Guide to the Java Version of the Simple Operating System (SOS) Simulator

Charles Crowley
August 1997

Introduction

The book *Operating Systems: A Design-Oriented Approach* (Charles Crowley, Irwin, 1997) contains code for a simple operating system called SOS. That code is written in C++. The SOS code in the book assumes it is running on a hardware platform called the CRA-1. This is a simple RISC machine that was developed just for the book. A certain amount of the SOS code depends on the details of this hardware architecture.

Two simulators exist that will run (modified versions of) this code.

The C++ Version of the SOS Simulator

The C++ version of the SOS simulator only works on UNIX systems and is available from http://www.cs.unm.edu/~crowley. This simulator does not implement the CRA-1. To ease the coding burden, the simulation was based on a MIPS simulator and a simulation architecture used in NACHOS (http://http.cs.berkeley.edu/~tea/nachos/). One aspect of this simulation architecture is that the operating system code itself does not run on the simulator, only the user process code does. As a consequence of this, the view of the hardware is different from that in the book. The details of saving and restoring processor state, for example, are still present in the simulation but are distinctly different from the code in the book and different from any code based on a real machine. For example, registers are set with function calls to the simulator code. As a result, this simulation comes with a special version of the SOS source code. It is still in C++ but it is changed to conform to the hardware simulation architecture. That simulation comes with commentary describing the differences from the SOS code in the book.

There are a few problems with this simulation. The main one is that it works only on UNIX systems and would require considerable effort to port to Windows. A minor problem is that it is text-only to avoid dependencies with graphics systems. Another problem is that it does not simulate the code in chapters 6 and 8 of Crowley's *Operating Systems* because the parallelism cannot be easily simulated using the simulation architecture it is based on.

The Java Version of the SOS Simulator

To deal with the portability issue I have developed a Java version of the SOS source code and a virtual machine simulator (in Java) for it to run on. The main advantage of this version is that it works on all systems that support Java. It will work on a web page with a Java-enabled browser so almost anyone can run the system.

The source code of the Java version of SOS is based on the book version but many changes were necessary. Of course, the conversion from C++ to Java required many small changes although the basic look of the code is quite similar. The change in virtual machine it runs on also required a number of changes in the code.

This document describes the Java version of the SOS simulator. It documents all the changes that were necessary for the conversion.

How To Run Java SOS

The Java simulator can be run with a Java interpreter or with a Java-enabled web browser. First I will describe the Java files necessary to run the simulator and then I will discuss how to run it.

The files

The Java SOS simulation code comprises the following groups of files. All files are written in version 1.0 of Java and the AWT.

- *Application files*, which implement test applications to run on SOS.
    - `AppGUICounter.java` contains an SOS application that puts up a simple GUI which contains a counter that you can start and stop with buttons.
    - `AppTests.java` contains several SOS test programs, some of which start AppGUICounters.
- *Simulation files*, which implement the virtual hardware SOS runs on and the simulation control code, which runs the simulation and creates the GUI to control the simulation. This is all code that would not be present in an operating system running on real hardware.
    - `HWSimulation.java` contains to code that simulates the virtual machine used by the simulation.
    - `SIM.java` contains the main function and the code that runs the simulation. Most of this code is to implement the user interface using the Java AWT.
    - `SIMIntHandler.java` is an interface definition for interrupt handlers.
    - `SIMList.java` redefined the List AWT component to have the dimensions I need for the GUI.
    - `SIMPauser.java` contains a class that allows SOS to be paused.
- *SOS source code files*, which is the source code of the system.
    - `SOSConstants.java` contains the constants required by SOS.
    - `SOSData.java` contains the class that defines all the data used by SOS.
    - `SOSDiskDriver.java` is the disk subsystem code.
    - `SOSDiskIntHandler.java` is the disk interrupt handler.
    - `SOSDiskRequest.java` is the class that defines items in the disk request queue.
    - `SOSMem.java` deals with memory management and is not used in the current version of the SOS simulator.
    - `SOSProcessDescriptor.java` is the class that defines a process descriptor.
    - `SOSProcessManager.java` is the code that handles process management.
    - `SOSProgIntHandler.java` is the program error interrupt handler.
    - `SOSStart.java` contains the code that is called when SOS starts up.
    - `SOSSyscallIntHandler.java` is the system call interrupt handler.
    - `SOSTimerIntHandler.java` is the timer interrupt handler.
    - `SOSWaitQueueItem.java` is the class that defines the items that go into the message wait queues.

