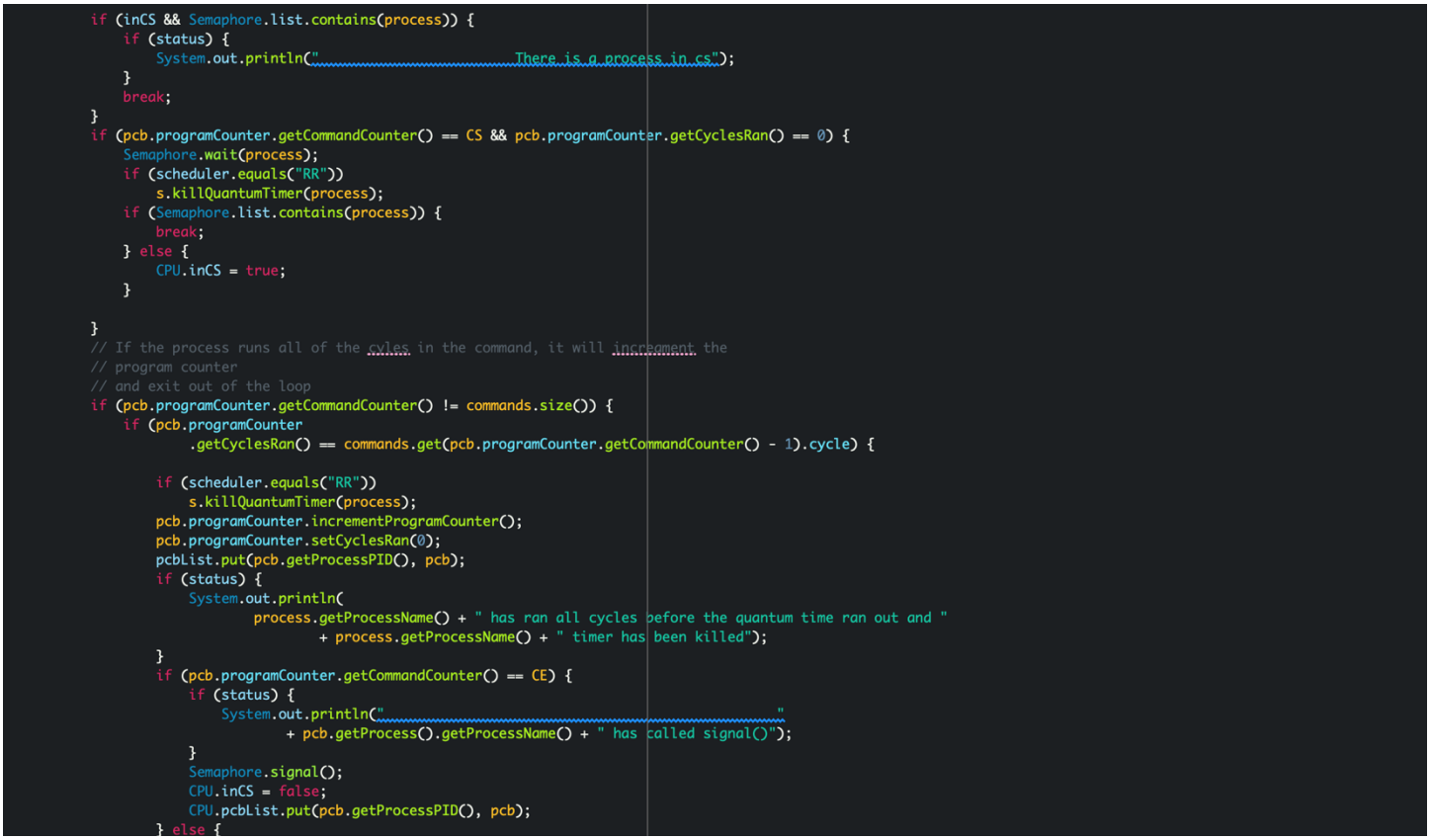
Github link: https://github.com/huhu72/Project-1

