

*Pace Operating Systems Simulator*

Richard Nemes, Lixin Tao, Cathy Zura

School of Computer Science and Information Systems  
Pace University

***Richard Nemes*** is an Associate Professor of Computer Science at Pace University.

***Lixin Tao*** is a Professor of Computer Science at Pace University.

***Cathy Zura*** is a graduate of Pace Computer Science Master Program.

# *Pace Operating Systems Simulator*

Richard Nemes, Lixin Tao, Cathy Zura

School of Computer Science and Information Systems  
Pace University  
New York

October 2002

---

## **Table of Contents**

1	Introduction.....	2
2	Pace OS Simulator Structure .....	3
2.1	The Simulated Machine .....	3
2.2	The Simulator Driver .....	5
2.3	Student Operating Systems .....	7
2.4	User Jobs (Processes).....	8
2.4.1	Characteristics of a Job .....	8
2.4.2	Life Cycle of a Typical Job.....	9
2.5	Student OS Interface .....	9
2.5.1	Student OS Initialization.....	10
2.5.2	Interrupt Calls of the Simulator Driver to a Student OS.....	10
2.5.3	Responses of a Student OS to Interrupt Calls.....	11
2.5.4	System Time and CPU Time for a Job .....	11
2.6	Performance Measurements.....	11
2.7	Project Constraints .....	12
3	Outline of a Student OS Class .....	13
4	Constants and Methods Inherited by a Student OS .....	16
4.1	Inherited Constants .....	16
4.2	Inherited Methods .....	16
5	Source Code for a Simplified Demonstration OS: DemoOS.java .....	19
6	How to Compile and Run Student OS with Sun JDK .....	25
7	Pace Operating Systems Simulation Debugger .....	27
8	Potential Student Projects Based on the Pace OS Simulator .....	28
9	Appendix: Pace OS Simulator Error Messages .....	28

---

Dr. Nemes designed and implemented the original C version of this simulator. Ms. Cathy Zura translated it into its first Java version. Dr. Tao revised, upgraded, and documented the current Java version.

## 1 Introduction

Operating systems is one of the core courses of any undergraduate computer science curriculum. The course covers both fundamental concepts in computing and practical issues inherent in the design and implementation of any operating systems, virtual machines, or large software systems. An OS course exclusively focuses on classical fundamental concepts can easily lose the attention of the students due to the lack of concrete application examples. On the other hand, over-emphasizing on implementation details could limit the coverage and depth of fundamental concepts. Good balance of theory and well-designed course projects is the key to the effectiveness of OS teaching.

For computer science programs adopting Java as the first programming language, there are additional blessings and challenges to the design of effective OS course projects. It is fairly easy to use Java programs to illustrate individual concepts and techniques like multithreading, thread synchronization, deadlocks, remote procedure calls (remote method invocation), networking, and security. But since most operating systems are implemented mainly in C/C++ for efficiency, there is no Java source code for complete example operating systems that can show the integration and interaction of the individual topics in action. Most existing OS course projects are based on OS simulators implemented in or depending on C/C++. For example, Nachos is a very popular OS simulator for OS education. But it is implemented in C/C++, and it performs instruction-level simulation of a particular MIPS architecture. Even with the recent port of Nachos to Java, student sample jobs to run on Nachos usually have to be written in C/C++ and cross-compiled for the MIPS architecture.

The objective of the *Pace OS Simulator* is to provide an effective vehicle to design course projects on the interaction and application of key OS concepts in a one-semester Java-based OS course. Efforts were made to balance the complexity of student projects and richness that such projects could be.

The Pace OS Simulator is written in Java, and it has three major components: a simulated hardware machine, a simulator driver, and a student OS class. The organization closely reflects the interrupt-driven nature of modern operating systems. To reduce the complexity of student projects, job (process) streams are simulated with randomly generated or predefined key job characteristics. Such simulated job streams could be repeated if necessary to facilitate the study of alternative OS design and implementation strategies. The simulator driver is responsible for generating such simulated job streams, issuing interrupts for a student OS to process, implementing job swapping and user-disk IO operations, and validating the scheduling of the student OS. The student OS class should manage a pool of jobs, manage user-disk IO requests from the jobs, manage the memory of the simulated machine, respond to interrupt requests of the simulator driver, swap in/out of jobs from/to the system-disk, and schedule the jobs for CPU execution.

The Pace OS simulator comes with a JAR file of all the classes for the simulated machine and the simulator driver, as well as a demonstration OS implementation. Students can compile and run their OS implementation with the basic Sun JDK tools. A graphic-user-interface (GUI) for the *Pace OS Simulator Debugger* allows students to trace system state before and after each step of interrupt processing. Rich performance feedbacks are provided as to the effectiveness of the design and implementation of a student OS.

Based on the Pace OS simulator, course projects could be designed to explore long-term and medium-term job scheduling, CPU scheduling, memory management, and disk IO scheduling. A well-prepared student should be able to complete such a project in around 20 hours.

In the following sections, we first illustrate the structure of the Pace OS simulator, and provide detailed description of its major components. An outline of the student OS class is then provided to allow student projects to have a smooth start. We summarize all information a student needs to know about the simulated machine and the simulator driver in Section 4. Section 5 provides the source code listing of a simplified demonstration OS class as a reference base for student projects. Sections 6 and 7 explain how to compile and run a student OS, and how to use the Pace OS simulator debugger GUI to facilitate the OS development. We conclude this document with a list of sample projects that can be implemented on the Pace OS simulator as well as a list of student OS errors that can be detected by the Pace OS simulator.

