CISC 3320 – Operating Systems

Fall 2013 – Prof. Jones

OS Project Submission

Group Members:

- Meagan Fenisey-Copes
- Jun Han Ooi
- Michael Lishnevsky
- Colene McFarlane

Included Files:

Source:

- os.java
- Job.java
- JobTable.java
- MemoryManager.java
- Swapper.java
- CPUScheduler.java
- IOScheduler.java

Output

```
import java.util.LinkedList;

public class os
{
    /**
     * VARIABLES*****************************************************************
     */
    public final static int MEMSIZE = 100;
    public final static int TIMESLICE = 200;
    public static int lastTime;
    public static int currentTime;
    public static JobTable jobTable;
    public static Dispatcher dispatcher;
    public static CPUScheduler cpuScheduler;
    public static IOScheduler ioScheduler;
    public static MemoryManager memoryManager;
    public static Swapper swapper;

    /**
     * NESTED CLASS DISPATCHER*************************************************
     */
    // This is used to update sos and report the current
    // state of the system
    static class Dispatcher
    {
        public void update (int[] a, int[] p)
        {
            // Get time details
            lastTime = currentTime;
            currentTime = p[5];
            // System.out.println("LAST    TIME: " + lastTime);
            // System.out.println("CURRENT TIME: " + currentTime);
            memoryManager.freeTerminated();
        }

        /**
         * changes parameters for a, p to return back to sos
         * @param a cpu status
         * @param p process details
         */
        public void report (int[] a, int[] p)
        {
            // System.out.println("\n*****REPORTS******");
            // Setting the sos's a, p values
            // If there is no job, set to idle
            // if (a[0] == 1)
            // {
            //  System.out.println("-Dispatcher has no job to run");
            // }
            // // If there is a job, set to run,
            // // Update p with address, size
            // else
            // {
            //  System.out.println("-Dispatcher job report");
            //  System.out.println("--Job ID      : " + p[1]);
            //  System.out.println("--Job Address : " +
            //      JobTable.getAddress(p[1]));
```

```
        //  System.out.println("--Job Size    : " +
        //      JobTable.getSize(p[1]));
        //  System.out.println("--Slice Time  : " + p[4]);
        //  System.out.println("--CPU Time    : " +
        //      JobTable.getCurrentCPUTime(p[1]));
        //  System.out.println("--Max CPU Time: " +
        //      JobTable.getMaxCPUTime(p[1]));
        //  System.out.println("--Time Left   : " +
        //      jobTable.getTimeLeft(p[1]));
        // }
        // Prints system status
        cpuScheduler.print();
        jobTable.print();
        memoryManager.print();
        swapper.print();
        ioScheduler.print();
    }
}
/**
 * PUBLIC METHODS************************************************************
 */
// Used to separate events in OS
public static void div()
{
    // System.out.println(
    //  "-------------------------------------------------");
}

// Called by SOS, used to initialize variables
public static void startup ()
{
    jobTable = new JobTable();
    dispatcher = new Dispatcher();
    cpuScheduler = new CPUScheduler();
    ioScheduler = new IOScheduler();
    memoryManager = new MemoryManager();
    swapper = new Swapper();
    lastTime = 0;
    currentTime = 0;
    div();

    // Turn off tracing for submission
    sos.offtrace();
}

/**
 * Called at end of each interrupt to check if new swapping should take place
 * @param a from sos
 * @param p from sos
 */
public static void rescan (int[] a, int[] p)
{
    // System.out.println("****RESCAN****");

    // Moves CPU to next job in queue if slice done
    // returnVars[0] : Job exceeding maxTime (needs memory freed)
    // returnVars[1] : Job exceeding priority time
```

```
    int[] returnVars = cpuScheduler.next(a, p);
    // If job exceeds max runtime, add to terminated llist
    memoryManager.newTerminated(returnVars[0]);
    // Checks if a blocked job is ready to be swappedout
    swapper.swapOut(ioScheduler.readyToLeave());
    // Checks if a blocked job is ready to be swapped in
    swapper.swapIn(memoryManager.add(ioScheduler.readyToReturn()));
    // Swaps out job that has exceeded max memory time
    swapper.swapOut(returnVars[1]);
    // Adds new job to swap queue
    swapper.swapIn(memoryManager.find());
    // Initiates any swapping
    ioScheduler.moveIO(swapper.swap());
    // Initiates any I/O
    ioScheduler.ioCheck();
}

/**
 * Accepts new job into system
 * @param []a to be modified for sos
 * @param []p p[1]: job number, p[2]: job priority,
 *            p[3]: job size (in kb) p[4]: maximum CPU time,
 *            p[5]: current time
 *            to be modified for sos
 */
public static void Crint (int []a, int []p)
{
    // System.out.println("CRINT START");
    // Update system times
    dispatcher.update (a, p);
    cpuScheduler.update();

    int jobID = p[1];
    // Adds to JobTable
    jobTable.add(p);
    // If room found in memory, add to swap queue
    swapper.swapIn(memoryManager.add(jobID));

    rescan(a, p);
    // Report
    dispatcher.report (a, p);

    // System.out.println("CRINT FINISH");
    div();
}

/**
 * An I/O operation has finished
 * @param []a to be modified for sos
 * @param []p to be modified for sos
 */
public static void Dskint (int []a, int []p)
{
    // System.out.println("DSKINT START");
    // Update system times
    dispatcher.update (a, p);
    cpuScheduler.update();
```

```
    // Gets jobID of completed I/O
    // and (if there is a job in I/O which is not in memory)
    // the jobID of the job which need to be brought in
    int jobID = ioScheduler.ioDone();
    // Will ready job if necessary
    cpuScheduler.ready(jobID);
    // Move IO if status changed
    ioScheduler.moveIO(jobID);
    // If there is a job that needs memory
    // Place it in the memory queue
    //swapper.swapIn(memoryManager.add(jobNeedsMemory));

    rescan(a, p);
    // Report
    dispatcher.report (a, p);

    // System.out.println("DSKINT FINISH");
    div();
}

/**
 * Memory swap complete
 * @param []a to be modified for sos
 * @param []p to be modified for sos
 */
public static void Drmint (int []a, int []p)
{
    // System.out.println("DRMINT START");
    // Update system times
    dispatcher.update (a, p);
    cpuScheduler.update();

    // Gets completed memory swap from swapper
    int jobID = swapper.swapDone();

    // Gets completed swap Direction
    int direction = jobTable.getDirection(jobID);
    // If swapped into memory, ready job
    if (direction == 0) {
        // Adds to memoryManager inMemory list
        memoryManager.addToMemory(jobID);
        // Adds to cpu ready list
