```java
import java.util.LinkedList;

public class CPUScheduler
{

    /**
     * VARIABLES*****************************************************************
     */
    final static int TIMESLICE =200;
    final int RUN_WAIT = 1000;
    LinkedList<Integer> queue;
    int runningJob;
    int slice;

    /**
     * CONSTRUCTOR***************************************************************
     */
    CPUScheduler ()
    {

        queue   = new LinkedList<Integer>();
        runningJob = -1;
        slice   = TIMESLICE;
    }

    /**
     * PRIVATE METHODS***********************************************************
     */
    /**
     * Returns the time remaining in slice for current job, checks against max
     * CPU time
     * @param  jobID        jobID of job to be queried
     * @param  currentSlice the jobs current slice time
     * @return              slice time for given job
     */
    int getSlice(int jobID, int currentSlice)
    {
        if (JobTable.getTimeLeft(jobID) < currentSlice) {
            return JobTable.getTimeLeft(jobID);
        }
        else {
            return currentSlice;
        }
    }

    /**
     * PUBLIC METHODS************************************************************
     */

    /**
     * Blocks currently running job by adding to blocked queue and
     * setting to unready in jobtable
     */
    public void block ()
    {
        JobTable.setBlocked(runningJob);
        JobTable.unsetReady(runningJob);
```

```
    runningJob = -1;
}

/**
 * Gets the currently running (head of queue) jobID and returns
 * If the queue is empty returns -1
 * @return int jobID or -1 if no job running
 */
public int current ()
{
    return runningJob;
}

/**
 * Sends next process to cpu by moving head element to end of queues
 * Checks to see if currently running job needs to be terminated ( over max
 * run time)or swapped from memory (over allowed time in memory)
 * Then updates a, p to return to sos
 */
public int[] next (int[] a, int[] p)
{
    // Will
    int[] returnVars = {-1, -1}; // {freeMemory, swapOut}

    if (runningJob != -1) {

        // If time remains in slice, continue
        if (slice > 0) {
            slice = getSlice(runningJob, slice);
            // Running stays the same
            // System.out.println("-CPUScheduler resumes Job " + runningJob
            //  + " with " + slice + " remaining");
        }
        // Check to see if job has exceeded its max CPU time
        // If it has, then need to free it's memory and terminate it
        else if (JobTable.getTimeLeft(runningJob) <= 0) {
            // System.out.println("-CPUScheduler stops Job " + runningJob +
            //  " (exceeds max CPU time)");
            returnVars[0] = runningJob;
            JobTable.terminate(runningJob);
            JobTable.unsetReady(runningJob);
            runningJob = -1;
        }
        // If no time remains, check if job has exceeded max time in
        // memory. If it has, then need to lower its priority and return
        // to memManager for potential swapout
        else if ((os.currentTime - JobTable.getPriorityTime(runningJob))
            >= RUN_WAIT) {
            if (!JobTable.doingIO(runningJob) && queue.size() > 4)
            {
                returnVars[1] = runningJob;
                //JobTable.lowerPriority(runningJob);
                JobTable.unsetReady(runningJob);
            }
            else {
                queue.add(runningJob);
                //queue.lowerPriority(runningJob);
```

```
            }
            runningJob = -1;
        }
        // The job has no slice remaining and needs to be put at back of
        // queue
        else {
            queue.add(runningJob);
            runningJob = -1;
        }
    }
    // If there is no running job yet
    if (runningJob == -1 && !queue.isEmpty()) {
        runningJob = queue.remove();
        // System.out.println("Next job = " + runningJob);
        slice = getSlice(runningJob, TIMESLICE);
    }
    // If there is absolutely nothing in the queues
    if (runningJob == -1) {
        // Set CPU to idle
        a[0] = 1;
    }
    else {
        a[0] = 2;
        p[1] = runningJob;
        p[2] = JobTable.getAddress(runningJob);
        p[3] = JobTable.getSize(runningJob);
        p[4] = slice;
    }
    return returnVars;
}

/**
 * Prints details of CPU Queues
 */
public void print ()
{
    // System.out.println("-CPU Report");
    // System.out.println("--In CPU  : " + runningJob);
    // System.out.println("");
}

/**
 * Provides the size of the ready queue
 * @return the size of the ready queue
 */
public int queueSize()
{
    return queue.size();
}

/**
 * Adds job to appropriate ready queue (will remove from blocked if not
 * a new job)
 * @param jobID unique identifier for jobs
 */
public void ready (int jobID)
{
```

```
        // If the job is not blocked
        if(!JobTable.isBlocked(jobID) && !JobTable.isReady(jobID) &&
            !JobTable.isTerminated(jobID)) {
            // Then add to appropriate ready queue
            queue.add(jobID);
            JobTable.setReady(jobID);
            // System.out.println("-CPUScheduler readies job " + jobID);
        }
        print();
    }

    /**
     * Terminates currently running job
     * @return jobID of terminated job
     */
    public int terminate ()
    {

        int killedJob = runningJob;
        runningJob = -1;
        // Sets to terminated in jobTable
        JobTable.terminate(killedJob);
        JobTable.unsetReady(killedJob);

        // System.out.println("-CPUScheduler terminates job " + killedJob);

        return killedJob;
    }

    /**
     * Updates current running jobs time
     */
    public void update()
    {
        int timeElapsed = os.currentTime - os.lastTime;
        // Increments interrupted job's time time & current slice
        if (runningJob != -1) {
            JobTable.incrementTime(runningJob, timeElapsed);
            slice = slice - timeElapsed;
        }
    }
}
```