## 2 Pace OS Simulator Structure

The Pace OS simulator has three major components: a simulated hardware machine, a simulator driver, and a student OS class.

### 2.1 The Simulated Machine

The simulated machine includes the following components:

- **A CPU.**

Up to one thread can execute with the CPU a time. The built-in interrupt handling mechanism does not support *nested* interrupt processing: interrupts are disabled at the very beginning of any interrupt processing, and re-enabled upon finishing such processing.

- **100K words of main memory for user jobs. Its physical addresses range from 0 to 99 (in unit of K words).**

Jobs will be assigned memory in units of K words. Only jobs residing in the main memory could be scheduled for execution.

- **A system-disk for swapping jobs, connected to memory through a DMA controller.**

The system-disk is for the storage of job executables and their memory images. A new job will first *magically* arrive on the system-disk, and the student OS will be notified of its arrival by an interrupt. Later, when there is enough memory space for the new job, the student OS may choose to swap the job into the memory for execution. The student OS may later swap a partially executed job out back to the system-disk to optimize some performance measurements. The working of the system-disk is independent of the user-disk. At any time, only up to one job may conduct swap-in or swap-out. The duration of such job swap in/out operations is proportional to the job memory size.

A student OS can initiate a job swap with the system-disk by method

```
public void systemDiskJobSwap(int jobID, int memorySize,
                             int startAddress, int swapDirection);
```

When the job swap finishes, the simulator driver will issue a system disk interrupt to the student OS by calling its interrupt handler method

```
public int systemDiskInterrupt();
```

- **A user-disk for job data files, connected to memory through a DMA controller.**

The user-disk is for the storage of job data files. We do not differentiate file read and file write. The working of the user-disk is independent of the system-disk. At any time, only up to one job may conduct one of its user-disk IO operations. The duration of such IO operations is part of job definition.

The simulator driver will notify a student OS of a user-disk IO request through a system-call interrupt. A student OS can initiate a user-disk IO operation by method

```
public void userDiskIO(int jobID);
```

When the user-disk IO operation finishes, the simulator driver will issue a user-disk interrupt to the student OS by calling its interrupt handler method

```
public int userDiskInterrupt();
```

- **A pair of memory bounds registers (base address, length).**

Each job has a memory size. Before a job is swapped in to memory, a consecutive block of memory Ks equal to the job's memory size must first be allocated to it, and the starting address of this block is called the job's base address. When this

job is later scheduled to run, a student OS should load this job's base address and memory size values into the hardware based register and length register of the simulated machine. The simulated machine will make sure that all memory accesses issued from the job will be in the range from base address to (base address) + length - 1. These two registers are part of fundamental hardware supports for memory protection.

A student OS mainly interacts with these two memory bound registers with their setter methods:

```
public void setBaseAddressReg(int jobMemoryStartAddress);  
public void setLengthReg(int jobMemorySize);
```

- **An interval timer.**

The time unit of the interval timer register is a millisecond. When a student OS schedules a job for execution, it should set up in the interval hardware timer register of the simulated machine how many milliseconds the job is allowed to run. During the execution of this job, the simulator driver will reduce the value in this timer register for the CPU time used by the job. When this timer register reaches value zero, or the job's maximum allowed CPU time has been used up, a timer-run-out interrupt will be issued to the student OS for processing.

A student OS mainly interacts with the interval timer register with its setter method:

```
public void setTimer(int jobRunQuantum);
```

- **A system clock.**

The system clock provides the system time shared by all jobs and the simulator driver. The time unit used is a millisecond. A student OS can access the current system time from this system clock, but a student OS is not allowed to modify the value of this system clock. The system time is the base for job scheduling and performance measurements.

A student OS mainly interacts with the system clock with its getter method:

```
public long getSystemTime();
```

## 2.2 The Simulator Driver

The simulator driver is the coordinator between the simulated machine and a student OS. Its main functions include the following:

1. Job stream generation

The simulator driver will generate a stream of jobs to submit to the student OS for execution. The jobs can be generated with random number generators, or use a particular predefined job stream. For randomly generated job streams, a student OS can set a seed for the random number generators so that the same stream of jobs could be repeated later by specifying the same seed. By default, the current time will be used as the seed.

## 2. System time management

The simulator driver is responsible for advancing the system clock.

## 3. Interrupt generation

The simulator driver is responsible for generating interrupts and calling the corresponding interrupt handler methods of a student OS. The interrupts may be caused by system events including a new job arriving at the system-disk, the interval timer running out, the current job swapping completed, and the current user-disk IO operation completed. The interrupts may also be caused by a system call (trap) by the current running job. The supported system call types include *user-disk IO* (the current running job needs to read or write a user-disk file, non-blocking), *block* (the current running job cannot run until all of its pending user-disk IO operations finish), and *terminate* (the current running job terminates execution normally).

## 4. Job execution

Each of the student OS interrupt handlers will return one of two possible values: RUN for running the scheduled job, or IDLE for failing to find a ready job. If an interrupt handler returns RUN, the simulator driver will use the CPU of the simulated machine to run the scheduled job until the interval timer runs out, or the scheduled job has used up all of its maximally allowed CPU time, whichever comes first.