        cpuScheduler.ready(jobID);
    }
    // Otherwise, free the space
    else if (direction == 1) {
        memoryManager.free(jobID);
    }
    // Moves the jobs I/O if status changes
    ioScheduler.moveIO(jobID);

    rescan(a, p);
    // Report
    dispatcher.report (a, p);

    // System.out.println("DRMINT FINISH");
```

```
    div();
}

/**
 * Timeslice ended
 * @param []a to be modified for sos
 * @param []p to be modified for sos
 */
public static void Tro (int []a, int []p)
{
    // System.out.println("TRO START");
    // Update system times
    dispatcher.update (a, p);
    cpuScheduler.update();

    rescan(a, p);
    // Report
    dispatcher.report (a, p);

    // System.out.println("TRO FINISH");
    div();
}

/**
 * Running Job is requesting service
 * @param []a to be modified for sos
 * @param []p to be modified for sos
 */
public static void Svc (int []a, int []p)
{
    // System.out.println("SVC START");
    // Update system times
    dispatcher.update (a, p);
    cpuScheduler.update();

    // The job is requesting termination
    if (a[0] == 5) {
        // System.out.println("Requesting termination");
        int jobID = cpuScheduler.terminate();
        jobTable.setDirection(jobID, -1);
        memoryManager.newTerminated(jobID);
        // Moves IO to terminated queue
        ioScheduler.moveIO(jobID);
    }
    // The job is requesting another I/O operation
    else if (a[0] == 6) {
        // System.out.println("Requesting another i/o operation");
        int jobID = cpuScheduler.current();
        ioScheduler.add(jobID);
    }
    // The job is requesting to be blocked until all pending
    // I/O requests are completed
    else if (a[0] == 7) {
        // System.out.println(
        //  "Block until all pending I/O requests are completed");
        int jobID = cpuScheduler.current();
        // If job is using I/O, block, but don't free
```

```
            if (ioScheduler.doingIO(jobID)) {
                // System.out.println("-I/O: Job is doing I/O");
                cpuScheduler.block();
                ioScheduler.moveIO(jobID);
            }
            // If jobs are pending, block and free
            else if (jobTable.getIO(jobID) > 0) {
                // System.out.println("-I/O: Job has pending I/O");
                cpuScheduler.block();
                if (memoryManager.smartSwap()) {
                    swapper.swapOut(jobID);
                }
                ioScheduler.moveIO(jobID);
            }
            // If job not using I/O and no pending I/O, ignore
            else {
                // System.out.println("-I/O: Job has no pending I/O");
            }
        }

        rescan(a, p);
        // Report
        dispatcher.report (a, p);

        // System.out.println("SVC FINISH");
        div();
    }
}
```

```java
public class Job
{
//      p[1]: job number p[2]: job priority
//      p[3]: job size (in kb) p[4]: maximum CPU time
//      p[5]: current time
        // This will model after job given by sos in Crint
    int idNum;      // The job's id number
    int size;           // Size of job in Memory-to-Drum
    int maxCPUTime; // Maximum time to run in CPU- terminate after
    int address;        // The address of the job if in memory
    int currentCPUTime; // Time ran in CPU
    int pendingIO;      // Number of I/O requests pending
    int direction;  // 0 = Drum-to-Memory, 1 = Memory-to-Drum, -1 = No Swap
    boolean inMemory;   // True is job is in memory
    boolean terminated; // True if job has terminated
    int priorityTime;   // Amount of time job has been in memory
    boolean latched;    // doing IO
    boolean ready;      // Job is in CPU ready queue
    boolean blocked;    // Job is blocked /(waiting for I/O)
    boolean swapped;    // Job has been swapped out of memory at least once
    boolean inSwapQueue;// Job is waiting to be swapped /(in or out)
    boolean inDrum; // Currently swapping into/out of memory

    /**
     * CONSTRUCTOR
     */
    Job (int[] p)
    {
        // From sos
        idNum = p[1];
        size = p[3];
        maxCPUTime = p[4];
        priorityTime = p[5];
        // Job defaults
        address = -1;
        currentCPUTime = 0;
        inMemory = false;
        pendingIO   = 0;
        direction   = -1;
        terminated  = false;
        latched = false;
        ready = false;
        blocked = false;
        swapped = false;
        inSwapQueue = false;
        inDrum = false;

    }
}
```

```
import java.util.LinkedList;

public class JobTable
{
    /**
     * VARIABLES*****************************************************************
     */
    // Table will store all jobs that have entered
    static LinkedList<Job> table;

    /**
     * CONSTRUCTOR***************************************************************
     */
    JobTable ()
    {
        table = new LinkedList<Job>();
    }

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

    public static void add (int[] p)
    {
        Job newJob = new Job(p);
        table.add(newJob);
    }

    public static void clearAddress (int jobID)
    {
        if (!table.get(jobID-1).inMemory)
        {
            table.get(jobID - 1).address = -1;
        }
    }

    public static void clearIO (int jobID)
    {
        table.get(jobID - 1).pendingIO = 0;
    }

    public static int decrementIO (int jobID)
    {
        Job decJob = table.get(jobID - 1);
        decJob.pendingIO--;
        // System.out.println("-JobTable decrements I/O");
        // System.out.println("--Job# " + decJob.idNum + " has " + decJob.pendingIO + " i/o
requests");
        return decJob.pendingIO;
    }

    public static boolean doingIO(int jobID)
    {
        if (jobID != -1) {
            return table.get(jobID - 1).latched;
        }
        else {
```

```
        return false;
    }
}

public static int getAddress (int jobID)
{
    return table.get(jobID - 1).address;
}

public static int getCurrentCPUTime (int jobID)
{
    return table.get(jobID - 1).currentCPUTime;
}

public static int getDirection (int jobID)
{
    return table.get(jobID - 1).direction;
}

public static int getIO (int jobID)
{
    return table.get(jobID - 1).pendingIO;
}

public static int getMaxCPUTime (int jobID)
{
    return table.get(jobID - 1).maxCPUTime;
}

public static int getPriorityTime (int jobID)
{
    return table.get(jobID -1).priorityTime;
}

public static int getSize (int jobID)
{
    return table.get(jobID -1).size;
}

public static boolean getSwapped (int jobID)
{
    return table.get(jobID - 1).swapped;
}

public static int getTimeLeft (int jobID)
{
    if (jobID != -1) {
        return (table.get(jobID - 1).maxCPUTime -
            table.get(jobID - 1).currentCPUTime);
    }
    else {
        return -1;
    }
}

public static int incrementIO(int jobID)
{
```

```
        Job incJob = table.get(jobID - 1);