Grade using commit ID: `ff1a660ce2275b193522d8a05555299e6ac34fb0`

```java
/// All of the below should belong in the semaphore class
public static synchronized void wait(Process P) {
    if(CPU.status)System.out.println("                                                "+P.getProcessName() + " called wait()");
    Semaphore.value--;
    if (Semaphore.value < 0) {
        Semaphore.list.add(P);
        block(P);
        if(CPU.status)  System.out.println(P.getProcessName() + "has been sent to the semaphore waiting queue since S < 0");
    }
}

public static synchronized void signal() {

    Semaphore.value++;
    if (Semaphore.value <= 0) {
        Process P = Semaphore.list.poll();
        wakeUp(P);
        if(CPU.status)System.out.println(P.getProcessName() + "has been put back into the ready queue");
    }
}

private static void wakeUp(Process P) {
    PCB pcb = CPU.getPCB(P.getPID());
    pcb.setState(STATE.READY);
    CPU.updatePCBList(pcb);
    Dispatcher.addToReadyQueue(P, pcb);
}

private static void block(Process p) {
    Semaphore.list = Scheduler.sortSemaphoreWaitingQueue(Semaphore.list);
    PCB pcb = CPU.getPCB(p.getPID());
    pcb.setState(STATE.WAIT);
    CPU.updatePCBList(pcb);
}
```

This is the code for counting semaphores. If the value for the semaphore is < than 0 when wait it called, it will add that process in its list and retrieves it when signal is called.



```
        if (inCS && Semaphore.list.contains(process)) {
            if (status) {
                System.out.println("                    There is a process in cs");
            }
            break;
        }
        if (pcb.programCounter.getCommandCounter() == CS && pcb.programCounter.getCyclesRan() == 0) {
            Semaphore.wait(process);
            if (scheduler.equals("RR"))
                s.killQuantumTimer(process);
            if (Semaphore.list.contains(process)) {
                break;
            } else {
                CPU.inCS = true;
            }

        }
        // 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.getCommandCounter() != commands.size()) {
            if (pcb.programCounter
                    .getCyclesRan() == commands.get(pcb.programCounter.getCommandCounter() - 1).cycle) {

                if (scheduler.equals("RR"))
                    s.killQuantumTimer(process);
                pcb.programCounter.incrementProgramCounter();
                pcb.programCounter.setCyclesRan(0);
                pcbList.put(pcb.getProcessPID(), pcb);
                if (status) {
                    System.out.println(
                            process.getProcessName() + " has ran all cycles before the quantum time ran out and "
                                    + process.getProcessName() + " timer has been killed");
                }
                if (pcb.programCounter.getCommandCounter() == CE) {
                    if (status) {
                        System.out.println("                                                    "
                                + pcb.getProcess().getProcessName() + " has called signal()");
                    }
                    Semaphore.signal();
                    CPU.inCS = false;
                    CPU.pcbList.put(pcb.getProcessPID(), pcb);
                } else {
```

The signal and wait methods are called in the while loop, that increases the program count, when it reaches either the critical section (CS) or the end of it (CE).

```
public static Queue<Process> sortSemaphoreWaitingQueue(Queue<Process> list) {
    Queue<Process> sortedList = new PriorityQueue<>((p1, p2) -> {
        return Scheduler.pcbInfo.get(p2.getPID()).getPriority() - pcbInfo.get(p1.getPID()).getPriority();
    });
    sortedList.addAll(list);
    return sortedList;
}
```

Every time signal gets called, it is sorted by the scheduler, based on the priority, before being dispatched

```
    public void createProcessesPrompt() throws FileNotFoundException {
        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]));

            }
            Thread s = new Thread(new Status(this.cpu));
            s.start();

        }
```

