INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

OPERATING SYSTEM SIMULATION (OSS) IN JAVA: THE USER INTERFACE

HONG WEI MAO

A MAJOR REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

AUGUST 1998

Abstract

Operating System Simulation (OSS) in Java: The User Interface

Hong Wei Mao

The OSS system is an operating system simulation tool. The simulated operating system is a multi-process system running on a single processor. The underlying simulated hardware contains CPU, registers, main memory, backing-store and other system resources controlled by semaphores. The simulated operating system itself contains six major components, which are process management, scheduler, memory management, message management, semaphore management and resource management. The OSS system can be used to help computer science students to get a clear understanding of how an operating system works and to study the impact of various algorithms on the performance of an operating system. As the implementing language, Java has outstanding features, such as the multiple threads programming, garbage collection, abstract window toolkit (AWT) and the portability, which are very much appreciated in the development of the OSS system.

Acknowledgments

I would like to express my sincere gratitude to Dr. Peter Grogono, my major report supervisor, for his guidance and valuable insight throughout this research. His support and patience were invaluable in the preparation of this major report.

Table of Contents

List of Figures

1. Introduction

An operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. It provides an environment in which a user can execute programs and it is an essential part of a computer system. Similarly, a course on operating systems is an essential part of a computer science education. An intuitive and interesting learning tool will certainly help students to get a clear understanding of how an operating system works.

The main goals of an operating system are to make a computer system convenient to use and to have the computer hardware used in an efficient manner. With a flexible simulation environment, analyzing the impact of various algorithms on the performance of an operating system becomes an easy task.

Nowadays, almost all operating systems are written in a systems-implementation language or in a higher-level language. This feature has improved the implementation, the maintenance and the portability of an operating system.

Having chosen Java as the implementing language of the OSS system, we are impressed by the capability of Java in the development of a computer software system with its advanced features, such as the multiple threads programming, the garbage collection, the abstract window toolkit (AWT) and the portability.

The rest of this report is organized as follows:

- Section 2, "**Operating System**", describes the role of an operating system.
- Section 3, "**Related Work**", gives a brief account of another OS simulation called ToyOS.
- Section 4, "**Implementation Language - Java**", describes the distinct features of Java.
- Section 5, "**User Interface**", describes what a user can do with the OSS system.
- Section 6, "**Conclusion**", summarizes the key features of the OSS system and proposes the future enhancements.
- "**References**", gives the references used in the development of the OSS system.
- "**Appendix A**", "**User's Manual**", describes how to use the OSS system.
- "**Appendix B**", gives a complete list of ToyOS source codes.
- Other aspects of OSS are described in the report "Operating System Simulation (OSS) In Java: The System Architecture" by Ming Hu [8]. Hu's report contains the source codes of OSS.

2. Operating System

An operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

When creating a system as large and complex as an operating system, the only feasible way to proceed is to partition it into smaller pieces. Then the responsibilities of an operating system are distributed into its subsystems. The major components of most modern operating systems are discussed in the following eight paragraphs.

2.1. Process Management

A program does nothing unless its instructions are executed by a CPU. A process can be thought of as a program in execution, but its definition will broaden as described below.

A process needs certain resources, including CPU time, memory, files and I/O devices, to accomplish its task. These resources are either given to the process when it is created, or allocated to it while it is running. When the process terminates, the operating system will reclaim any reusable resources.

A program by itself is not a process; a program is a passive entity, much like the contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute. A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently, by multiplexing the CPU among them.

The operating system is responsible for the following activities in connection with process management [1]:

- The creation and deletion of both user and system processes.
- The provision of mechanisms for process synchronization.
- The provision of mechanisms for process communication.
- The provision of mechanisms for deadlock handling.

2.2. Main Memory Management

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices. Main memory is generally the only storage device that the CPU is able to address directly. For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute

addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

The operating system is responsible for the following activities in connection with memory management [1]:

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes or which part of a process are to be loaded into memory when memory space becomes available.
- Allocate and de-allocate memory space as needed.

2.3. File Management

Computers can store information on several different types of physical media. Magnetic tape, magnetic disk and optical disk are the most common media. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. The operating system maps files onto physical media, and accesses these files via the storage devices. A directory is an abstraction. Users can use directory structure to create their own subdirectories and to organize their files accordingly. A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

The operating system is responsible for the following activities in connection with file management [1]:

- The creation and deletion of files.
- The creation and deletion of directories.
- The support of primitives for manipulating files and directories.
- The mapping of files onto secondary storage.
- The backup of files on stable (nonvolatile) storage media.

2.4. I/O System Management

Input and output devices include all hardware for interfacing between the computer and the 'real world'. These devices are printer, mouse, keyboard, monitor, touch-screen, light pen, speakers, microphones, temperature sensors, etc. Usually there is a device-driver that acts like a go-between for the CPU and the actual I/O device – simplifying modifications to the system.

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. The I/O system consists of

- A buffer-caching system.
- A general device-driver interface.
- Drivers for specific hardware devices.

Only the device driver knows the peculiarities of the specific device to which it is assigned.

2.5. Secondary-Storage Management

Main memory is too small to accommodate all data and programs, and its data are lost when power is turned off or lost. The computer system must provide secondary storage to back up main memory. Secondary storage is also needed by virtual memory technique, which allows the execution of processes that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Further, it abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. Most modern computer systems use disks as the principal on-line storage medium, for both programs and data. Hence, the proper management of disk storage is of central importance to a computer system.

The operating system is responsible for the following activities in connection with disk management [1]:

- Free-space management.
- Storage allocation.
- Disk scheduling.