        incJob.pendingIO++;
        // System.out.println("-JobTable increments I/O");
        // System.out.println("--Job# " + incJob.idNum + " has " + incJob.pendingIO + " i/o
requests");
        return incJob.pendingIO;
    }

    public static void incrementTime (int jobID, int time)
    {
        table.get(jobID - 1).currentCPUTime =
        table.get(jobID - 1).currentCPUTime + time;
    }

    public static void inMemory(int jobID)
    {
        table.get(jobID -1).inMemory = true;
    }

    public static boolean isBlocked(int jobID)
    {
        if (jobID != -1) {
            return table.get(jobID - 1).blocked;
        }
        else {
            return false;
        }
    }

    public static boolean isReady(int jobID)
    {
        if (jobID != -1) {
            return table.get(jobID - 1).ready;
        }
        else {
            return false;
        }
    }

    public static boolean isSwapping (int jobID)
    {
        return table.get(jobID-1).inDrum;
    }

    public static boolean isTerminated (int jobID)
    {
        return table.get(jobID-1).terminated;
    }

    public static void outMemory(int jobID)
    {
        table.get(jobID -1).inMemory = false;
    }

    public void print ()
    {
        // System.out.println("-JobTable Report");
```

```
    // System.out.print("--Jobs ");
    // for (int i = 0; i < table.size(); i++)
    // {
    //  String t = "";
    //  String b = "";
    //  String r = "";
    //  String io = ("(" + table.get(i).pendingIO + ")");
    //  if (table.get(i).terminated)
    //  {
    //      t = "T";
    //  }
    //  if (table.get(i).blocked)
    //  {
    //      b = "B";
    //  }
    //  if (table.get(i).ready)
    //  {
    //      r = "R";
    //  }
    //  System.out.print((table.get(i).idNum) +
    //      ":" + t + b + r + io + ", ");
    // }
    // System.out.println("");
}

public static void resetPriorityTime (int jobID)
{
    table.get(jobID -1).priorityTime = os.currentTime;
}

public static Job returnJob (int jobID)
{
    if (table.get(jobID - 1) != null) {
        return table.get(jobID - 1);
    }
    else {
        return null;
    }
}

public static void setAddress (int jobID, int address)
{
    table.get(jobID - 1).address = address;
}

public static void setBlocked (int jobID)
{
    if (jobID != -1) {
        // System.out.println("-JobTable sets " + jobID + " to blocked");
        table.get(jobID - 1).blocked = true;
    }
}

public static void setDirection (int jobID, int direction)
{
    table.get(jobID - 1).direction = direction;
    // System.out.println("-JobTable sets swap direction");
```

```
}

public static void setDoingIO (int jobID)
{
    if (jobID != -1) {
        // System.out.println("-JobTable sets " + jobID + " to latched");
        table.get(jobID - 1).latched = true;
    }
}

public static void setReady (int jobID)
{
    if (jobID != -1) {
        // System.out.println("-JobTable sets " + jobID + " as ready");
        table.get(jobID - 1).ready = true;
    }
}

public static void setSwapped (int jobID)
{
    table.get(jobID - 1).swapped = true;
}

public static void setSwapping (int jobID)
{
    table.get(jobID - 1).inDrum = true;
}

public static void stopSwapping (int jobID)
{
    table.get(jobID-1).inDrum = false;
}

public static void terminate(int jobID)
{
    table.get(jobID-1).terminated = true;
}

public static void unsetBlocked (int jobID)
{
    if (jobID != -1) {
        // System.out.println("-JobTable sets " + jobID + " to unblocked");
        table.get(jobID - 1).blocked = false;
    }
}

public static void unsetDoingIO (int jobID)
{
    if (jobID != -1) {
        // System.out.println("-JobTable sets " + jobID + " to unlatched");
        table.get(jobID - 1).latched = false;
    }
}

public static void unsetReady (int jobID)
{
    if (jobID != -1) {
```

```
            // System.out.println("-JobTable sets " + jobID + " to unready");
            table.get(jobID - 1).ready = false;
        }
    }
}
```

```java
import java.util.LinkedList;

public class MemoryManager
{
    // For the freeSpaceTable
    class FreeSpace
    {
        int start;
        int size;

        FreeSpace (int _start, int _size)
        {
            start = _start;
            size = _size;
        }
    }

    /**
     * VARIABLES*****************************************************************
     */
    final int MEMSIZE = 100;
    LinkedList<FreeSpace> freeSpaceTable;
    LinkedList<Integer> jobsInMemory;
    LinkedList<Integer> unswappedQueue;
    LinkedList<Integer> swappedQueue;
    LinkedList<Integer> blockedQueue;
    LinkedList<Integer> terminated;

    /**
     * CONSTRUCTOR***************************************************************
     */
    MemoryManager ()
    {
        // Initial freeSpace is all of memory
        FreeSpace empty = new FreeSpace (0, MEMSIZE);
        freeSpaceTable = new LinkedList<FreeSpace>();
        freeSpaceTable.add(empty);
        // Keeps track of jobs in memory and their base addresses
        jobsInMemory = new LinkedList<Integer>();
        unswappedQueue = new LinkedList<Integer>();
        swappedQueue = new LinkedList<Integer>();
        blockedQueue = new LinkedList<Integer>();
        terminated = new LinkedList<Integer>();
    }

    /**
     * PRIVATE METHODS***********************************************************
     */

    /**
     * Adds new job to the correct queue
     * @param jobID [description]
     */
    void addToQueues (int jobID)
    {
        if (JobTable.getAddress(jobID) == -1) {
            blockedQueue.remove((Integer)jobID);
```

```
            unswappedQueue.remove((Integer)jobID);
            swappedQueue.remove((Integer)jobID);
            if (JobTable.isBlocked(jobID)) {
                blockedQueue.add(jobID);
            }
            else if (JobTable.getSwapped(jobID)) {
                swappedQueue.add(jobID);
            }
            else {
                unswappedQueue.add(jobID);
            }
        }
    }

    /**
     * Finds a job from queues to add to memory
     * @return jobID of job to add
     */
    int find ()
    {
        int swapInJob = -1;
        // Send another job
        // System.out.print("-MemoryManager attempts to add another job to memory");

        // See if we can find free space
        swapInJob = findFreeSpace();
        // If freespace is found
        if (swapInJob != -1) {
            // Update address in jobTable
            // System.out.println("--" + swapInJob + " added and sent to swapper");
        }
        else {
            // System.out.println("--No Space found");
        }

        return swapInJob;
    }

    /**
     * Checks for availability in freeSpaceTable
     * If found, adds job to jobsInMemory, updates freeSpaceTable
     * Else, add jobs to memQueue
     * @param job to be added
     * @return  jobID of job which space was found