## 5. Job swapping

The simulator driver is responsible for carrying out job swap in/out initiated by a student OS. When the job swapping finishes, the simulator driver will issue a system-disk interrupt to the student OS by calling its method `systemDiskInterrupt()`.

## 6. Job user-disk IO

The simulator driver is responsible for carrying out a job's user-disk IO operations initiated by a student OS. When the current user-disk IO finishes, the

simulator driver will issue a user-disk interrupt to the student OS by calling its method `userDiskInterrupt()`.

#### 7. Validation of student OS scheduling

The simulator driver will validate any job scheduling or user-disk IO scheduling by a student OS before carrying it out.

#### 8. Validation of student OS resource management

The simulator driver will validate any resource management decisions, including memory management, by a student OS before carrying it out.

#### 9. Performance evaluation

The simulator driver reports the utilization of system resources including CPU, system-disk, and user-disk. It also reports the turnaround time or dilation ratio of the jobs. The performance measurements are defined in Subsection 2.6.

#### 10. System shutdown

At a predefined system shutdown time, the simulator driver will stop generating new jobs, and the system will actually shutdown when all system jobs finish execution. A student OS can also request an abrupt system shutdown at a specific time by calling method `setShutdownTime(int shutdownTime)` in method `startup()`.

### 2.3 Student Operating Systems

A student OS incarnates the main functions of a simple multiprogramming operating system. It is usually implemented as one or two classes. A student OS is mainly responsible for the following tasks:

#### 1. Job management

A student OS should declare a Job class (Process Control Block) to capture the state and information for a job, represent each job by a Job object, use some data structure to maintain all the Job objects for non-terminated jobs, and support the concept of ready queue and waiting queue.

#### 2. Memory management

A student OS should declare a data structure to manage the main memory, and support methods for the allocation and recycling of memory blocks for jobs.

### 3. Interrupt processing

The student OS should have a method as the interrupt handler for each type of system interrupts.

### 4. Long-term job scheduling

The student OS is responsible for designing efficient algorithms to choose and swap in new jobs on the system-disk to memory for execution. This scheduling policy will determine the degree of multiprogramming.

### 5. Medium-term job scheduling

The student OS is responsible for designing efficient algorithms to swap out partially executed jobs thus release their memory space, and swap in jobs with higher *scheduling priority* on the system-disk to improve system performance.

### 6. CPU scheduling

The student OS is responsible for designing efficient algorithms to schedule the ready jobs to run on the CPU.

### 7. Job user-disk IO operation management and scheduling

The student OS is responsible for the design and implementation of data structures and algorithms to maintain all the pending user-disk IO operations requested by the jobs, schedule their sequential execution, and initiate their execution.

## 2.4 User Jobs (Processes)

In this document, jobs mean processes. Jobs are generated by the simulator driver, and their characteristics are passed as parameter values to a student OS by an interrupt for new job's arrival on the system-disk.

### 2.4.1 Characteristics of a Job

At its birth, each job is characterized by the following information:

- Job ID: a unique positive integer associated with each job.
- Priority: an integer between 1 and 10 representing the job's priority, with 10 as the highest priority.
- Memory size: number of memory slots (in unit of K words) needed for the execution of the job.

- ❑ Maximum CPU time: maximum CPU time the job is allowed to use. The job should be terminated if it reaches its maximum CPU time.
- ❑ Job start time: time the job arrives on the system-disk.

After a job has been swapped in to the main memory, its job object also maintains the following information:

- ❑ Accumulated CPU time that it has used up
- ❑ Starting address of the allocated memory block

The time and type of system calls issued by a job as well as the time and frequency of user-disk IO operations requested by a job are determined by the simulation driver and passed to a student OS through interrupts.

### 2.4.2 Life Cycle of a Typical Job

A new job arrives on the system-disk magically. The simulator driver issues a *New Job* interrupt to inform a student OS of the new job's job ID, priority, memory size, maximum allowed CPU time, and arrival time. When the student has enough memory for the new job and decides to swap it in, it will issue a system-disk swap-in command to initiate the swap-in of the new job into the memory for execution. When the swap-in completes, a *System Disk* interrupt will be issued to the student OS, and the new job will first be inserted in the ready queue. When the student OS schedules it for execution, it will set up values of the base register, the length register, and the interval timer register for the job, and use an interrupt handler's return value to inform the simulation driver to run the job with the CPU. When the timer runs out or the job has reached its maximal allowed CPU time, a *Timer* interrupt will be generated by the simulator driver for the student OS to process. A job may issue one or more asynchronous user-disk IO operation requests through system calls. The student OS will manage such IO requests, and sequentially initiate them on the user-disk. When the user-disk finishes one IO operation, the simulator driver will issue a *User-Disk* interrupt to the student OS and inform the OS of the ID of the job that issued that user-disk IO operation request. If a job issues a *Block* system call, the job goes to the waiting queue, and it will not return to the ready queue until the user-disk has completed all of its pending IO operations. The student OS may choose to swap out a partially executed job back to the system-disk to vacate its memory space for jobs with higher scheduling priority, and later swap it in to memory to continue its execution. When a job issues a system call for termination, or when it has reached its maximum allowed CPU time, the job will terminate.

### 2.5 Student OS Interface

A student OS class must implement public methods to allow the simulator driver to initialize the OS and issue system interrupts.

### 2.5.1 Student OS Initialization

The student OS must implement the following method to initialize itself. This method is the first to be called by the simulator driver.

```
public void startup();
```

The following methods may be, and could only be, called inside the startup method:

- ❑ void setSeed(long seed)
- ❑ void setTrace(boolean v)
- ❑ void setShutdownTime(long time)

Refer to Section 4 for the definition of these methods.