2.6. Networking

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory and the processor communication lines, such as high-speed buses or telephone lines. The processors in a distributed system are connected through a communication network.

The operating system is responsible for the following activities in connection with network management:

- Naming and name resolution
- Routing strategies (not on end stations)
- Connection strategies
- Contention

2.7. Protection System

If a system has multiple users and allows multiple concurrent processes, the various processes must be protected from one another's activities. For that purpose, mechanisms are provided to ensure that the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

2.8. Command Interpreter System

An important system program for an operating system is the command interpreter, which is the interface between the user and the operating system. The command statements themselves deal with process creation and management, I/O handling, secondary storage management, main memory management, file-system access, protection and networking.

3. Related Work

A ToyOS developed in Modula-3 was used in the operating systems course (COMP 346/546, Summer 1997) in the Department of Computer Science at Concordia University. The ToyOS simulates a simple multi-programming environment and includes process management, CPU scheduling and memory management. A complete list of ToyOS system source codes can be found in Appendix B.

The ToyOS contains the following major components:

- Module User. This module contains some test procedures, which invokes system calls and the simulation.
- Module SysCall. This module implements some system calls such as Fork, Getpid, and Yield. The system calls use registers to pass values to the operating system and to get the returned values.
- Module OS. This module provides the entry points of the operating system layer.
- Module MyProcess. This module is responsible for process creation, execution, switching and termination.
- Module MyScheduler. This module manages the ready-to-run processes and decides who gets to run next, whenever a scheduling decision has to be made.
- Module MyMemMgr. This module is responsible for allocating a given number of frames to the process and freeing the frames allocated to a process.
- Module MachineOS. This module provides a shortcut to the hardware.

- Module Machine. This module provides the hardware emulation. It includes the following emulation: registers, main memory, backing store, DMA emulation and clock.

We argue that the OSS system is better than the ToyOS system for the following reasons. First, more operating system components and algorithms are implemented in the OSS system. Second, a graphical user interface is implemented in the OSS system. Users can observe some internal operating system data structures through the user interface. Third, a configuration manager is implemented in the OSS system, and this provides users with great convenience of shaping the OSS system in the way they want. Fourth, Java is chosen to be the implementing language for the OSS system. The advantages of using Java are described in detail in the following section.

4. Implementing Language - Java

Java is chosen to be the implementing language. In general, a good simulation implementing language should at least hold the following aspects:

- Providing a mechanism to handle user's interaction with the system and internal system computation simultaneously.
- Providing a development toolkit, which facilitates building application user interface.
- Containing object oriented technologies that simplify software development and maintenance.
- Having a high portability that enables an application to run on different platforms.

Java is a new programming language, with elements from C, C++ and other languages, and with libraries highly tuned for the Internet environment. It draws inspiration from many sources such as Mesa, Modula-3, Lisp, Objective C, C and so on. The overall ability of Java makes it well qualified for implementing a simulation. Some distinct features of Java are discussed in the following six paragraphs.

4.1. Multiple Threads

Threads simply extend the concept from switching between several different programs to switching between several different functions executing simultaneously within a single program. Instead of the costly overhead of saving the state (virtual memory map, file

descriptors, etc.) of an entire process, a low-overhead context switch (saving just a few registers, a stack pointer, the program counter, etc.) within the same address space is done [4].

The Java development environment supports multithreaded programs through the language, the libraries, and the runtime system. Java has a lot of features for threads such as the life cycle of a thread, scheduling threads, grouping threads, and synchronizing threads.

4.2. Garbage Collection

Garbage collection means the automatic reclaiming of memory that is no longer in use. In C language, *malloc()* allocates memory, and *free()* makes it available for reuse. In C++, the *new* and *delete* operators have the same effect. Both of these languages require explicit de-allocation of memory. The programmer has to tell what memory to give back, and when. In practice, this has turned out to be an error-prone task.

Java takes a different approach. Instead of requiring the programmer to take the initiative in freeing memory, the job is given to a runtime component called the garbage collector. It is the job of the garbage collector to sit on top of the memory heap and periodically scan it to determine objects that are not being used any more. It can reclaim that memory and put it back in the free store pool within the program [4].

Taking away the task of memory management from the programmer gives him or her one less thing to worry about, and makes the resulting software more reliable in use. It may take a little longer to run Java programs as compared to programs written in a language like C++ with explicit memory management. On the other hand, it is much quicker to debug and get the program running in the first place.

4.3. The Abstract Window Toolkit (AWT)

The Java AWT provides a set of primitive objects for partitioning the display area and for simple interactive objects such as buttons, text fields, lists etc. We can thus build a user interface quickly.

The AWT is a set of classes and packages, which can be used without concern for platform specific issues. The AWT leaves the actual rendering and behavior of different components to the native windowing systems.

The platform dependent work is done by the *java.awt.peer* classes. An AWT peer is just a wrapper around native code (platform dependent code). This allows application windows to have the same look and feel as the platform windowing environment.

4.4. Applet

Java is a fine general purpose programming language. It can be used to good effect to generate stand-alone executables, just as C, C++, Visual Basic, Pascal or Fortran can. Java offers the additional capability of writing code that can live in a web page and then get downloaded and executed when the web page is browsed [4].

The OSS system is implemented as a stand-alone application and can be transferred to an applet with little difficulty.

4.5. "Almost" Platform Independence

The "write once, run anywhere" slogan is synonymous with the Java programming language. The Java run-time library provides application developers with the ability to write code in one single language and the confidence that the code will execute without modification on virtually any hardware, any operating system, and any application environment. Developers are freed from the costs of porting their applications to a myriad of target platforms and worse, of maintaining those multi-code bases. Powerful cross-platform run-time systems have existed for many other interpreted languages. However, Java is the first to achieve widespread popularity among commercial developers [6].