     */
    int findFreeSpace ()
    {
        // set initial address/job to invalid
        int jobID = -1;
        // Checks unswapped first
        if (!unswappedQueue.isEmpty()) {
            jobID = iterateFreeSpace(unswappedQueue);
        }
        // Then swapped, but only if unswapped is empty
        else if (!swappedQueue.isEmpty()) {
            jobID = iterateFreeSpace(swappedQueue);
        }
```

```
        // Then blocked, but only if both unswapped & swapped are empty
        else if (!blockedQueue.isEmpty()) {
            jobID = iterateFreeSpace(blockedQueue);
        }
        return jobID;
    }

    /**
     * Given a list of freespaces, checks if there is freespace for any
     * pending jobs
     * @param  queue LinkedList of freespace
     * @return       the jobID of job which fits
     */
    int iterateFreeSpace (LinkedList<Integer> queue)
    {
        int jobID = -1;
        int address = -1;
        for (int j = 0; j < queue.size(); j++) {
            // Gets next element of queue
            int jobSize = JobTable.getSize(queue.get(j));
            // System.out.println("--Checking for " + jobSize
            //  + " free space...");

            // Checks freeSpaceTable for first available memory location
            FreeSpace iterator;
            for (int i = 0; i < freeSpaceTable.size(); i++) {
                iterator = freeSpaceTable.get(i);
                // System.out.println("---FreeSpace : Address=" +
                //  iterator.start + " Size=" + iterator.size);

                // If the space will hold new job
                if (iterator.size >= jobSize) {
                    if (jobSize > 40) {
                        if (iterator.start + jobSize > 60 &&
                            iterator.start < 50) {
                            break;
                        }
                    }
                    address = iterator.start;
                    iterator.start = iterator.start + jobSize;
                    iterator.size = iterator.size - jobSize;
                    // System.out.println("----Fit success");

                    FreeSpace newSpace =
                        new FreeSpace(iterator.start, iterator.size);

                    freeSpaceTable.remove(i);
                    if (iterator.size != 0) {
                        freeSpaceTable.add(i, newSpace);
                        // System.out.println("----New : Address=" + newSpace.start + " Size=" +
newSpace.size);
                    }
                    else {
                        // System.out.println("----Used entire space");
                    }
                    jobID = queue.get(j);
                    JobTable.setAddress(jobID, address);
```

```
                queue.remove((Integer)jobID);
                break;
            }
            // If the space is too small
            else {
                // System.out.println("----Too small");
            }
        }
        // Checks whether freespace was found for job and ends iterative check
        if (address != -1) {
            break;
        }
    }
    return jobID;
}

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

/**
 * adds new job to memory from id number
 * checks to see if space is available
 * Sets address in job table
 * returns true if space is found and swapper should run
 * false if otherwise
 * @param  idNum job to add into memory
 * @return       jobID of new job to swap into memory
 */
public int add (int jobID)
{
    // If the job is not in memory or in the queue already & is valid
    if (jobID != -1 && !jobsInMemory.contains((Integer)jobID)) {
        // Sets swap direction to-Memory
        JobTable.setDirection(jobID, 0);
        // System.out.println ("-MemoryManager adds job " +
        //  jobID + " to queue");
        addToQueues(jobID);

        // With new element in memQueue, attemp  ts to find space
        jobID = findFreeSpace();
        return jobID;
    }
    else {
        return -1;
    }
}

/**
 * Adds job to jobsInMemory
 * @param jobID [description]
 */
public void addToMemory (int jobID)
{
    if (jobID != -1) {
        //memQueue.remove((Integer)jobID);
        jobsInMemory.add(jobID);
```

```
        }
    }

    /**
     * Allocates jobs memory to freespace table, will append current
     * freespace if they are contiguous
     * Removes job from jobsInMemory
     * @param jobID jobID of job to be swapped
     */
    public void free (int jobID)
    {
        if (jobID != -1 && !JobTable.doingIO(jobID)) {
            // System.out.println("-MemoryManager begins to free job " + jobID + " with " +
JobTable.getTimeLeft(jobID) + " left");

            // Variables needed in iteration
            // Iterates across freespaces in mem
            FreeSpace iterator;
            // Used when checking to append between 2 freespaces
            FreeSpace iterator2;
            // Will replace current freespace if appended, or be added if no
            // appending occurs
            FreeSpace newSpace;
            Job job = JobTable.returnJob(jobID);

            // If the just freed job still has CPU time remaining,
            // then add it back into the memqueue
            if (JobTable.getTimeLeft(jobID) > 0 &&
                !JobTable.isTerminated(jobID) &&
                    !JobTable.isBlocked(jobID)) {
                JobTable.setDirection(jobID, 0);
                // System.out.println("ADDEDTOQUEUS");
                addToQueues(jobID);
            }

            // Free the space
            // First check to see if current freespace can be appended to
            boolean append = false;
            for (int i = 0; i < freeSpaceTable.size(); i++) {
                iterator = freeSpaceTable.get(i);
                // Status print
                // System.out.println("--Job to Free start=" + job.address +
                //  " end=" + (job.size + job.address - 1));
                // System.out.println("--Iterator start=" + iterator.start +
                //  " end=" + (iterator.size+iterator.start - 1));

                // Check to see if perfect fit between freespaces
                // If freespace does not start at zero && there is
                // another freespace
                if ((i + 1) < freeSpaceTable.size()) {
                    iterator2 = freeSpaceTable.get(i+1);
                    // If the boundaries match on both sides
                    if (job.address == (iterator.start + iterator.size) &&
                       (job.address + job.size) == iterator2.start) {
                        // Details to print
                        // System.out.println("--Freespace appended btw");
                        // System.out.println("--Existing1: Address=" +
```

```
            //  iterator.start + " Size=" + iterator.size);
            // System.out.println("--Addition : Address=" +