A separate status thread is created and started after the user inputs the number processes they wish to create and those processes are created.

```
public class Status implements Runnable {
    Scanner scanner = new Scanner(System.in);
    CPU cpu;

    Status(CPU cpu) {
        this.cpu = cpu;
    }

    @Override
    public void run() {
        String input;
        while (true) {
            System.out.print("Command: ");
            input = scanner.nextLine();
            input.toLowerCase().trim();
            if (input.equals("status")) {
                CPU.status = true;
                CPU.print();
                CPU.status = false;
            }
        }

    }

}
```

The Status thread will continuously to listen to user input. If the user types status, the thread will print everything that is happening on the CPU as well as the status for all the processes. In the CPU class, there are segments of print commands that will only print if the status variable in the cpu is true.

```
if (CPU.inCS) {
    processThreadArray[0] = new Thread(dispatchProcess());
    for (int i = 1; i < 4; i++) {
        processThreadArray[i] = new Thread(dispatchProcess());
    }
    for (Thread t : processThreadArray) {
        t.start();
    }
} else {
    for (int i = 0; i < 4; i++) {
        processThreadArray[i] = new Thread(dispatchProcess());
    }
    for (Thread t : processThreadArray) {
        t.start();
    }
}
```

4 process thread will be created when the cpu first runs. First, it grabs a process from the respected queue and adds it into a Runnable ArrayList. Then it will assign the runnable to a thread and runs it

```
int min = 1;
int max = 10;
int randomNum = (int) Math.floor(Math.random() * (max - min + 1) + min);
this.pid = 1;
for (int i = 0; i < numProcessInput; i++) {
    createCommands(templateName);
    process = new Process("Process" + processCreationCounter, getCommands(), (this.pid +
            this.critStart, this.critEnd);
    this.pidCounter++;
    this.processCreationCounter++;

    if (randomNum == 1) {
        Process childProcess = fork();
        process.setChildPID(childProcess.getPID());
    }
```

Fork() is called randomly by using a random number generator.

```
    public Process fork() throws FileNotFoundException {
        createCommands("child.txt");
        Process childProcess = new Process("Process" + processCreationCounter, getCommar
                (this.pid + this.pidCounter), this.critStart, this.critEnd);
        childProcess.setParentPID(this.getPID());
        this.pidCounter++;
        this.processCreationCounter++;
        this.cpu.addToProcessQueue(childProcess);
        PCB childPCB = new PCB(childProcess);
        childPCB.setParentPID(childProcess.getParentPID());
        this.cpu.addPCB(childPCB);
        return childProcess;
    }
```

Fork() will create a new process using the child.txt template and will attach the parents PID and attach its own pid to the parent for referencing purposes. After the child is created, it will add it to the ready queue.

```
    // If the process is out of commands to run
    if (pcb.programCounter.getCommandCounter() > commands.size()) {
        pcb.setState(STATE.EXIT);
        CPU.pcbList.put(pcb.getProcessPID(), pcb);
        // System.out.println(pcb.getChildPID());
        if (pcbList.get(pcb.getChildPID()) != null) {
            PCB childPCB = pcbList.get(pcb.getChildPID());
            childPCB.programCounter.setCounter(3);
            childPCB.setState(STATE.EXIT);
            CPU.pcbList.put(childPCB.getProcessPID(), childPCB);
            if (status) {
                System.out.println(process.getProcessName() + " and its child "
                        + childPCB.getProcess().getProcessName() + "has been terminated");
            }
        } else {
            if (status) {
                System.out.println(process.getProcessName() + " been terminated");
            }
```