### 2.5.2 Interrupt Calls of the Simulator Driver to a Student OS

The student OS must implement the following interrupt handler methods:

- ❑ public int newJobInterrupt(int jobID, int priority, int memorySize, long maxCpuTime)

A new job has just arrived on the system-disk. The parameters specify the job ID, priority, required memory size, and maximum allowed CPU time for the new job.

- ❑ public int systemCallInterrupt(int serviceType)

The job that was running has issued a system call (service call). The supported service types are:

- DISK\_IO : The job needs to read or write one of its data files on the user-disk.
- BLOCK : The job can no longer use the CPU and wishes to wait until all I/O requests that it issued have completed.
- TERMINATE : The job has completed execution.

- ❑ public int systemDiskInterrupt()

The most recently issued job swap-in or swap-out to the system disk has just finished.

- ❑ public int userDiskInterrupt()

The most recently issued I/O operation to the user disk has just finished.

- ❑ `public int timerInterrupt()`

The interval timer has run out, and the job that was running has used the CPU for an amount of time equal to the quantum specified by the student OS in the timer register, or has reached its maximum allowed CPU time.

### 2.5.3 Responses of a Student OS to Interrupt Calls

A student OS controls job execution by returning one of the following two mutually exclusive constant values back to the simulator driver:

- ❑ **RUN:** This causes the CPU to execute a particular user job, which must be in memory and not blocked. The bounds registers and interval timer must be set appropriately by the student OS for the job.
- ❑ **IDLE:** This indicates that the student OS found no ready job to execute.

### 2.5.4 System Time and CPU Time for a Job

At any time a student OS can get the current system time by calling method

```
public long getSystemTime();
```

The CPU execution for a scheduled run of a job starts when a student OS schedules it and returns constant **RUN** from one of its interrupt handler methods, and stops when the simulator driver calls one of the interrupt handler methods of the student OS the next time.

## 2.6 Performance Measurements

The simulator driver reports the statistics for the following performance measurements for a student OS:

- ❑ CPU utilization
- ❑ User-disk utilization
- ❑ System-disk utilization
- ❑ Job turnaround time for short jobs  
A job's turnaround time is the total time a job stayed in the system.
- ❑ Dilation ratio for long jobs  
A job's dilation ratio = (its turnaround time) / (its job time).  
A job's job time = (its total CPU time) + (its total user-disk IO time) + (its first swap-in time).

- Priority-weighted job turnaround time for short jobs  
Similar to job turnaround time, but weighted by job priorities.
- Priority-weighted dilation ratio for long jobs  
Similar to job dilation ratio, but weighted by job priorities.

## 2.7 Project Constraints

- We assume that interrupts are disabled while the student OS executes.
- `systemCallInterrupt(BLOCK)` issued by a job that has no outstanding I/O requests should be ignored (i.e., treated as a NOP or No Operation).
- A job must be aborted if it attempts to exceed its maximum allowed CPU time.
- A student OS must let dying jobs, either by a `systemCallInterrupt(TERMINATE)` or by attempting to exceed maximum allowed CPU time, finish any pending user-disk I/O operations.
- A dying job must be in memory. It should not be swapped out.
- A job active in a user-disk operation cannot be swapped out. Such a job is called “latched”.

### 3 Outline of a Student OS Class

```
import java.util.*;
import simulator.*;

public class OS extends simulator.InterruptHandlers {

    private Job[] jobTable = new Job[JOB_POOL_SIZE];

    // memory[] is for memory management. Each of its slots represents 1 K words.
    // Suppose memory[i] = k. If k = 0, memory[i] is free.
    // If k > 0, memory[i] belongs to the job with k as its job ID
    private int[] memory = new int[MEMORY_SIZE];

    // Data member declarations
    // Job table declaration
    // User-disk IO request queue declaration

    // startup() is the first method called by the simulator driver to initialize OS
    public void startup() {
        // Announce the launch of your OS
        // Initialize job table
        // Any other initialization for your OS
        // Call "setSeed(PREDEFINED_JOB_STREAM)" to use predefined job stream
        // Call "setSeed(123)" to use 123 as seed for random job generator. Every time
        // you use 123 as seed, you get the same job stream generated
        // Not calling "setSeed()" will make the current time be the seed
        // Call "setTrace(true)" if you want the dumping of detailed trace information
    }

    public int newJobInterrupt(int jobID, int priority, int size, long maxCPUtime) {
        // A new job just arrived on the system disk
        // The job is ready to be swapped in to memory and run.
        // Call getSystemTime() to get job submission time
        // Find a free entry in job table, and update it
        // If current job reaches its allowed max CPU time, process accordingly
        swapper();
        return scheduler();
    }

    public int systemCallInterrupt(int serviceType) {
        // Process system call interrupt based on service type:
        //   DISK_IO:    current job requests a user-disk I/O operation; non-blocking
        //               Enter this request in IO request queue. If it is the only one,
        //               issue userDiskIO to initiate data transfer for it
        //   BLOCK:     Put current job in waiting state until all its pending I/O
    }
}
```

```

        requests have been served
//  TERMINATE: current job finishes execution
//  Update job table
//  If current job reaches its allowed max CPU time, process accordingly
swapper();
return scheduler();
}

public int systemDiskInterrupt() {
//  Last job swap-in or swap-out just finished
//  Update job table
//  If current job reaches its allowed max CPU time, process accordingly
swapper();
return scheduler();
}

public int userDiskInterrupt() {
//  The first user-disk I/O request in the I/O request queue just finished
//  Delete the corresponding I/O request in the I/O request queue
//  Update job table and memory
//  If there are still requests in I/O request queue, issue userDiskIO for the first
//  request in the queue
//  If current job reaches its allowed max CPU time, process accordingly
swapper();
return scheduler();
}

public int timerInterrupt() {
//  Timer just ran out
//  If current job reaches its allowed max CPU time, process accordingly
swapper();
return scheduler();
}

private void swapper() {
//  Implement the function of both a long-term job scheduler and a
//  medium-term job scheduler
//  Decide whether to swap in a job, which job
//  Otherwise, decide whether to swap out a job, which job
//  If needed, use systemDiskJobSwap() to initiate the job swap
//  If needed, call findMemory() to get starting memory address for a swap-in
//  job
//  What is the best policy to choose a swap-in job?
//  What is the best policy to choose a swap-out job?
}