            //  job.address + " Size=" + job.size);
            // System.out.println("--Existing2: Address=" +
            //  iterator2.start + " Size=" + iterator2.size);
            // Updates iterator to new size
            iterator.size =
                iterator.size + job.size + iterator2.size;
            // System.out.println("--New      : Address=" +
            //  iterator.start + " Size=" + iterator.size);
            newSpace =
                new FreeSpace(iterator.start, iterator.size);

            // Replaces 2 iterators with newSpace
            freeSpaceTable.remove(i);
            freeSpaceTable.remove(i);
            freeSpaceTable.add(i, newSpace);

            append = true;
            break;
        }
    }
    // If freedspace ends at existing freespace
    if (job.address == (iterator.start + iterator.size)) {
        // Details to print
        // System.out.println("--Freespace appended to end");
        // System.out.println("--Addition : Address=" +
        //  job.address + " Size=" + job.size);
        // System.out.println("--Existing : Address=" +
        //  iterator.start + " Size=" + iterator.size);
        // Updates iterator to new size
        iterator.size = iterator.size + job.size;
        // System.out.println("--New      : Address=" +
        //  iterator.start + " Size=" + iterator.size);
        newSpace =
            new FreeSpace(iterator.start, iterator.size);
        // Replaces iterator with newSpace
        freeSpaceTable.remove(i);
        freeSpaceTable.add(i, newSpace);

        append = true;
        break;
    }
    // If freedspace starts at existing freespace
    if ((job.address + job.size) == iterator.start) {
        // Details to print
        // System.out.println("--Freespace appended to start");
        // System.out.println("--Existing : Address=" +
        //  iterator.start + " Size=" + iterator.size);
        // System.out.println("--Addition : Address=" +
        //  job.address + " Size=" + job.size);
        // Updates iterator to new size
        iterator.start = job.address;
        iterator.size = iterator.size + job.size;
        // System.out.println("--New      : Address=" +
        //  iterator.start + " Size=" + iterator.size);
        newSpace = iterator;
```

```
                // Replaces iterator with newSpace
                freeSpaceTable.remove(i);
                freeSpaceTable.add(i, newSpace);

                append = true;
                break;
            }
        }
        // If the space couldn't be appended, need to add into correct location
        if (!append) {
            // System.out.println("--New freespace added : " +
            //  job.address + ", " + job.size);
            newSpace = new FreeSpace(job.address, job.size);
            for (int i = 0; i < freeSpaceTable.size(); i++) {
                iterator = freeSpaceTable.get(i);
                if (i == 0 && newSpace.start < iterator.start) {
                    // System.out.println("---At "+ i);
                    freeSpaceTable.add(i, newSpace);
                    break;
                }
                if ( (i+1) < freeSpaceTable.size()) {
                    iterator2 = freeSpaceTable.get(i+1);
                    if (iterator.start < newSpace.start
                        && newSpace.start < iterator2.start) {
                        // System.out.println("---At "+ (i+1));
                        freeSpaceTable.add(i+1, newSpace);
                        break;
                    }
                }
                if ( i == freeSpaceTable.size()-1) {
                    // System.out.println("---At end");
                    freeSpaceTable.add(newSpace);
                    break;
                }
            }
            if (freeSpaceTable.isEmpty()) {
                freeSpaceTable.add(newSpace);
            }
        }
        // Removes freed job from memory & clears address in jobTable
        jobsInMemory.remove((Integer) jobID);
        JobTable.clearAddress(jobID);
    }
}

/**
 * Frees memory of terminated jobs if all I/O done
 */
public void freeTerminated()
{
    for (int i = 0; i < terminated.size(); i++) {
        if (JobTable.getIO(terminated.get(i)) == 0) {
            free(terminated.remove(i));
        }
    }
}
```

```
/**
 * Adds to list of jobs terminated with memory needing to be freed
 * @param jobID [description]
 */
public void newTerminated(int jobID)
{
    if (jobID != -1) {
        if (JobTable.getIO(jobID) > 0 || JobTable.doingIO(jobID)) {
            terminated.add(jobID);
        }
        else {
            free(jobID);
        }
    }
}

/**
 * Prints the status of freeSpace and the jobs in memory
 */
public void print()
{
    // // System.out.println("-Memory Report");
    // FreeSpace iterator;
    // for (int i = 0; i < freeSpaceTable.size(); i++)
    // {
    //  iterator = freeSpaceTable.get(i);
    //  System.out.println("--FreeSpace " + i + " : Address=" +
    //      iterator.start + " Size=" + iterator.size);
    // }
    // System.out.println("--Jobs waiting for memory: ");
    // System.out.print("---Unswapped : ");
    // for (int i = 0; i < unswappedQueue.size(); i++)
    // {
    //  System.out.print(unswappedQueue.get(i) + ", ");
    // }
    // System.out.println("");
    // System.out.print("---Swapped : ");
    // for (int i = 0; i < swappedQueue.size(); i++)
    // {
    //  System.out.print(swappedQueue.get(i) + ", ");
    // }
    // System.out.println("");
    // System.out.print("---Blocked : ");
    // for (int i = 0; i < blockedQueue.size(); i++)
    // {
    //  System.out.print(blockedQueue.get(i) + ", ");
    // }
    // System.out.println("");
    // System.out.print("--Jobs in memory: ");
    // for (int i = 0; i < jobsInMemory.size(); i++)
    // {
    //  System.out.print(jobsInMemory.get(i) + ", ");
    // }
    // System.out.println("");
}

/**
```

```
     * Returns positive if there are not enough running jobs in memory
     * @return boolean whether it is wise to swap a job out
     */
    public boolean smartSwap ()
    {
        int blockedCount = 0;
        for (int i = 0; i < jobsInMemory.size(); i++) {
            if (JobTable.isBlocked(jobsInMemory.get(i))) {
                blockedCount++;
            }
        }
        if (blockedCount > 0) {
            return true;
        }
        else {
            return false;
        }
    }
}
```

```
import java.util.LinkedList;

public class Swapper
{

    final int IN = 0;
    final int OUT = 1;

    /**
     * VARIABLES*****************************************************************
     */
    LinkedList<Integer> blockedOutQueue;    // To be swapped out
    LinkedList<Integer> blockedInQueue;     // To be swapped in
    LinkedList<Integer> defaultOutQueue;    // To be swapped out
    LinkedList<Integer> defaultInQueue;     // To be swapped in