In general, Java's accomplishments toward platform independence are rather amazing, but not quite perfect. As a shining example, when Sun Microsystems announced the 1.0

release of its Java development environment "Java Workshop", it was touted as "completely written in Java". It turns out that Java Workshop now only runs on Windows and Solaris platforms (not exactly platform independent) [7].

The worst thing about architecture independence is that it keeps you independent of the architecture - meaning that Java needs to assume the lowest common denominator of available resources. Java supports only one mouse button because Macs only have one mouse button. Java can only assume very generic features of the host system: it assumes a system has a CPU, memory, and a graphics subsystem. It also assumes the operating system is multithreaded, which negates the idea of Java on Windows 3.1. Joysticks, special keypads, and the like can not be accessed from Java [7].

To get system-specific, you need to write some C code and interface it with Java. The problem is that if your program is even 1% C code, you lose Java's platform independence and its solid security model [7].

4.6. Slow Performance

Java is slow. The reason for Java's speed can be simplified in one word: abstraction. To OO/high-level language designers, abstraction is the goal. To the computer, any abstraction puts the code another step away from what it understands. It is also the computer's job to decode those steps down to its machine language so it can actually run the program [7].

Machine language coding is the lowest level we can code in. Above that, we can abstract machine code to assembly code. Then the computer geniuses invented the high-level language abstraction. Java puts the object-orientation abstraction on top of this. Finally, Java adds the platform-independence abstraction. That's a lot of abstraction and it takes a lot of compilers, optimizers, and smart run-time environments to remove them all, to get our program back down to machine code so it can actually run.

How could Java's performance be improved? The compiler companies came out with their just-in-time (JIT) compilers. These programs (which are closer to assemblers than compilers) convert Java's stack-based intermediate representation into native machine code immediately prior to execution on your machine. That way, the Java program actually runs as a real executable. JIT compilers do speed up Java programs. Java performance could someday approach C or C++ performance [7].

5. User Interface

In this section, we describe the user interface, which is what user can do with the system.

The user interface for OSS consists of 4 windows, which are the main window, the configuration window, the adding new process window, the help window and the selecting file window.

5.1. Main Window

As shown in Figure 1, the main window includes the following components:

5.1.1. The Title Bar

The title bar contains the name of the system and the current execution mode. The two execution modes are:

- Interactive - During the execution, the user can request to create a new process.
- Non-interactive - Execution is based on the current system configurations and the user can not interfere with the progress of the simulation.

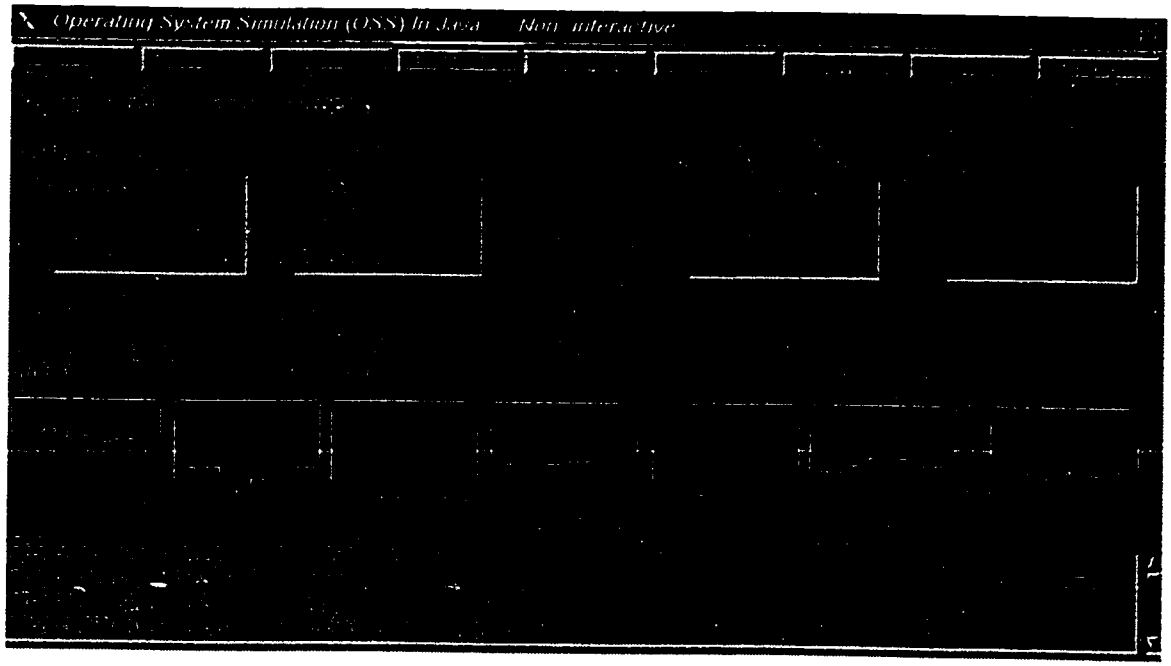

Figure 1. Main Window

5.1.2. The Control Panel

The control panel contains several buttons. They are:

- The "Add" button. Click to add a new user process.
- The "Start" button. Click to start a simulation.
- The "Config" button. Click to specify the system configurations.
- The "Freeze" button. Click to pause the simulation.
- The "Slower" button. Click to make the simulation slower.
- The "Faster" button. Click to make the simulation faster.
- The "End" button. Click to stop the current simulation.
- The "Help" button. Click to see the explanations on various questions the user may have while using the system.
- The "Shutdown" button. Click to shut the system down.