When a process is terminated, it will also terminate its child to fulfill cascading termination

```
public static Runnable dispatchProcess() {
    Runnable runnableProcess;
    Process process;
    // Grabs from the ready queue if the waiting queue is empty(Usually when the cpu
    // is first initiated)
    if (Dispatcher.getWaitingQueue().isEmpty() && !Dispatcher.getReadyQueue().isEmpty()) {
        process = Dispatcher.getProcess();
        runnableProcess = process;
        if (process != null) {
            if (CPU.status) {
                System.out.println("                    The waiting queue is empty, grabbing "
                        + process.getProcessName() + " from the ready queue");
            }
            return runnableProcess;
        } else {
            return null;
        }

        /*
         * If both the ready and waiting queue is empty, then signal the semahpore so
         * that it can remove it from its own waiting queue and populate it into the
         * ready queue. This is needed just in-case there aren't enough signal calls to
         * pull all the Processes out of its own waiting queue
         */
    } else if (Dispatcher.getWaitingQueue().isEmpty() && Dispatcher.getReadyQueue().isEmpty()) {
        Semaphore.signal();
        process = Dispatcher.getProcess();
        runnableProcess = process;
        if (process != null) {
            if (status) {
                System.out
                        .println("                    The waiting and ready queue is empty is empty, grabbing "
                                + process.getProcessName() + " from the ready queue after calling signal");
            }
            return runnableProcess;
        } else {
            return null;
        }
     } else {
         process = Dispatcher.getProcessFromWaitingQueue();
         runnableProcess = process;
         if (process != null) {
             if (status) {
                 System.out.println(
                         "                    Grabbing " + process.getProcessName() + " from the wating queue");
             }
             return runnableProcess;
         } else {
             return null;
         }

     }

 }
```

This is the code for grabbing a process from the respected queues

```
if (!Dispatcher.getReadyQueue().isEmpty() || !Dispatcher.getWaitingQueue().isEmpty()
        || !Semaphore.list.isEmpty()) {
    // System.out.println(" Thread is being re assigned from the terminated stage");
    runnableProcess = dispatchProcess();
    currentThread = new Thread(runnableProcess);
    if (runnableProcess != null)
        currentThread.start();
}
// if it ran all its cycles but not all of the commands, add it to the respected
```

This is the code for re-referencing the process threads. This segment of code will run when the process is out of commands to run. It will also run when the process it being put into a queue.

```
if (randomNum == 1) {
    if (CPU.status) {
        System.out.println("Process is put to sleep");
    }
    Thread.sleep(10);
}
Timer interruptTimer = new Timer();
TimerTask interruptTimerTask = new TimerTask() {

    @Override
    public void run() {
        if (processThreadArray[randomNum].getState() == Thread.State.TIMED_WAITING) {
            processThreadArray[randomNum].interrupt();
            if (status) {
                System.out.println("Interrupt thrown");
            }
        }

    }
```