    int inDrum;

    /**
     * CONSTRUCTOR***************************************************************
     */
    Swapper ()
    {
        blockedOutQueue = new LinkedList<Integer>();
        blockedInQueue = new LinkedList<Integer>();
        defaultOutQueue = new LinkedList<Integer>();
        defaultInQueue = new LinkedList<Integer>();
        inDrum = -1;
    }

    /**
     * PRIVATE METHODS***********************************************************
     */

    /**
     * Adds new swap job to appropriate queue
     * @param jobID the job to be added
     */
    void add (int jobID)
    {
        if (jobID != inDrum) {
            remove(jobID);
            // In Queues
            if (JobTable.getDirection(jobID) == 0) {
                // Blocked
                if (JobTable.isBlocked(jobID)) {
                    blockedInQueue.add(jobID);
                }
                // Default
                else {
                    defaultInQueue.add(jobID);
                }
            }
            else {
                // Blocked
                if (JobTable.isBlocked(jobID)) {
                    blockedInQueue.remove((Integer)jobID);
                    blockedOutQueue.add(jobID);
```

```
            }
            // Default
            else {
                defaultOutQueue.add(jobID);
            }
        }
    }
}

/**
 * Removes any or all instances of job in queues
 * @param jobID job to be removed
 */
void remove (int jobID)
{
    blockedInQueue.remove((Integer)jobID);
    blockedOutQueue.remove((Integer)jobID);
    defaultInQueue.remove((Integer)jobID);
    defaultOutQueue.remove((Integer)jobID);
}

/**
 * PUBLIC METHODS************************************************************
 */
/**
 * Handles the actual searching and swap calling
 */
public int swap ()
{
    // System.out.println("-Swap Queues:");
    // System.out.println("--BlockedOutQueue has " + blockedOutQueue.size());
    // System.out.println("--BlockedInQueue has " + blockedInQueue.size());
    // System.out.println("--DefaultInQueue has " + defaultInQueue.size());
    // System.out.println("--DefaultOut Queue has " + defaultOutQueue.size());
    if (inDrum == -1) {
        if (!defaultInQueue.isEmpty()) {
            inDrum = defaultInQueue.remove();
        }
        else if (!blockedOutQueue.isEmpty()) {

            inDrum = blockedOutQueue.remove();
        }
        else if (!blockedInQueue.isEmpty()) {
            inDrum = blockedInQueue.remove();
        }
        else if (!defaultOutQueue.isEmpty()) {
            inDrum = defaultOutQueue.remove();
        }
        if (inDrum != -1) {
            if (JobTable.doingIO(inDrum)) {
                add(inDrum);
                inDrum = -1;
            }
            else {
                JobTable.setSwapping(inDrum);
                Job swapJob = JobTable.returnJob(inDrum);
                sos.siodrum (swapJob.idNum, swapJob.size,
```

```
                    swapJob.address, swapJob.direction);
                String descriptor = "";
                if (swapJob.direction == 0) {
                    descriptor = " to ";
                }
                else if (swapJob.direction == 1) {
                    descriptor = " from ";
                }
                // System.out.println("--Begin swapping Job " +
                //  swapJob.idNum +" with size " + swapJob.size +
                //  descriptor + swapJob.address);
            }
        }
    }
    return inDrum;
}

/**
 * Prints status of Drum and Drum Queue
 */
public void print ()
{
    // System.out.println("-Swap Report:");
    // System.out.println("--In Drum : " + inDrum);
    // System.out.print("--In Queue: ");
    // System.out.print("DefaultIn:");
    // for (int i = 0; i < defaultInQueue.size(); i++)
    // {
    //  System.out.print(defaultInQueue.get(i) + ", ");
    // }
    // System.out.print("BlockedOut:");
    // for (int i = 0; i < blockedOutQueue.size(); i++)
    // {
    //  System.out.print(blockedOutQueue.get(i) + ", ");
    // }
    // System.out.print("BlockedIn:");
    // for (int i = 0; i < blockedInQueue.size(); i++)
    // {
    //  System.out.print(blockedInQueue.get(i) + ", ");
    // }
    // System.out.print("DefaultOut:");
    // for (int i = 0; i < defaultOutQueue.size(); i++)
    // {
    //  System.out.print(defaultOutQueue.get(i) + ", ");
    // }
    // System.out.println("");
}

/**
 * Handles drmint
 * @return idNum of job drum-to-memory, -1 if memory-to-drum
 */
public int swapDone ()
{
    // System.out.println("-Swapper getting swap details");
    int jobID = inDrum;
    inDrum = -1;
```

```
        JobTable.stopSwapping(jobID);
        JobTable.resetPriorityTime(jobID);
        Job job = JobTable.returnJob(jobID);

        // Status
        if (job.direction == 1) {
            // System.out.println("--Memory-to-Drum done for job " + jobID);
            JobTable.setSwapped(jobID);
            JobTable.outMemory(jobID);
        }
        else {
            // System.out.println("--Drum-to-Memory done for job " + jobID);
            JobTable.inMemory(jobID);
        }
        return jobID;
    }

    /**
     * [swapIn description]
     * @param jobID [description]
     */
    public void swapIn (int jobID)
    {
        if (jobID != -1) {
            // System.out.println("-Swapper beginning swap in of Job " + jobID);
            // System.out.println("--Added Job " + jobID + " to swap queue");
            JobTable.setDirection(jobID, 0);
            add(jobID);
        }
    }

    /**
     * [swapOut description]
     * @param jobID [description]
     */
    public void swapOut (int jobID)
    {
        if(jobID != -1) {
            // System.out.println("-Swapper beginning swap out of Job " +
            //  jobID);
            // System.out.println("--Added Job " + jobID + " to swap queue");
            JobTable.setDirection(jobID, 1);
            add(jobID);
        }
    }
}
```

```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;
        }
    }
}
```

```java
import java.util.LinkedList;

public class IOScheduler
{
    /**
     * VARIABLES*****************************************************************
     */
    LinkedList<Integer> defaultQueue;    // I/O for non-blocked jobs
    LinkedList<Integer> blockedInQueue; // I/O for blocked, in-memory jobs
    LinkedList<Integer> blockedOutQueue;// I/O for blocked, not in-memory jobs
    LinkedList<Integer> terminatedQueue;// I/O for terminated jobs
    LinkedList<Integer> reportedToSwap;
    int inIO;                        // current process in I/O

    /**
     * CONSTRUCTOR***************************************************************
     */
    IOScheduler ()
    {
        defaultQueue = new LinkedList<Integer>();
        blockedInQueue = new LinkedList<Integer>();
        blockedOutQueue = new LinkedList<Integer>();