```

```

private int scheduler() {
    // Decide which ready job will run next
    // Function as short-term job scheduler (CPU scheduler)
    // What is your best CPU scheduling policy to optimize system performance?
    // Update memory bounds registers for the chosen ready job with
    //   methods setBaseAddressReg() and setLengthReg()
    // Decide how long the chosen job should run, and set timer with setTimer()
    //   accordingly
    // return RUN if a ready job found and chosen, IDLE otherwise
}

private int findMemory(int jobMemorySize) {
    // Find 'jobMemorySize' consecutive Ks of memory not in use
    // return its starting address if found, -1 otherwise
}

// more private utility methods

class Job {
    // Declare your Process Control Block (PCB)
    // It will be used to declare your job table
    // Carefully choose and document data members
}
}

```

## 4 Constants and Methods Inherited by a Student OS

### 4.1 Inherited Constants

- Memory size MEMORY\_SIZE in units of K words

```
public static final int MEMORY_SIZE
```

- Maximum job pool size JOB\_POOL\_SIZE, which defines maximum number of jobs that can be in the system *at the same time*

```
public static final int JOB_POOL_SIZE
```

- Interrupt handlers' return value: IDLE, RUN

```
public static final int IDLE  
public static final int RUN
```

- A special seed value for random number generator to indicate the use of a predefined job stream: PREDEFINED\_JOB\_STREAM

```
public static final long PREDEFINED_JOB_STREAM
```

- Job swap direction: SWAP\_IN, SWAP\_OUT

```
public static final int SWAP_IN  
public static final int SWAP_OUT
```

- Types of system calls: DISK\_IO, BLOCK, and TERMINATE

```
public static final int DISK_IO  
public static final int BLOCK  
public static final int TERMINATE
```

### 4.2 Inherited Methods

- **public void setSeed (long randomSeed)**

setSeed allows you to specify the job stream for a simulation session. It can be invoked only from the startup method, and only once. Its argument is used to seed random number generators that determine the characteristics of the jobs in a random fashion. If setSeed is not invoked, the random number is seeded dynamically with the system clock (each run is different). In that case, the value of the seed will be printed at the end of a run session so that the run can be re-enacted. This is useful for debugging and performance tuning. A predefined job

stream, hard-coded in the simulator, can be requested by passing the constant `PREDEFINED_JOB_STREAM` as the argument to `setSeed`.

❑ **public void setTrace (boolean traceSetting)**

`setTrace (true|false)` allows you to turn the tracing mechanism on or off. The default value is `false` (off). **WARNING:** `setTrace (true)` produces a verbose description of each event and results in an extremely large amount of output. It should be used only as a troubleshooting aid. Even with the trace off, performance statistics are generated at regular intervals and a diagnostic message appears in case of a crash. In either case, your OS need not print anything. `setTrace` should be called in `startup()` only.

❑ **public void userDiskIO(int jobID)**

This method is used for initiating a user-disk I/O. It starts an I/O for the specified job on the user-disk but does not wait for the I/O to complete. The method call returns immediately. A `userDiskInterrupt` at a later time will notify the OS that the I/O is complete.

❑ **public void systemDiskJobSwap (int jobID, int memorySize, int startAddress, int swapDirection)**

This method is used for initiating a system-disk (swap) I/O. It starts a swap of the specified job. Use one of the following constants for `swapDirection`: `SWAP_IN` for read (i.e. system disk to memory), and `SWAP_OUT` for write (i.e. memory to system disk). The method call returns immediately, but the swap will not have completed yet. A `systemDiskInterrupt` at a later time will notify the OS that the swap is complete.

❑ **public void setBaseAddressReg (int startAddress)**

Setter for machine's base address register

❑ **public int getBaseAddressReg ()**

Getter for machine's base address register

❑ **public void setLengthReg (int jobMemoryLength)**

Setter for machine's job memory length register

❑ **public int getLengthReg ()**

Getter for machine's job memory length register

❑ **public void setTimer (long runOutTime)**

Setter for job's interval timer. A timerInterrupt will normally be generated later when the current job has run for runOutTime milliseconds.

□ **public long getTimer ()**

Getter for job's interval timer

□ **public long getSystemTime()**

for system clock time. In each interrupt handler, the current time is obtained through this method.

□ **public void setShutdownTime(long time)**

Shutting down a simulation run abruptly at the specified time.