5.1.3. The Area Monitoring the Simulation States

Some important operating system data structures are shown in this area. They are the current running process, the ready queue, the sleep queue, the message waiting queue and the semaphore waiting queue. These system data structures are quite important because the changing on them reflect the system configuration and job mix. The effect of showing them on the user interface is like taking snapshots of the system status.

5.1.4. The Area Monitoring the Simulation Performance

There are seven different canvases corresponding to seven criteria that measure the system performance in this area. The seven criteria are: the CPU utilization, the throughput, the turnaround time, the response time, the waiting time, the page fault rate, and the memory availability.

5.1.5. The Area Tracing the Simulation History

This area contains a list showing the history of simulation execution.

5.2. Configuration Window

As shown in Figure 2, the configuration window includes the following components:

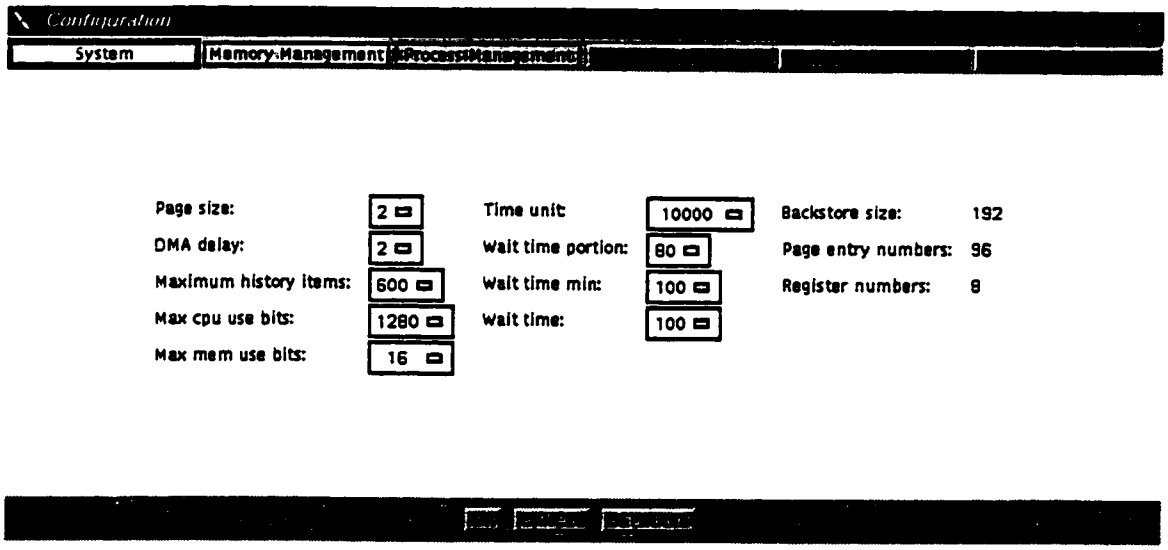

Figure 2. Configuration Window

5.2.1. The Title Bar

It contains the name of the window.

5.2.2. The Catalogue Panel

There are six different buttons corresponding to six catalogues of configuration information. The six catalogues are: the system, the memory management, the process management, the IPC (Inter Process Communication), the simulation mode, and the

output results. When the user clicks on one of these buttons, the corresponding configuration information will be displayed in the window. For example, if the user clicks the *Memory Management* button, the configuration information related to memory management will be displayed in the window.

5.2.3. The Area Displaying Configuration Information

This is where the configuration information is displayed and where the user sets the preferable configuration values. For example, when setting configuration information about memory management, the user can reset memory size, the maximum number of memory access per process, and page replacement algorithm.

5.2.4. The Control Panel

There are three buttons in the control panel. They are:

- The "OK" button. Click to store the modified configuration information and return to the main window.
- The "CANCEL" button. Click to ignore the modified configuration information and return to the main window.
- The "DEFAULTS" button. Click to set the configuration information in the current window to default values.

5.3. Adding New Process Window

The adding new process window is shown in Figure 3. Users can use this window to request OSS to add a new user process in the interactive execution mode. The information that users must provide consists of the priority level of new user process and whether the new user process contains the memory, semaphore and message operations.

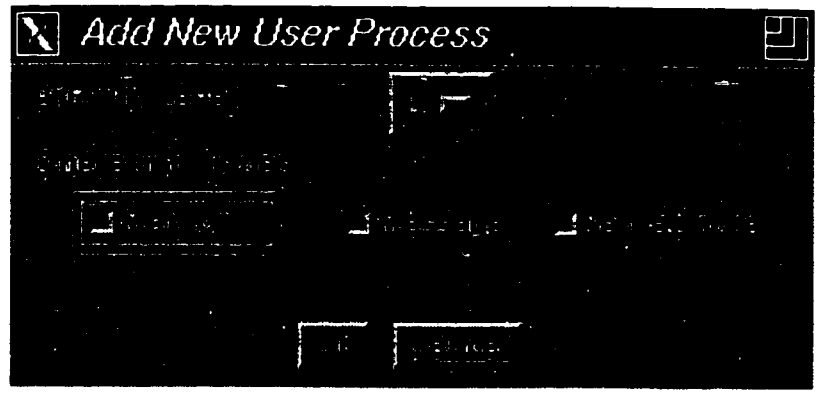

Figure 3. Adding New Process Window

5.4. Help Window

The help window is shown in Figure 4. The purpose of the help window is to provide explanations on questions the user may raise while using the OSS system. There are several menu items from which the user can choose to display the corresponding topics.