Starting the simulation using a Java interpreter.

To run the simulator you need all these files and you need to compile them into "`.class`" files. Then you run the simulation by running the class "SIM" (e.g., `Java SIM`). The files are all written in version 1.0 of Java and the AWT.

Starting the simulation from a web browser.

You can also run the simulation from a Java-enabled web browser by loading the file `SIM.html`. The file `SIM.class` and all the other `.class` files all need to be in the `CLASSPATH`. Normally it is sufficient to have them all in the same directory as `SIM.html`.

Or you can run it from my web page (http://www.cs.unm.edu/~crowley/SIM.html).

Running the simulation

When the simulation starts you will see a single window (or a new web page in your browser). You can select from several choices before you run the simulation.

1. Pick an application to run among the choices in the radio buttons at the top of the window. You have the following choices:
    - *2 GUI Apps*: this will start two of the GUI counter apps. The default time slice is 5 seconds so you will see one count for five seconds and then the other counts for 5 seconds, etc.
    - *MsgApp*: this will start two processes that send create a message queue and use it to send messages one to the other. The sender sends 10 messages and then exits. These processes have no GUI but they do generate trace messages.
    - *Disk App*: this starts a process that writes the disk and then reads it back. It does this 10 times. It has no GUI but it does generate trace messages.
2. Pick which things you want traced during execution. These trace messages show up in the list boxes you see in the simulation window. You have the following choices:
    - *App*: show application trace messages. You generally want this on. Applications usually do not generate all that many trace messages.
    - *HW*: show trace message relating to the hardware simulation. You do not normally want to see these. There are a fair number of HW trace messages.
    - *SIM*: show trace messages relating to the software simulation. You do not normally want to see these messages.
    - *Syscall*: show the messages generated by the system call interrupt handler in SOS.
    - *PM*: show the trace messages generated by the process manager component of SOS
    - *Disk*: show trace messages generated by the disk subsystem of SOS.
3. Start the simulation by clicking on the “Start SOS” button.
4. During execution all trace messages will go to the list box under the top row of buttons. You can pause and resume the simulation with the “Pause SOS” and “Resume SOS” buttons.
5. When you are finished click on the “Exit Simulator” button.
6. The lower list box shows the trace messages divided up based on the current process that was running when the trace message is created. This is a separate list for each process in the system. These list boxes are in an AWT “card deck” which means that you can only see one of them at a time. There is a row of buttons above the lower list box that controls which of the process list boxes you see. The “Next Process” button goes through all the processes and you can go directly to processes 1, 2, 3, or 4 with buttons. The process 0 list is for trace messages generated when there is no current process.
7. These list boxes have scroll bars if necessary that allow you to look through the messages.

Modifying and testing SOS

You can edit any of the SOS files to make changes in the operation of SOS and run the simulation to see how they work. Just edit the files, recompile them, and run SIM again. You could run it under the Java debugger if you run into exceptions. This will allow you to program all the exercises from the book that involve modifications to SOS and test them on the simulator.

The Java Virtual Machine

The hardware virtual machine in this simulator is not really too much like a real machine but it does the job of providing a base for the simulation. All of the simulation code is in `HWSimulation.java` and `SIM.java`. The simulation has several parts:

- Simulation of running processes
- Memory simulation
- Interrupt simulation
- Timer simulation
- Disk simulation

Let's look at each of these in turn.

Simulation of running processes

There are four calls related to running processes.

- `CreateProcess()`: This call creates a simulated process. A process will be a Java class that implements the `Runnable` interface. This call creates a Java thread for each process. All the threads run at normal priority. The thread is created and started. We rely on the application to suspend itself immediately after it is started. In a real machine creating a process would be done by (1) loading the code for the process into memory and (2) creating and initializing a register save area for the process. In the Java simulation, the Java interpreter takes care of loading the code and it also handles the context switching.
- `RunProcess()`: This call runs a process by resuming the thread that represents it. In a real machine this would be done by loading the registers from the save area and then executing a special instruction that loads the program counter and the status word.
- `WaitForInterrupt()`: This call suspends the calling thread and so causes it to wait for the next interrupt (usually a disk interrupt).
- `SystemCall(int arg)`: This call simulates a system call. The current thread (of the user process) is used to execute the operating system code in the system call interrupt handler. This call generates an interrupt that calls the system call interrupt handler. When the system call interrupt handler returns the code for this call will suspend the thread of the process making the system call unless that same process was chosen to run again by the dispatcher.