## 5 Source Code for a Simplified Demonstration OS: DemoOS.java

```
// A simplified working OS example as base for student projects
import simulator.*;
import java.util.*;

public class DemoOS extends InterruptHandlers {

    private int[] memory = new int[MEMORY_SIZE];
    private Job[] jobTable = new Job[JOB_POOL_SIZE];

    final int JOB_QUANTUM = 100; // default length (milliseconds) of each job run

    private int runningJobIndex = -1; // jobTable index for the running job
    private int swappingJobIndex = -1; // jobTable index for the job being swapped in or out

    private long jobRunStartTime = -1; // start time of the current job run
    private LinkedList ioQueue = new LinkedList(); // user-disk IO request queue

    public void startup() {
        System.out.println("Demo OS Launched");
        for (int i=0; i<jobTable.length; i++)
            jobTable[i] = new Job();
        setSeed(PREDEFINED_JOB_STREAM);
    }

    // Interrupt handler for the arrival of a new job on the system-disk
    public int newJobInterrupt(int jobID, int priority, int memorySize, long maxCpuTime) {
        updateCpuTime();
        int jobIndex = findFreeJobEntry();
        Job job = jobTable[jobIndex];
        job.jobID = jobID;
        job.priority = priority;
        job.memorySize = memorySize;
        job.maxCpuTime = maxCpuTime;
        job.usedCpuTime = 0;
        job.isFree = false;
        job.isInMemory = false;
        job.isBlocked = false;
        job.isDying = false;
        job.nbrPendingIO = 0;
        swapper();
        return scheduler();
    }
}
```

```

// Interrupt handler for a system call from the current running job
public int systemCallInterrupt(int serviceType) {
    updateCpuTime();
    Job job = jobTable[runningJobIndex];
    switch (serviceType) {
        case BLOCK:
            if (job.nbrPendingIO > 0)
                job.isBlocked = true;
            break;
        case TERMINATE:
            if (job.nbrPendingIO == 0) {
                job.isFree = true;
                freeMemory(job);
            }
            else
                job.isDying = true;
            break;
        case DISK_IO:
            ioQueue.addLast(new Integer(runningJobIndex));
            job.nbrPendingIO++;
            if (ioQueue.size()==1) // this is the only IO request; start it
                userDiskIO(job.jobID);
            break;
    }
    swapper();
    return scheduler();
}

```

```

// Interrupt handler for the completion of a job swap in/out
public int systemDiskInterrupt() {
    updateCpuTime();
    Job job = jobTable[swappingJobIndex];
    job.isInMemory = !job.isInMemory;
    if (job.isInMemory == false)
        freeMemory(job);
    swappingJobIndex = -1;
    swapper();
    return scheduler();
}

```

```

// Interrupt handler for the completion of the first user disk I/O request on the ioQueue
public int userDiskInterrupt() {
    updateCpuTime();
    int jobIndex = ((Integer)ioQueue.removeFirst()).intValue();
    Job job = jobTable[jobIndex];
    job.nbrPendingIO--;
}

```

```

if (job.nbrPendingIO == 0) {
    job.isBlocked = false;
    if (job.isDying) {
        // this is the last I/O for a dead job; release resources
        job.isFree = true;
        freeMemory(job);
    }
}
// keep user-disk spinning
if (ioQueue.size() > 0) {
    jobIndex = ((Integer)ioQueue.getFirst()).intValue();
    userDiskIO(jobTable[jobIndex].jobID);
}
swapper();
return scheduler();
}

// Interrupt handler for the current job having used up its time quantum
public int timerInterrupt() {
    updateCpuTime();
    swapper();
    return scheduler();
}

// Update CPU time for the current running job
private void updateCpuTime() {
    if (runningJobIndex == -1) return;
    Job job = jobTable[runningJobIndex];
    job.usedCpuTime += getSystemTime()-jobRunStartTime;
    if (job.usedCpuTime >= job.maxCpuTime) { // max CPU time used up; terminate
        if (job.nbrPendingIO == 0) {
            job.isFree = true;
            freeMemory(job);
        }
        else
            job.isDying = true;
    }
}

// Schedule job swap-in or swap-out: long-term job scheduling only
private void swapper() {
    if (swappingJobIndex >= 0) return; // already have a swapping in progress

    for (int i = 0; i < jobTable.length; i++) {
        Job job = jobTable[i];
        if (!job.isFree && !job.isInMemory && !job.isDying) {

```

```

    int startAddress = findMemory(job.memorySize);
    if (startAddress == -1) // failed to find memory
        continue;
    job.startAddress = startAddress;
    allocateMemory(job, startAddress);
    swappingJobIndex = i;
    systemDiskJobSwap(job.jobID, job.memorySize, startAddress, SWAP_IN);
    return;
}
}
}

```

```

// Short-term job scheduler, or CPU scheduler: round-robin
private int scheduler() {
    jobRunStartTime = getSystemTime(); // record new job run's start time
    if (runningJobIndex == -1)
        runningJobIndex = 0;
    for (int j = 0; j < jobTable.length; j++) {
        int i = (j + runningJobIndex + 1) % jobTable.length;
        if (i == swappingJobIndex) continue; // a swapping job is not a ready job
        Job job = jobTable[i];
        if (!job.isFree && job.isInMemory && !job.isBlocked && !job.isDying) {
            // found a ready job
            runningJobIndex = i;
            setBaseAddressReg(job.startAddress);
            setLengthReg(job.memorySize);
            long timeLeft = job.maxCpuTime - job.usedCpuTime;
            if (timeLeft < JOB_QUANTUM)
                setTimer(timeLeft);
            else
                setTimer(JOB_QUANTUM);
            return RUN;
        }
    }
    runningJobIndex = -1;
    return IDLE;
}