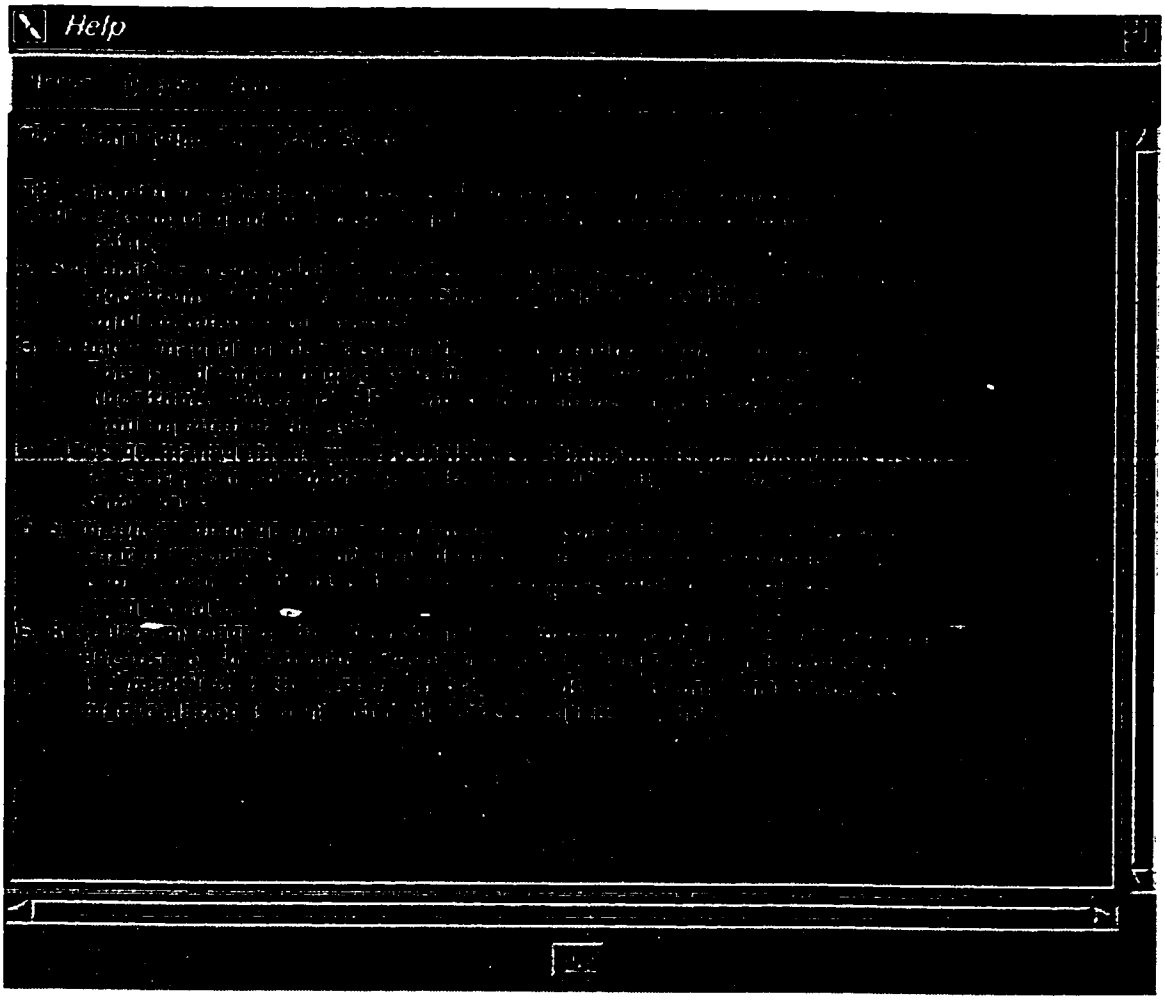

Figure 4. Help Window

5.5. Selecting File Window

Figure 5 shows the selecting file window. The selecting file window provides the convenience for the user to traverse the directory tree and find a particular file in which to save the simulation results.