Interrupt simulation

There are four interrupts: system call, timer, program error, and disk. All interrupt handlers are classes that implement the `SIMIntHandler` interface, which consists of one operation (`HandleInterrupt(int arg)`). When the system is initialized, handles to each of the four interrupt handlers is placed in specific memory location reserved for the interrupt vector area. An interrupt is handled by fetching the appropriate handler and calling its `HandleInterrupt` function.

Memory simulation

Memory simulation is not as realistic as it is in the book (CRA-1) or the UNIX simulator. Since Java itself takes care of allocating memory for Java classes and for loading them into memory, the SOS simulator cannot do that. Instead it only simulates part of the data memory. There is a large array of cells that represents the physical memory of the virtual machine. Currently it contains 10,000 cells. SOS uses 1000 cells for itself and allocates 1000 cells to each process. In a real machine, each cell would be a byte but in the simulation each cell is a Java Object. This allows us to store object handles (such as interrupt handlers) in the simulated memory. Most uses of the memory store integers (actually Java Integer objects) and so there are special memory access functions that treat the memory cells as integers.

Base/bound memory mapping is simulated. The simulated hardware base and bound registers can be set and read. There are mapped and unmapped versions of all the memory access functions. The unmapped versions access the array directing using the memory address provided as an array index. The mapped versions add the base register to the address provided and also check it against the limit register.

This simulated memory is accessed with the following calls:

- `Object GetCellUnmapped(int address)`: This call goes to index `address` of the physical memory array and returns the Object stored there. This is the basic memory fetch operation.
- `int GetCellUnmappedAsInt(int address)`: This is the same as `GetCellUnmapped` except that it converts the Object into an integer before it returns it.
- `Object GetCell(int address)`: This call is the same as `GetCellUnmapped` except it adds the memory base register to the address before it uses the address to index into the physical memory array. This simulates a base/limit register memory-mapping scheme. If the original value of `address` is equal to or greater than the limit register then a program error interrupt is generated. This simulates a base/limit register memory-mapping scheme.
- `int GetCellAsInt(int address)`: This is the same as `GetCell` except it converts the Object into an integer before returning it. This is a convenience function since integers are the main thing stored in memory cells.
- `void SetCellUnmapped(int address, Object obj)`: This call stores `obj` in cell `address` of the physical memory array.
- `void SetCellUnmappedAsInt(int address, int n)`: This call converts `n` to an Integer object and stores it in cell `address` of the physical memory array.
- `void SetCell(int address, Object obj)`: This call adds the base register of the current process to `address` and then stores `obj` in cell `address` of the physical memory array. If the original value of `address` is equal to or greater than the limit register then a program error interrupt is generated.
- `void SetCellAsInt(int address, int n)`: This call is the same as `SetCell` except it stores an integer (converted into a Java Integer object) in the cell..

The operating system keeps a number of things in the simulated memory.

- Cells 0-3 contain handler of the four interrupt handlers
- Cells 10-12 contain the three registers of the simulated disk. These addresses are handled specially by the memory access functions. Stores and fetches to these cells are passed on to the simulated disk. This is the simulation of memory-mapped I/O. Nothing is ever actually stored in cells 10 through 12.
- Cells 300-427 contain the system disk buffer. The simulated disk reads from the disk into these cells and writes to the disk from these cells.
- Cells 500-899 contain the 50 system message buffers.
- Cells 1000-1999 are allocated to process 1.
- Cells 2000-2999 are allocated to process 2.
- Cells 3000-3999 are allocated to process 3.
- And so on for processes 4, 5, 6, 7, and 8.

Each process keeps a few things in the simulated physical memory. The simulated memory is the common memory between processes and the operating system and all data passed between them goes through the simulated physical memory.

- System call arguments are passed in memory cells and the system call return argument is passed back in a memory cell.
- Message buffers sent and received are in memory cells.
- Later applications will put other things in system memory when I add virtual memory capabilities to the simulator.

Right now SOS allocates 1000 cells to each process and only base/limit memory mapping is implemented. Later versions of the Java SOS simulator will implement paging in the simulated memory and applications will use the simulated memory for calculations that test the effectiveness of the virtual memory.

Timer simulation

The simulated hardware timer is quite similar to the one defined for the CRA-1 in the book. The operating system sets the timer with a value and the timer counts the initial value down to 0. When the timer value gets to 0, a timer interrupt is generated and counting stops. There is only one call:

- `int setTimer(int time_to_interrupt)`: This call sets the timer to the specified value and returns the value that was left in the timer before it was reset. This has the effect of starting a new time interval if the timer value was 0 and canceling the time interval (and possibly starting a new one) if the timer was not 0. If the value set is 0 this turns off the timer (and cancels and pending time interval).