        terminatedQueue = new LinkedList<Integer>();
        reportedToSwap = new LinkedList<Integer>();
        inIO = -1;
    }

    /**
     * PRIVATE METHODS***********************************************************
     */
    /**
     * Gets next available I/O task
     */
    void ioCheck ()
    {
        // System.out.println("-I/O Queues:");
        // System.out.println("--BlockedIn Queue has " + blockedInQueue.size());
        // System.out.println("--Terminated Queue has " + terminatedQueue.size());
        // System.out.println("--Default Queue has " + defaultQueue.size());
        if (inIO == -1)
        {
            // Checks terminated first
            if (!terminatedQueue.isEmpty()) {
                inIO = terminatedQueue.remove();
            }
            // Then in-memory-blocked
            else if (!blockedInQueue.isEmpty()) {
                inIO = blockedInQueue.remove();
            }
            // Then regular jobs
            else if (!defaultQueue.isEmpty()) {
                inIO = defaultQueue.remove();
            }
            // If an I/O task was found, run it
            if (inIO != -1) {
                // System.out.println("--Job is sent to do I/O");
                JobTable.setDoingIO(inIO);
```

```
            sos.siodisk(inIO);
        }
    }
}

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

/**
 * Accepts new job for I/O
 * Adds new job to default I/O queue
 * @param jobID of the job to add
 */
public void add (int jobID)
{
    // System.out.println("-IOScheduler accepts new job");
    JobTable.incrementIO(jobID);
    defaultQueue.add(jobID);
    ioCheck();
}

/**
 * Checks to see if given job is doing I/O
 * @param  jobID job to query
 * @return       boolean true if job in I/O, false otherwise
 */
public boolean doingIO (int jobID)
{
    if (jobID == inIO) {
        // System.out.println("--Job " + jobID + " is doing I/O");
        return true;
    }
    else {
        // System.out.println("--Job " + jobID + " is not doing I/O");
        return false;
    }
}

/**
 * When I/O has finished, removes current job from inIO
 * Tries to start next job if available
 * Updates inIO
 * @return jobID of process that just finished I/O and jobID of
 *               process that needs to be brought into memory
 */
public int ioDone ()
{
    // If job which finished I/O is valid
    int jobID = inIO;
    inIO = -1;
    if (jobID != -1) {
        // Decrement & get it's I/O pending
        JobTable.decrementIO(jobID);
        JobTable.unsetDoingIO(jobID);
        // If the job is ready to be unblocked
        if (JobTable.getIO(jobID) == 0 && JobTable.isBlocked(jobID)) {
```

```
            JobTable.unsetBlocked(jobID);
        }
    }

    ioCheck();
    return jobID;
}

/**
 * Prints status of I/O
 */
public void print () {
    // System.out.println("-I/O Report:");
    // System.out.println("--In I/O: " + inIO);
    // System.out.println("--Next In Queue: ");
    // System.out.print("---Terminated: ");
    // for (int i = 0; i < terminatedQueue.size(); i++)
    // {
    //  System.out.print(terminatedQueue.get(i) + ", ");
    // }
    // System.out.print("---blockedInQueue: ");
    // for (int i = 0; i < blockedInQueue.size(); i++)
    // {
    //  System.out.print(blockedInQueue.get(i) + ", ");
    // }
    // System.out.print("---blockedOutQueue: ");
    // for (int i = 0; i < blockedOutQueue.size(); i++)
    // {
    //  System.out.print(blockedOutQueue.get(i) + ", ");
    // }
    // System.out.print("---DefaultQueue: ");
    // for (int i = 0; i < defaultQueue.size(); i++)
    // {
    //  System.out.print(defaultQueue.get(i) + ", ");
    // }
}

/**
 * Checks blocked in-memory for jobs to swap out
 * @return jobID of job to swap out
 */
public int readyToLeave ()
{
    // Job that is ready to leave, initialized to invalid
    int swapOutBlockedJob = -1;

    // If blocked jobs are in memory
    if (!blockedInQueue.isEmpty()) {
        // Checks if last element in queue is already swapping,
        // if not, then set to swap in
        if (blockedInQueue.getLast() != inIO) {
            swapOutBlockedJob = blockedInQueue.getLast();
        }
        // Checks if a blocked job has exceeded its in memory time,
        // if it has, then set to swap out
        else if (JobTable.getPriorityTime(blockedInQueue.getFirst())
            > 1000) {
```

```
            swapOutBlockedJob = blockedInQueue.getFirst();
        }
        // Checks to if job selected hasn't already requested to
        // be swapped. If it has, then invalidate job to swap out.
        // If it hasn't then add to reported to swap list
        if (reportedToSwap.contains(swapOutBlockedJob)) {
            swapOutBlockedJob = -1;
        }
        else {
            reportedToSwap.add(swapOutBlockedJob);
        }
    }
    return swapOutBlockedJob;
}

/**
 * Checks blocked not in-memory for jobs to swap in if there is no
 * blocked job in memory
 * @return jobID of job to swap in
 */
public int readyToReturn ()
{
    int swapInBlockedJob = -1;
    if (blockedInQueue.isEmpty() && !blockedOutQueue.isEmpty()) {
        swapInBlockedJob = blockedOutQueue.getFirst();
        if (reportedToSwap.contains(swapInBlockedJob)) {
            swapInBlockedJob = -1;
        }
        else {
            reportedToSwap.add(swapInBlockedJob);
        }
    }
    return swapInBlockedJob;
}



/**
 * Moves a jobs I/O to correct queue when status changes
 * @param jobID the job to move
 */
public void moveIO (int jobID)
{
    if (jobID != -1) {
        int ioCount = JobTable.getIO(jobID);
        if (ioCount > 0) {
            if (reportedToSwap.contains(jobID)) {
                reportedToSwap.remove((Integer)jobID);
            }
            // System.out.println("-IO");
            // Query the jobTable to determine which list I/O should move to
            // Move to terminated
            if (JobTable.isTerminated(jobID)) {
                // Remove from default, blockedIn
                // System.out.println("--Moving to terminatedQueue");
                while (defaultQueue.contains((Integer)jobID)) {
                    defaultQueue.remove((Integer)jobID);
```

```
                    terminatedQueue.add(jobID);
                }
                while (blockedInQueue.contains((Integer)jobID)) {
                    blockedInQueue.remove((Integer)jobID);
                    terminatedQueue.add(jobID);
                }
            }
            // Move to blockedOut
            else if (JobTable.isBlocked(jobID)) {
                // Move to blockedOut
                if (JobTable.getAddress(jobID) == -1 ||
                    (JobTable.getAddress(jobID) != -1 &&
                        JobTable.isSwapping(jobID))) {
                    // System.out.println("--Moving to blockedOutQueue");
                    // Remove from blockedIn