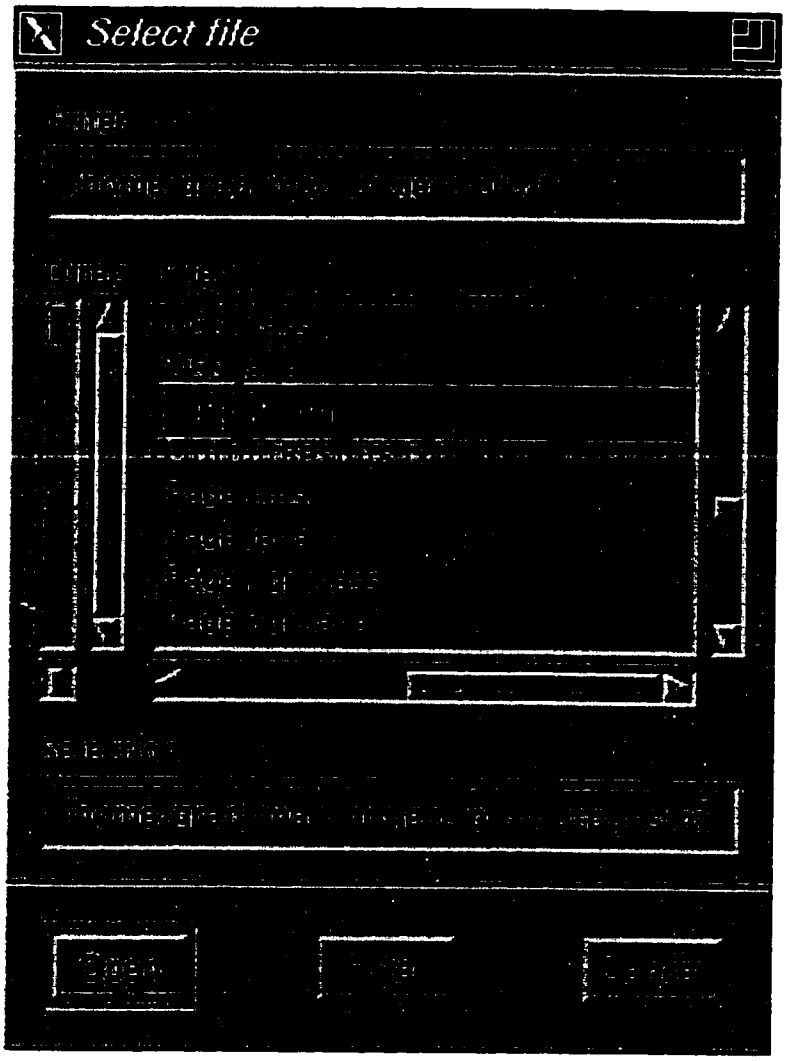

Figure 5. Selecting File Window

6. Conclusion

6.1 Summary

The OSS system consists of two major components:

- The user interface;
- The underlying simulated operating system.

The underlying simulated operating system is a multi-process system running on a single processor and it runs on top of the simulated hardware.

The simulated hardware contains CPU, registers, main memory, backstore and the system resources controlled by the semaphores.

The simulated operating system itself contains six major components. They are: process management, scheduler, memory management, message management, semaphore management and resource management.

The current implementation of the OSS system has following features:

- Two execution modes: interactive and non-interactive.
- Two page replacement algorithms: FIFO and second chance.
- Two scheduling algorithms: round robin and priority.
- Flexibility of specifying system configurations.
- Interfering in the system execution speed.

- Providing help.
- Monitoring the simulation states.
- Monitoring the simulation performance.
- Multiple operations for a user process.
- Output of the simulation results to a file.
- Java inheritance – easy to add more operating system algorithms.
- Java portability – platform independent.

6.2 Future Work

The following enhancement in the future are proposed:

- More scheduling and page replacement algorithms.
- Animations on the representation of system internal states and performance criteria.
- Virtual terminals that can take users' commands.
- Communications among processes running on different hosts.

References

1. Abraham Silberschatz, Peter B. Galvin, "***Operating System Concepts***", Forth Edition, Addison-Wesley, 1994.
2. A ***ToyOS*** program implemented in Modula-3 and used by operating systems course (COMP 346/546, Summer 1997) in the Department of Computer Science at Concordia University.
3. Gary Cornell, Cay S. Horstmann, "***Core Java***", Second Edition, Sun Microsystems, Inc., 1997.
4. Peter van der Linden, "***Just Java***", Second Edition, SunSoft Press and Prentice Hall, 1997.
5. http://www.javasoft.com:80/products/jdk/1.1/docs/api/a-index.html
6. Sandeep Singbal, Binb Nguyen, "***The Java Factor***", Communications of the ACM, June 1998-Volume 41, Number 6.
7. Paul Tyma, "***Why Are We Using Java Again?***", Communications of the ACM, June 1998-Volume 41, Number 6.
8. Ming Hu, "Operating System Simulation (OSS) in Java: The System Architecture", Major Report, Department of Computer Science, Concordia University, August 1998.

Appendix A – User's Manual

A.1. Starting the OSS System

Enter the command:

java OSS

A.2. Setting the System Configurations

Click on the "Config" button in the main window. The configuration window will pop up.

A.2.1. Setting Configuration Information about System

Click on the "System" button in the configuration window to display the system information, which can be specified by the user.

A.2.2. Setting Configuration Information about Memory Management

Click on the "Memory Management" button in the configuration window to display memory management information, which can be specified by the user.

A.2.3. Setting Configuration Information about Process Management

Click on the "Process Management" button in the configuration window to display the process management information, which can be specified by the user.

A.2.4. Setting Configuration Information about IPC

Click on the "IPC" button in the configuration window to display the inter-process communication information, which can be specified by the user.

A.2.5. Setting Configuration Information about Simulation Mode

Click on the "Simulation Mode" button in the configuration window to display simulation mode information, which can be specified by the user.

A.2.6. Setting Configuration Information about Output Results

Click on the "Output Results" button in the configuration window to display the results output information, which can be specified by the user.

The user may specify if the simulation results are stored in a file. If so, the user also needs to specify the name and contents of the file. A selecting file window, which is initiated by clicking on the "Browse" button, may provide the user with convenience in finding the right file name.

A.2.7. Making the Newly Specified Configuration Information to Take Effect

Click on the "OK" button in the configuration window.

A.2.8. Ignoring the Newly Specified Configuration Information

Click on the "CANCEL" button in the configuration window.

A.2.9. Setting Configuration Information Displayed in the Current Window into Default Values

Click on the "DEFAULTS" button in the configuration window.

A.3. Starting the Simulation

Click on the "Start" button in the main window.

A.4. Adding A New User Process (Interactive Mode)

Click on the "Add" button in the main window. The adding new process window will pop up.

A.5. Pausing the Simulation

Click on the "Freeze" button in the main window. Afterwards, this button will read as "Un-Freeze".

A.6. Resuming the Frozen Simulation

Click on the "Un-Freeze" button in the main window. Afterwards, this button will read as "Freeze".