The timer has its own Java thread and ticks continually, even when there is no timer interval.

Disk simulation

The disk simulation is quite simple. It has its own Java thread. It is a loop that continually checks for a disk request. A disk request is made by storing into the disk's control register. This is done by storing in memory cell 10. This then calls the appropriate call in the disk simulation. Once a disk command is issued the disk simulation delays 500 ms, transfers the data, and then generates the disk interrupt.

The disk class defines two functions:

- `int GetStatusRegister()`: This function returns a status word that tells if the disk is busy or not.
- `void SetAddressRegister(int address)`: This function sets the memory address (in the physical memory array) where the next disk transfer should come from or go to.
- `void SetCommandRegister(int address)`: This function sets the command register which has the side effect of starting the disk transfer.

These disk functions are never called by SOS but only by the memory simulation code which simulates memory mapped I/O by treating memory cells 10, 11, and 12 as disk registers.

Later versions of the simulator will use a more sophisticated disk simulation so that disk accesses will take more or less time depending on where the read head is located when the request is started.

The Simulation

The hardware simulation provides processes (using threads), memory (using an array of Objects), a timer (using a thread and sleep statements), a disk (also using a thread and sleep statements), and hardware services (creating processes, running processes, system call, access to memory, access to the disk). The simulation code builds on this to provide a user interface to the hardware simulation.

The simulation code generates the user interfaces described in the section **How To Run Java SOS** above. The user interface mainly displays trace messages sent by SOS and the application processes. Most of the simulation code is AWT calls to create and manage the user interface.

It also defines a SIMList class which subclasses the AWT List class to be a specific size.

The Test Applications

The file `AppGUICounter.java` creates a GUI to a simple counter. A counter was chosen because it is obvious from looking at it when it is running and when it is idle.

The file `AppTests.java` implements the `AppTests` class. An `AppTests` object can be any of four different applications.

- Application 1 creates two AppGUICounter applications and then exits. This application tests the process switching code.
- Application 2 creates a message queue, creates an application of type 3 (passing it the message queue id), and then enter a loop where it sends a message to the message queue every 500 ms. This application (along with application 3) tests the message passing system calls.
- Application 3 is a loop that reads messages from the message queue created by an application 2 process.
- Application 4 is a loop that writes then reads the same disk block (with a different number in it each time). This application tests the disk system calls.

These applications use simulated memory cells 101 to 103 for system call arguments, cell 100 for system call return values, and cells 200-207 for a message buffer. These applications also include calls to the `Trace` function to generate trace messages.

Each application suspends itself immediately after it is started. This is necessary because of the way the SOS simulator using suspend and resume for dispatching.

Difference Between the Book's SOS and Java SOS

There are three sources of differences between the book version of the SOS code and the version in the Java SOS.

- Difference due to Java
- Difference due to the change in the underlying virtual machine
- Other differences

Difference due to Java

Java is similar to C++ but there are still many differences between the languages. These differences required many small changes in the SOS code when it was converted. Here are the main categories of changes:

Defined constants

In Java we use the `static final` modifiers to define constants.

Use of Vector

I have used the `Vector` class from the java.util library to implement queues. In the book we assumed a `Queue` class. This changes the declarations and how the queue operations are specified.

Visibility of names

Java does not have a global name space (or rather only class names exist in the global name space) and all names must exist inside of a class. This means that many names that appeared without any qualification in the C++ version now require a class qualification. All of the constants are static final variables in some class. The same is true of many function calls. The C++ version did not require qualification but the Java version does.

This is probably the most noticeable change in that it affects the most the lines of code.

Construction of objects

Java requires that all the elements of an array be constructed individually. This requires some new code in the class constructors.

Difference due to the change in virtual machine

Process handling

Processes have to be managed by calls to the hardware simulation. This includes calls to create a process, run a process, make a system call, and access memory. Java handles process state so no process state needs to be saved or restored. When a process is started, its base and limit registers need to be loaded into the simulated hardware base and limit registers.

The Java thread suspend command is used in various places to handle the simulation of processes with threads.

Memory

Memory access is through function calls (`GetCell` and `SetCell`)rather than simply accessing variables. The simulated hardware base and limit registers must be set explicitly (they are public variables in the `HWSimulation` class.