                    while (blockedInQueue.contains((Integer)jobID)) {
                        blockedInQueue.remove((Integer)jobID);
                        blockedOutQueue.add(jobID);
                    }
                    while (defaultQueue.contains((Integer)jobID)) {
                        defaultQueue.remove((Integer)jobID);
                        blockedOutQueue.add(jobID);
                    }
                }
                // Move to blockedIn
                else if (JobTable.getAddress(jobID) != -1 ||
                    (JobTable.getAddress(jobID) == -1
                        && JobTable.isSwapping(jobID))) {
                    // System.out.println("--Moving to blockedInQueue");
                    // Remove from default
                    while (defaultQueue.contains((Integer)jobID)) {
                        defaultQueue.remove((Integer)jobID);
                        blockedInQueue.add(jobID);
                    }
                    while (blockedOutQueue.contains((Integer)jobID)) {
                        blockedOutQueue.remove((Integer)jobID);
                        blockedInQueue.add(jobID);
                    }
                }
            }
        }
    }
}
```

OPERATING SYSTEM SIMULATION

```
*** Clock:  314, Job 1 terminated normally (terminate svc issued)
 Response Time:  314 CPU time:  21 # I/O operations completed:  3
 # I/O operations pending:  0


*** Clock:  4621, Job 2 terminated normally (terminate svc issued)
Dilation:  1.01 CPU time:  2000  # I/O operations completed:  2
 # I/O operations pending:  0


*** Clock:  6265, Job 3 terminated abnormally (max cpu time exceeded)
 Response Time:  365 CPU time:  20 # I/O operations completed:  3
 # I/O operations pending:  0


*** Clock:  7611, Job 7 terminated normally (terminate svc issued)
 Response Time:  252 CPU time:  90 # I/O operations completed:  0
 # I/O operations pending:  0


... continues


*** Clock:  876454, Job 444 terminated abnormally (max cpu time exceeded)
 Response Time:  198 CPU time:  18 # I/O operations completed:  0
 # I/O operations pending:  0


*** Clock:  876714, Job 446 terminated normally (terminate svc issued)
 Response Time:  418 CPU time:  60 # I/O operations completed:  0
 # I/O operations pending:  0


*** Clock:  882414, Job 445 terminated normally (terminate svc issued)
Dilation:  2.45 CPU time:  2500  # I/O operations completed:  2
 # I/O operations pending:  0


*** Clock:  882429, Job 449 terminated abnormally (max cpu time exceeded)
 Response Time:  524 CPU time:  15 # I/O operations completed:  0
 # I/O operations pending:  0


*** Clock:  882843, Job 447 terminated normally (terminate svc issued)
 Response Time:  6528 CPU time:  14 # I/O operations completed:  0
 # I/O operations pending:  0


*** Clock:  886497, Job 448 terminated normally (terminate svc issued)
Dilation:  2.44 CPU time:  3000  # I/O operations completed:  3
 # I/O operations pending:  0
```

```
*** Clock:  889136, Job 451 terminated normally (terminate svc issued)
 Response Time:  431 CPU time:  21 # I/O operations completed:  0
 # I/O operations pending:  0


*** Clock:  895156, Job 453 terminated abnormally (max cpu time exceeded)
 Response Time:  551 CPU time:  20 # I/O operations completed:  0
 # I/O operations pending:  0


*** Clock:  897756, Job 450 terminated normally (terminate svc issued)
Dilation:  2.96 CPU time:  4000  # I/O operations completed:  4
 # I/O operations pending:  0


*** Clock:  897756, Job 452 terminated normally (terminate svc issued)
Dilation:  3.23 CPU time:  2000  # I/O operations completed:  2
 # I/O operations pending:  0


*** Clock:  898167, Job 455 terminated normally (terminate svc issued)
 Response Time:  2132 CPU time:  11 # I/O operations completed:  0
 # I/O operations pending:  0


*** Clock:  898367, Job 396 terminated normally (terminate svc issued)
Dilation:  2.42 CPU time:  50000  # I/O operations completed:  50
 # I/O operations pending:  0


*** Clock:  899194, Job 458 terminated normally (terminate svc issued)
 Response Time:  280 CPU time:  9 # I/O operations completed:  0
 # I/O operations pending:  0



                        FINAL STATISTICS



 * * * SYSTEM STATUS AT 900000 * * *
 ==================================

 CPU:  job #457 running
 Disk:  idle
 Drum:  idle
Memory:  84 K words in use
Average dilation:  2.64
Average Response time:  8963.44


                CORE MAP

 Partition Job    Partition Job    Partition Job    Partition Job
```

0	456	25	456	50	457	75	0
1	456	26	456	51	457	76	0
2	456	27	456	52	457	77	0
3	456	28	456	53	457	78	0
4	456	29	456	54	457	79	0
5	456	30	456	55	457	80	0
6	456	31	456	56	457	81	0
7	456	32	456	57	457	82	0
8	456	33	456	58	457	83	0
9	456	34	456	59	457	84	0
10	456	35	456	60	457	85	0
11	456	36	456	61	457	86	0
12	456	37	456	62	457	87	454
13	456	38	456	63	457	88	454
14	456	39	456	64	457	89	454
15	456	40	456	65	457	90	454
16	456	41	456	66	457	91	454
17	456	42	456	67	457	92	454
18	456	43	456	68	457	93	454
19	456	44	456	69	457	94	454
20	456	45	456	70	457	95	454
21	456	46	456	71	457	96	454
22	456	47	457	72	457	97	0
23	456	48	457	73	457	98	0
24	456	49	457	74	0	99	0

JOBTABLE

Job#	Size	Time Arrived	CPUTime Used	MaxCPU Time	I/O's Pending	Priority	Blocked	Latched	InCore	Term
24	14	46798	12	18	1	1	yes	no	no	no
12	19	22349	138	550	1	1	yes	no	no	no
421	18	829458	12	23	1	2	yes	no	no	no
60	15	115694	24	14000	1	1	yes	no	no	no
84	14	165292	6	18	1	1	yes	no	no	no
25	17	46828	162	5300	1	2	yes	no	no	no
109	7	210439	6	21	1	2	yes	no	no	no
28	5	49707	42	7100	1	1	yes	no	no	no
288	8	563181	6	15	1	1	yes	no	no	no
34	10	66547	162	3500	1	1	yes	no	no	no
45	15	86326	228	40000	1	2	yes	no	no	no
236	21	461567	48	62	1	1	yes	no	no	no
37	27	66606	72	100	1	1	yes	no	no	no
87	23	165351	6	17	1	2	yes	no	no	no
47	10	86355	258	1300	1	2	yes	no	no	no
66	47	125834	318	65000	1	5	yes	no	no	no
67	27	125853	42	100	1	1	yes	no	no	no
69	14	131443	36	1500	1	2	yes	no	no	no
454	10	896005	1600	3500	0	1	no	no	yes	no
107	10	204849	72	1300	1	2	yes	no	no	no
213	8	420629	18	20	1	2	yes	no	no	no
137	10	264096	12	1300	1	2	yes	no	no	no
239	8	467176	12	15	1	2	yes	no	no	no

167	10	323343	126	1300	1	2	yes	no	no	no
216	47	422069	48	65000	1	5	yes	no	no	no
217	27	422088	30	100	1	1	yes	no	no	no
306	47	599810	138	65000	1	5	yes	no	no	no
253	23	499625	12	1400	1	2	yes	no	no	no
417	23	817068	12	17	1	2	yes	no	no	no
255	15	501055	12	40000	1	2	yes	no	no	no
367	27	718323	30	100	1	1	yes	no	no	no
257	10	501084	12	1300	1	2	yes	no	no	no
259	7	506674	12	21	1	2	yes	no	no	no
287	10	560331	72	1300	1	2	yes	no	no	no
307	27	599829	24	100	1	1	yes	no	no	no
377	10	738072	42	1300	1	2	yes	no	no	no
397	27	777570	54	100	1	1	yes	no	no	no
426	47	836798	72	65000	1	5	yes	no	no	no
437	10	856566	42	1300	1	2	yes	no	no	no
456	47	896045	800	65000	0	5	no	no	yes	no
457	27	896064	24	100	0	1	no	no	yes	no

Total jobs: 458 terminated: 417
% utilization CPU: 96.16 disk: 15.92 drum: 15.96 memory: 62.05