A.7. Speeding Up the Simulation

Click on the "Faster" button in the main window. When the speed of the simulation reaches the specified limit, the "Faster" button will be disabled.

A.8. Slowing Down the Simulation

Click on the "Slower" button in the main window.

A.9. Ending the Current Simulation (Interactive Mode)

Click on the "End" button in the main window.

A.10. Requesting Help

Click on the "Help" button in the main window.

A.11. Shutting Down the System

Click on the "Shutdown" button in the main window.

Appendix B – ToyOS Source Code

B.1. Machine.i3

```
INTERFACE Machine;

IMPORT Word ;

CONST

  NREGS  = 8 ;      (* number of registers *)
  MEMSIZE  = 128 ;  (* memory size in words *)
  ADDRSPACESIZE = 64; (* size of address space *)
  PAGESIZE = 4 ;    (* number of words in a page *)
  NUMPAGES = 16;    (* number of pages per process *)


VAR

  (* general purpose registers. The registers may be used as variables. *)
  (* in user programs *)
  Registers : ARRAY[1..NREGS] OF Word.T ;



  (* ---------- User Mode Operations ---------- *)

PROCEDURE Load(vaddr: Word.T ; reg: Word.T);
  (* memory to register transfer *)

PROCEDURE Store(vaddr: Word.T ; reg: Word.T);
  (* register to memory transfer *)

PROCEDURE StoreI(vaddr: Word.T ; value: Word.T);
  (* Immediate to memory transfer *)

PROCEDURE GenSwInterrupt() ;
  (* generate a software interrupt *)

PROCEDURE SetDebug(flag: BOOLEAN) ;
  (* Allows debugging of the Machine Module. *)

END Machine.
```