We use a generic `MemoryCopy` function to transfer data between system and user memory. This replaces the `CopyToSystemSpace` and `CopyFromSystemSpace` functions from the book.

Because the memory array holds Java objects we frequently have to convert between the Integer class and ints.

System calls

System calls are function calls. System call arguments and return values are passed in memory cells.

Interrupt handling

The interrupt vectors are set up with Java code that puts class handles in the interrupt vector area..

Timer

The timer is set using a function call (`SetTimer`).

Disk

The disk interface is quite similar.

Other differences

Tracing

I have added extensive tracing of SOS operations. This tracing code does not appear in the book version of SOS. This code is easy to recognize.

Reformatting and other small changes

I have changed the format of some of the code. In some cases I have stored a value in a local variable and used that variable rather than repeating the name of the value two or more times. In `SOSStart` I have added more functions.

The SOS Files

In this section we will go through each of the files in SOS and describe what they do and how they differ from the book's version of the same code.

SOSData.java

The `SOSData` class contains the data for the operating system. It is a singleton, that is, only one copy of `SOSData` will ever be instantiated. The `SOSProcessDescriptor` class, the `SOSWaitQueueItem` class, and the `SOSDiskRequest` class are defined in their own files since Java requires that classes that are used by more than one other class be defined in their own files.

SOSDiskDriver.java

The `SOSDiskDriver` class implements most of the disk subsystem. The rest is implemented in the `DiskIntHandler` class.

The disk queue is a Java `Vector` instead of the book's `Queue` class so the usage is a little different.

The `DiskBusy` function must interface with the Java virtual machine's disk simulation and so is different from the version I the book. This is also true of `IssueDiskRead` and `IssueDiskWrite`. We still send the commands and parameters to the disk by setting disk registers but these exist in the address space of the physical disk array in the Java virtual machine.

SOSDiskIntHandler.java

The SOSDiskIntHandler class implements the SIMIntHandler interface so it can be called by the disk simulation. Saving the process state and resetting the hardware timer are different due to differences in the Java virtual machine.

SOSDiskRequest.java

The SOSDiskRequest class must be defined in its own file because Java requires this.

SOSMem.java

The `SOSMem` class is not completed yet.

SOSProcessDescriptor.java

The `SOSProcessDescriptor` class defines the process descriptor (naturally) for SOS. The main change here is that the save area is not required because Java saves the state of the thread when we suspend it. We have no way to get to this state using Java code.

SOSProcessManager.java

The `SOSProcessManager` class contains the `CreateProcessSysProc` function and the `Dispatcher` function. These are separated in the book version of SOS.

`CreateProcessSysProc` is different because of differences in the Java virtual machine. The `CreateProcess` function handles the virtual machine process creation. This was not necessary in the book version of SOS. The Java version does not allocate memory for code since this is handled automatically by Java. Space in the physical memory array is allocated.

In `SelectProcessToRun`, the variable `next_proc` has been moved from being a static local variable to being a variable in `sosData`. Other than the addition of tracing, this is the only change to `SelectProcessToRun`.

The `RunProcess` function is totally different because it must interface with the Java virtual machine instead of the CRA-1 virtual machine. Setting the timer and running a process are done completely differently.

SOSProgIntHandler.java

The `SOSProgIntHandler` class implements the `SIMIntHandler` interface. This is difference from the book version of SOS. The change was required because of the change to Java.

SOSStart.java

The `SOSStart` class does system initialization. The interrupt vectors are handled differently in the Java version. The interrupt vector area is kept in the physical memory array as handles to the interrupt handler objects. `SOSStart` initializes the interrupt vector area. I have also restructured the initialization code so that the I/O system and the process system each have their own initialization procedures. This seemed more modular. Process manager initialization requires the construction of a number of Java objects in arrays. The message buffers are in the physical memory array and so are accessed differently as they are linked up. We use Java `Vectors` for queues and they are constructed (and used) differently than the `Queue` class in the book.

Java requires us to keep more handles, like the one to the disk driver that is initialized in `InitializeIOSystem`.

In the Java virtual machine calls to the `Dispatcher` actually return since we are not manipulating actual stacks. This means we must explicitly suspend the startup process after it returns from the dispatcher. The startup thread only does this startup code and then it is no longer used.

SOSSyscallIntHandler.java

The `SOSSyscallIntHandler` class implements the `SIMIntHandler` interface. This is how we cause interrupts in the Java virtual machine. This is required by the stronger typing rules in Java.