The thread will put itself to sleep for 1 second, pausing what its currently doing, randomly and then will wake up by calling interrupt()

```
        Process p = new Process(cpu);
        try {
            p.createCompareProcesses();
        } catch (FileNotFoundException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        // Get total number of cycles in all 100 processes that will be used to compare
        // the schedulers

        Dispatcher.setPCBList(cpu.getComparePCBList());
        Dispatcher.setReadyQueue(cpu.getCompareQueue());

        int RRCycles = cpu.compareRR();
        cpu.compareQueue = new LinkedList<Process>();
        CPU.comparePCBList = new HashMap<>();
        try {
            p.createCompareProcesses();
        } catch (FileNotFoundException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        Dispatcher.setPCBList(cpu.getComparePCBList());
        Dispatcher.setReadyQueue(cpu.getCompareQueue());
        int PQCycles = cpu.comparePQ();
        if(RRCycles < PQCycles) {
            CPU.scheduler = "PQ";

        }else {
            CPU.scheduler = "RR";

        }
```

```
public void createCompareProcesses() throws FileNotFoundException {
    Process process;
    long pid = 1;
    long pidCounter = 0;
    int processCreationCounter = 1;
    for (int i = 0; i < 1000; i++) {
        createCommands("compare.txt");
        process = new Process("Process" + processCreationCounter, getCommands(), (pid + pidCounter),
                this.critStart, this.critEnd);
        pidCounter++;
        processCreationCounter++;
        this.cpu.addToCompareQueue(process);
        PCB pcb = new PCB(process);
        this.cpu.addToComparePCBList(pcb);
    }
    //System.out.println(this.cpu.getCompareQueue());

}
```

```java
    public int compareRR() {
        int totalCyclesRanForRR = 0;
        Timer compareTimer = new Timer();
        TimerTask compareTimerTask = new TimerTask() {

            @Override
            public void run() {
                compare = false;
            }

        };
        compareTimer.schedule(compareTimerTask, 30);
        // Using round robin first
        while (compare && !Dispatcher.getReadyQueue().isEmpty()) {

            Process p = Dispatcher.getProcess();
            if (p != null) {
                // System.out.println(p);
                PCB pcb = comparePCBList.get(p.getPID());
                ArrayList<Command> commands = p.getCommands();
                Scheduler s = new Scheduler();
                s.run(p);
                while (s.getQuantumStatus() && pcb.programCounter.getCyclesRan() < commands.get(0).cycle) {
                    pcb.programCounter.incrementProgramCycle();
                    totalCyclesRanForRR++;
                    CPU.comparePCBList.put(pcb.getProcessPID(), pcb);

                    if (pcb.programCounter.getCyclesRan() < commands.get(0).cycle) {
                        Dispatcher.addToReadyQueue(p, pcb);
                    }
                }

            }

        }

        return totalCyclesRanForRR;
    }
```

```java
    public int comparePQ() {
        compare = true;
        int totalCyclesRanForPQ = 0;
        Timer compareTimer = new Timer();
        TimerTask compareTimerTask = new TimerTask() {

            @Override
            public void run() {
                compare = false;
            }

        };
        compareTimer.schedule(compareTimerTask, 30);
        // Using round robin first
        while (compare && !Dispatcher.getReadyQueue().isEmpty()) {
            Process p = Dispatcher.getProcess();
            if (p != null) {
                PCB pcb = comparePCBList.get(p.getPID());
                ArrayList<Command> commands = p.getCommands();

                while (pcb.programCounter.getCyclesRan() < commands.get(0).cycle) {
                    pcb.programCounter.incrementProgramCycle();
                    totalCyclesRanForPQ++;
                    CPU.comparePCBList.put(pcb.getProcessPID(), pcb);
                }
            }
        }

        return totalCyclesRanForPQ;

    }

    public static void updateComparePCBList(Process p, PCB pcb) {
        comparePCBList.put(p.getPID(), pcb);

    }

}
```

The CPU decides what scheduler to use, (Round robin vs priority queue) by first creating 1000 processes based on the compare.txt template. Then it will first run the round robin scheduler to see how many cycles it can run within 3 seconds. Then It will do the same thing with the priority queue scheduler. Which ever scheduler runs the most cycles will be the scheduler that the cpu will use

```
Process(String processName, ArrayList<Command> commands, long pid, int critStart, int critEnd) {
    this.processName = processName;
    this.processCommands = commands;
    this.pid = pid;
    this.critStart = critStart;
    this.critEnd = critEnd;
    this.timeLimit = 4;
    this.priority = (int) ((Math.random() * (10 - 1)) + 1);
    this.memory = (int) ((Math.random() * (1024 - 1)) + 1);
}
```

```
    this.cpu.addToProcessQueue(process);
    PCB pcb = new PCB(process);
    pcb.setChildPID(process.childPID);
    this.cpu.addPCB(pcb);
    memoryCount += process.memory;
    if(memoryCount>TOTAL_MEMORY) {
        Dispatcher.addToReadyQueue(process, pcb);
    }

}
```

```
        if (status) {
            System.out.println(process.getProcessName() + " and its child "
                    + childPCB.getProcess().getProcessName() + "has been terminated");
            Process.memoryCount -= process.memory-childPCB.getProcess().memory;
        }
    } else {
        if (status) {
            Process.memoryCount -= process.memory;
            System.out.println(process.getProcessName() + " been terminated");
        }
    }
```

When a process gets create, it will randomly assign memory size from 1 – 1024 to the process. After the creation, it will take the process' memory and add it to the total memory used. If the total memory is > the max memory allowed (1024), it will directly add it to the ready queue. As a process is being terminated, it will subract its memory, and its childs if it exists, from total memory

Commands:

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

Requirements:

- Atleast java 8