```

```

// Return the index of a free job table entry
private int findFreeJobEntry() {
    int i;
    for (i=0; (i<jobTable.length) && !jobTable[i].isFree; i++);
    if (i == jobTable.length)
        crash("Job Table Overflow");
    return i;
}

```

```

// Find the required number of consecutive memory slots
// Return starting address of the slots if found, -1 otherwise
private int findMemory(int size) {
    int i;
    // find beginning of free slots in memory
    for (i=0; i<memory.length; i++) {
        int count = 0;
        if (memory[i] == 0 && (memory.length-i >= size)) {
            for (int j=i; j<i+size; j++) {
                if (memory[j] == 0) // free
                    count++;
                else
                    break;
            }
            if (count >= size)
                break;
        }
    } // end for i
    if (i == memory.length)
        return -1;
    else
        return i;
}

// Free memory slots for a terminated job
private void freeMemory(Job job) {
    for (int i = job.startAddress; i < job.startAddress+job.memorySize; i++)
        memory[i] = 0;
}

// Allocate memory for the job
private void allocateMemory(Job job, int memoryIndex) {
    for (int i = job.startAddress; i < job.startAddress+job.memorySize; i++)
        memory[i] = job.jobID;
}

// Print error message and terminate the running session
private void crash(String errorMessage) {
    System.out.println(errorMessage);
    System.exit(-1);
}

// Process Control Block (PCB)
class Job {
    int        jobID;                // Job's unique ID, an integer > 0

```

```

int      priority;          // Priority of the job: 1-10, 1 as the lowest
int      memorySize;       // Job's memory size requirement
long     maxCpuTime;       // CPU time limit for the job
long     startTime;        // Job's starting time
int      startAddress;     // Job's memory starting address (slot number)
long     usedCpuTime;      // Number of milliseconds the job has used up on
                          // the CPU
boolean  isInMemory;       // Job is in memory
int      nbrPendingIO;     // Number of pending user disk IO requests for the
                          // job
boolean  isBlocked;        // The job is blocked: it cannot run until all its
                          // pending user-disk I/O operations finish
boolean  isDying;          // The job is dying but it has pending user-disk I/O
                          // operations to wait for
boolean  isFree;           // This job object can be recycled

Job() {
    isFree = true;
}
}
}

```

## 6 How to Compile and Run Student OS with Sun JDK

- To compile a student OS with Sun JDK:
  - a. Download and install a Java 2 (version 1.2 or higher) JDK
  - b. Create a directory for your project
  - c. Get a copy of file `ossim.jar` from your professor, which contains all supporting class files for your project, and copy it into your project directory
  - d. Use a text editor to create your OS in a file called `OS.java` in your project directory
  - e. Inside your project directory, use a command-prompt window to run

**`javac -classpath .;ossim.jar OS.java`**

- To run the Pace OS Simulator with your OS:

The simulator and your OS execute together using class `Run` included in `ossim.jar`. By default, `Run` looks for a Java class named `OS.class` in the current directory, but you can specify a different class on the command line if you wish. This feature can be used to test different versions of your OS. Class `Run` can be run with several options, none of which are required:

**`java -classpath .;ossim.jar Run [-debug | -report | -trace | -help]  
[ ClassName ]`**

`-debug`

Starts the simulator with a graphical debugger. You can set breakpoints for each interrupt handler in your OS. The simulation will pause before entering your code and just after leaving your code. Values of the registers and other state information are displayed along with the simulator's statistics, trace data, and any output from your OS.

`-report`

Runs the simulator, redirecting statistics output to a file named `report.txt` in the current directory. It can be used to produce final output to hand in. If both `-debug` and `-report` are specified, the simulator will be run in debug mode.

`-trace`

Turns on the trace facility of the simulator driver

`-help`

Prints the command syntax.

`ClassName`

Used to specify a class name other than OS. The class must extend InterruptHandlers.

If no options are specified, the simulator will run with output directed to the console and using the OS in OS.class.

- To run the simulator with the provided Demo OS:
  - a. Copy the provided files DemoOS.class and DemoOS\$Job.class in your project directory (skip this step if the two files are included in ossim.jar).
  - b. Inside your project directory, use a command-prompt window to run

```
java -classpath .;ossim.jar Run DemoOS
```

or

```
java -classpath .;ossim.jar Run DemoOS -debug
```

## 7 Pace Operating Systems Simulation Debugger

The graphics user interface for the Pace OS simulation debugger looks like the following. You can use the first row of check boxes to choose what kind of interrupts that you want to debug. In the table, you can read the values of registers, job attributes, and performance measurements immediately before and after your interrupt handlers are called. The memory map shows you visually the memory occupancy by the jobs. Different letters represent different jobs, but the letters are dynamically assigned and not associated with particular jobs. The letter to the right of the current Job ID represents the current job in the memory map. Table entry for “Number of Jobs in System” gives both total number jobs in the system and total number of jobs in memory. The Simulator Messages pane is used to print out messages from the OS simulator. The OS Messages panel is used to print out messages from your OS class. You can use button “Set Trace On/Off” to turn on/off the trace facility at run time thus control the volume of information shown in the Simulator Message pane. If you have checked some interrupt types for debugging, you can then use button “Run to Next Break” to step through the execution. At any time of your debugging session, you can click on button “Run to End” to ignore all break points and execute the session to the end. You can click on the button “Save Report” to save the information in the Simulator Message pane in a file of your choice. You can terminate the simulation at any time by clicking on the button “Exit”.

