```
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();
    }
}
```