The system call argument in the Java virtual machine are handled differently than they were in the CRA-1. The Java virtual machine has no registers and so system call arguments cannot be passed in register. In the Java virtual machine we get them out of the physical memory array. The return value of the system call is also placed in the physical memory array instead of in a register.

Resetting the timer is different in the Java virtual machine.

Process state does not have to be saved in the Java virtual machine.

The message queues are implemented with Java Vectors which are used a bit differently than the Queue class postulated in the book.

Message buffers in the Java version of SOS are kept in the physical memory array so we use `MemoryCopy` instead of calls to transfer data between address spaces. We see this in the `TransferMessage` function and the `MemoryCopy` function.

The `GetMessageBuffer` and `FreeMessageBuffer` functions are included in this class instead of with the global data where the book places them.

SOSTimerIntHandler.java

The `SOSTimerIntHandler` class handles timeout of time slices allocated to processes. The thread representing the user process must be suspended explicitly here.

SOSWaitQueueItem.java

The `SOSWaitQueueItem` class must be defined in its own file because Java requires this. In the book the structure definition was kept with the global data definitions.

Control Flow in Java SOS

It will help you to understand how the system works if you can see how control flows through the system. Let's start with the path of a system call. Each user process has its own thread and it is that thread that will make the system call. This same thread handles the system call in the kernel and then returns. Here is a chart of the control flow.

App*: User Thread*

1. Actions before system call
2. Make system call

    HW*: SystemCall*

    1. Raise priority of this thread
    2. Call SOS system call interrupt handler

        SOS*: SyscallIntHandler*

        1. Save process state (nothing to do in Java SOS)
        2. Reset timer (and record how much was left)
        3. Handle system call (this varies depending on which system call it is)
        4. Call dispatcher

            SOS*: Dispatcher*

            1. Pick process to run
            2. Set timer
            3. Run process

                HW*: RunProcess*

                1. Resume the thread of the process to run
                2. Return (from *RunProcess*)

            4. Return (from *Dispatcher*)

        5. Return (from *SyscallIntHandler*)

    3. If this process is not next then suspend its thread
    4. Return (from *SystemCall*)(either right away to continue this process or later when this process is dispatched again)

3. Actions after system call

Note that each call is returned from normally. There are no abrupt shifts of control in a single thread as there would be in the CRA-1 simulation or in a real machine. The shifts of control are achieved by switching between threads. If this system call causes the current process to changes then action1 in the hardware `RunProcess` function will resume the thread of the new process. The thread of the old process will continue to run. It will return from RunProcess, return from Dispatcher, return from SyscallIntHandler and then, in the hardware SystemCall function the thread will be suspended. If the process making the system call is chosen to run again then the resume in RunProcess will have no affect since the thread is not suspended and the suspend in SystemCall will not be done. The process will then return from the hardware system call and the calling process will resume execution.

Now let us look at the flow of control when there is a timer interrupt. Two threads will be running. The first is the thread of the running process and the second is the thread of the timer. The timer thread will be sleeping. When the sleep ends it will gain control (because it has higher priority than threads for user processes). It will then take the following actions:

***HW**: Timer*

1. Sleep for the duration of the timer interval
2. Wake up
3. Call the timer interrupt handler

   ***SOS**: TimerIntHandler*

   1. Save process state (nothing to do in Java SOS)
   2. Suspend the thread of the current process
   3. Call the dispatcher

      ***SOS**: Dispatcher*

      1. Pick process to run
      2. Set timer
      3. Run process

         ***HW**: RunProcess*

         1. Resume the thread of the process to run
         2. Return (from *RunProcess*)
      4. Return (from *Dispatcher*)
   4. Return (from *TimerIntHandler*)
4. Loop (back to action 1)

The timer is a continuous loop, sleeping for the timer interval and then signaling the timer interrupt. The timer interrupt handler is run using the thread of the timer and it always returns to the timer loop.

Disk interrupt handling is similar to timer interrupt handling. The disk interrupt handler is run using the thread of the disk simulation.

Conclusions

The Java version of SOS has made some compromises to make it run under Java. The hardware simulation is quite different from the CRA-1. There is no process state to save or restore. The memory simulation is rough since only a little of the data memory is simulated. The advantage is that it runs anywhere.

Later versions of the Java SOS will have more extensive memory simulation and be able to simulate paging and virtual memory. They will also emulate the two-processor system from Chapter 6 and the semaphores of chapter 8.