The two message panes are part of a split pane, so you can easily close one of the two panes and let another one have more visible space.

**Check BreakPoints to Enable**

New Job     Timer Run Off     System Disk     Disk I/O     System Call

**Exiting System Disk(Swap) Interrupt**

Simulator Item	Value Entering	Value Exiting
Job ID	35 : a	48 : b
CPU Time [Used, Max]	[3950, 27366]	[1666, 25400]
Base Address Register	16	0
Length Register	8	9
Clock Register	64988	64988
Timer Register	100	100
CPU State Before/After Interrupt	Run	Run
Number of Jobs in System	10 [6 in memory]	10 [6 in memory]
I/Os Pending?	false	false
System Disk Busy?	false	false
User Disk Busy?	false	false
Average CPU Utilization (%)	84.52	84.52

**Memory Map:** |bbbbbbbbb-----aaaaaaaaadddddddddddddddddddccccccc-----eeeeeeefffffffffff-----|

**Simulator Messages**

```

50 13 58120 0 23 0 2 no no no no
51 24 59746 0 854 0 8 no no no no

Total Jobs: 51   Completed: 42

% Utilization   CPU: 84.52   Memory: 47.59
                User Disk: 3.89   System Disk: 3.61

* Time: 62636; job 52 arrived; size 12; priority 2; max CPU time 31
* Time: 63949; job 53 arrived; size 9; priority 5; max CPU time 103
* Time: 64916; job 47 completed; job started at time 55275

```

**OS Messages**

```

Demo OS Launched

```

Buttons: Set Trace On, Run to Next Break, Run to End, Save Report, Exit

## 8 Potential Student Projects Based on the Pace OS Simulator

- Long-term job scheduling
- Medium-term job scheduling
- Short-term job (CPU) scheduling
- Priority-based job scheduling
- Memory management (first-fit, best-fit, ...)
- Job queues management
- Disk IO request management
- .....

## 9 Appendix: Pace OS Simulator Error Messages

- \*\*\* Set Seed Error \*\* setSeed called more than once \*\*\*
- \*\*\* Set Seed Error \*\* Job stream cannot be changed once simulation run begins \*\*\*
- \*\*\* Startup Error \*\* OS is not allowed to initialize system clock \*\*\*
- \*\*\* OS is not allowed to change system clock \*\*\*
- \*\*\* Incorrect CPU state set by OS -- must be IDLE or RUN \*\*\*
- \*\*\* Too many jobs in system -- OS processing jobs too slowly \*\*\*
- \*\*\* System Disk Job Swap Error -- System disk is busy \*\*\*
- \*\*\* System Disk Job Swap Error -- Job does not exist \*\*\*"
- \*\*\* System Disk Job Swap Error -- Size of job specified is incorrect \*\*\*
- \*\*\* System Disk Job Swap Error -- Job specified has completed \*\*\*
- \*\*\* System Disk Job Swap Error -- Job specified not in memory \*\*\*
- \*\*\* System Disk Job Swap Error -- Job has active IO - it cannot be swapped \*\*\*
- \*\*\* System Disk Job Swap Error -- Direction of swap is not SWAP\_IN or SWAP\_OUT \*\*\*
- \*\*\* System Disk Job Swap Error -- Starting address of job specified is incorrect \*\*\*
- \*\*\* System Disk Job Swap Error -- Job specified is already in memory -- no need to swap it in \*\*\*
- \*\*\* System Disk Job Swap Error -- Starting address is negative \*\*\*
- \*\*\* System Disk Job Swap Error -- Job memory addresses overlap other job's \*\*\*
- \*\*\* System Disk Job Swap Error -- Job ID specified is negative or zero \*\*\*
- \*\*\* System Disk Job Swap Error -- Attempt to swap job with size negative or zero \*\*\*
- \*\*\* System Disk Job Swap Error -- Attempt to swap in job beyond end of memory \*\*\*
- \*\*\* Run Error -- Bounds Registers are incorrect \*\*\*
- \*\*\* Run Error -- Timer is negative or zero \*\*\*
- \*\*\* Run Error -- Job specified to run is blocked \*\*\*
- \*\*\* Run Error -- Job terminated or has reached Maximum CPU time \*\*\*
- \*\*\* Run Error -- Timer exceeds job's remaining CPU time \*\*\*

- \*\*\* Idle Error -- Unblocked non-dead job exists in memory \*\*\*
- \*\*\* Job in memory with pending I/O but user disk idle \*\*\*
- \*\*\* User Disk Error -- Job specified does not exist \*\*\*
- \*\*\* User Disk Error -- Job specified not in memory \*\*\*
- \*\*\* User Disk Error -- Job specified has completed \*\*\*
- \*\*\* User Disk Error -- Job specified has no pending I/O \*\*\*
- \*\*\* User Disk Error -- User disk is busy \*\*\*
- \*\*\* User Disk Error -- Job ID specified is negative or zero \*\*\*
- \*\*\* OS failed to swap jobs from system disk into empty memory \*\*\*