B.2. Machine.m3

```
MODULE Machine EXPORTS Machine, MachineOS ;
(* This Module provides the Hardware (Machine) Emulation *)

(*

  This module supports the following hardware emulation.

  Registers:
  ---------

  User visible registers (numbered 1..8) are available to programs
  like any variable, and therefore, user programs can perform arithmetic
  using them.

  Two special registers are used: PTBR -- the page table base register,
  and FaultingPage -- to store the page number that gets a page fault.
  The FaultingPage is needed by OS to find out where the page fault
  occured, and to take appropriate actions based on that.

  Main Memory and Backing Store:
  ------------------------------

  Both Main memory and backing are viewed as arrays, and both are viewed
  as paged. The Main memory is accessible only through the Load and Store
  operations which are simulated here. The Backing Store is accessed
  through DMA emulation, explained below.

  DMA Emulation:
  --------------

  The DMA emulation provides a way to transfer data between the main
  memory and backing store. One DMA transfer request moves one page worth
  of data.

  The DMA emulation makes the I/O asynchronous (as in the real world).
  The OS requests a DMA transfer using the call DMASetup(). At that
  time, the request is simply stored within this module, and a counter
  start. When the counter goes down to 0, the operation is completed,
  and the OS is interrupted. This effectively simulates a time delay
  for the DMA operation.

  Clock:
  -----

  The module also simulates a clock, which can be used to drive round
  robin scheduling. The clock is also used to model delay for DMA as
  identified above. A clock interrupt is generated whenever the clock
  counter goes down to zero.

*)


IMPORT IO, Fmt, Text;
IMPORT Word ;
IMPORT OS ; (* needed to call OS Interrupt Handler *)


CONST

 DMADELAY = 10 ; (* Assume that each DMA operation requires 10 ticks *)

VAR

 (* Main Memory Emulation *)
 MainMem   : ARRAY [0..MEMSIZE-1] OF Word.T;
```

```
 (* Backing Store Emulation *)
 BackStore  : ARRAY [0..BACKSTORESIZE-1] OF Word.T;

 (* Clock Interrupt Counter *)
 ClockCounter : Word.T := 0;

 (* DMA Emulation *)

 (* how much more ticks before the current DMA finishes *)
 DMACounter : Word.T := 0 ;

 (* the currently pending DMA request *)
 DMAReq    : DMAReqT ;

 (* whether there is a DMA request active *)
 DMAPending : BOOLEAN ;

 debugflag : BOOLEAN ;

(*---------------------- User Operations ---------------------- *)

PROCEDURE Load(vaddr:Word.T ; reg:Word.T) =
 (* load the value at virtual address vaddr into register reg *)
 VAR
  page, offset, frame, paddr: Word.T;
 BEGIN

  Debug("Load: vaddr " & Fmt.Int(vaddr) & " reg " & Fmt.Int(reg) & "\n") ;

  WHILE ( ( vaddr >= ADDRSPACESIZE) OR (reg > NREGS)) DO
   GenInterrupt(ILLEGALADDRESS) ;
   (* The OS should terminate this process/thread, that is we should *)
   (* not return here. *)
  END;

  Tick() ;  (* advance time *)

  (* do the address translation using page table *)
  page   := vaddr DIV PAGESIZE ;
  offset := vaddr MOD PAGESIZE ;

  (* Check if the page is in memory or not *)
  WHILE PageTable[page+PTBR].valid = FALSE DO
   (* Store the faulting page number somewhere *)
   Debug("Page Fault on (page, PTBR) (" & Fmt.Int(page) & "," &  Fmt.Int(PTBR) & ")\n") ;
   FaultingPage := page ;
   GenInterrupt(PAGEFAULT) ;
  END;

  (* At this point, the page should be in the main memory. *)
  frame := PageTable[page+PTBR].memframe;
  paddr := frame * PAGESIZE + offset;
  Registers[reg] := MainMem[paddr];
  PageTable[page+PTBR].ref := TRUE ;

 END Load;

PROCEDURE Store(vaddr:Word.T ; reg:Word.T) =
 (* store the value in reg into the virtual address vaddr *)
VAR
 VAR
  page, offset, frame, paddr: Word.T;
 BEGIN

  Debug("Store: vaddr " & Fmt.Int(vaddr) & " reg " & Fmt.Int(reg) & "\n") ;

  WHILE ( ( vaddr >= ADDRSPACESIZE) OR (reg > NREGS)) DO
   GenInterrupt(ILLEGALADDRESS) ;
   (* The OS should terminate this process/thread, that is we should *)
```

```
    (* not return here. *)
   END;

   Tick() ; (* advance time *)

   (* do the address translation using page table *)
   page   := vaddr DIV PAGESIZE ;
   offset := vaddr MOD PAGESIZE ;

   (* Check if the page is in memory or not *)
   WHILE PageTable[page+PTBR].valid = FALSE DO
    (* Store the faulting page number somewhere *)
    Debug("Page Fault on (page, PTBR) (" & Fmt.Int(page) & "," &  Fmt.Int(PTBR) & ")\n") ;
    FaultingPage := page ;
    GenInterrupt(PAGEFAULT) ;
   END;

   (* At this point, the page should be in the main memory. *)
   frame := PageTable[page+PTBR].memframe;
   paddr := frame * PAGESIZE + offset;
   MainMem[paddr] := Registers[reg];

   PageTable[page+PTBR].dirty := TRUE ;
   PageTable[page+PTBR].ref := TRUE ;
  END Store;

PROCEDURE StoreI(vaddr:Word.T ; value:Word.T) =
 (* store the value into the virtual address vaddr *)
VAR
 VAR
  page, offset, frame, paddr: Word.T;
 BEGIN

  Debug("StoreI: vaddr " & Fmt.Int(vaddr)  & "\n") ;

  WHILE  ( vaddr >= ADDRSPACESIZE)  DO
   GenInterrupt(ILLEGALADDRESS) ;
   (* The OS should terminate this process/thread, that is we should *)
   (* not return here. *)
  END;

  Tick() ; (* advance time *)

  (* do the address translation using page table *)
  page   := vaddr DIV PAGESIZE ;
  offset := vaddr MOD PAGESIZE ;

  (* Check if the page is in memory or not *)
  WHILE PageTable[page+PTBR].valid = FALSE DO
   (* Store the faulting page number somewhere *)
   Debug("Page Fault on (page, PTBR) (" & Fmt.Int(page) & "," &  Fmt.Int(PTBR) & ")\n") ;
   FaultingPage := page ;
   GenInterrupt(PAGEFAULT) ;
  END;

  (* At this point, the page should be in the main memory. *)
  frame := PageTable[page+PTBR].memframe;
  paddr := frame * PAGESIZE + offset;
  MainMem[paddr] := value;

  PageTable[page+PTBR].dirty := TRUE ;
  PageTable[page+PTBR].ref := TRUE ;
 END StoreI;


PROCEDURE GenSwInterrupt() =
 BEGIN
  GenInterrupt(SOFTWARE) ;
 END GenSwInterrupt;
```

```
(*---------- Priviliged Operations ----------*)

PROCEDURE Idle() =
 BEGIN
   Tick() ;  (* advance time by one unit *)
 END Idle;

PROCEDURE SetClockCounter(value: Word.T) =
 BEGIN
  ClockCounter := value ;
 END SetClockCounter;

 (* Assume that DMASetup is called only when there is no pending DMA  *)
 (* requests. If there were any, they will be lost. That is, there is *)
 (* no queuing of DMA requests in the hardware. Any queueing must be  *)
 (* done by the operating system. *)

PROCEDURE DMASetup(req: DMAReqT) =
 (* Set up a DMA request and return - the data transfer is done later. *)
 BEGIN
  DMAPending := TRUE ;
  DMAReq := req ;            (* store the request locally *)
  DMACounter := DMADELAY;    (* delay before the IO completes *)
 END DMASetup;

 (*---------- Internal Operations ----------*)


PROCEDURE GenInterrupt(num: Word.T) =
 BEGIN
  OS.InterruptHandler(num) ;
 END GenInterrupt;


PROCEDURE Tick() =
 BEGIN

  (* decrement counters *)
  DEC(ClockCounter) ;
  IF DMAPending THEN DEC(DMACounter) END ;

  (* first generate clock interrupt, if needed *)
  IF (ClockCounter <= 0 ) THEN
   GenInterrupt(CLOCK) ;
  END ;

  (* now generate DMA interrupt, if needed *)
  IF DMAPending AND (DMACounter <= 0) THEN
   (* finish the pending DMA operation *)
   IF DMAReq.direction = MEMTOBS THEN
    MemtoBS(DMAReq.memframe, DMAReq.bsframe)
   ELSE
    BStoMem(DMAReq.bsframe, DMAReq.memframe)
   END ;
   DMAPending := FALSE;
   (* generate the interupt *)
   GenInterrupt(DMA) ;
  END ;

  END Tick;

PROCEDURE BStoMem(bsframe, memframe: Word.T) =
 (* Copy a page from backing store to main memory frame *)
 VAR
  memaddr, bsaddr : Word.T ;
 BEGIN
```

```
    memaddr := memframe * PAGESIZE ;
    bsaddr  := bsframe * PAGESIZE ;

    FOR i := 0 TO PAGESIZE - 1 DO
      MainMem[memaddr] := BackStore[bsaddr] ;
      INC(memaddr) ;
      INC(bsaddr) ;
    END ;
  END BStoMem;

PROCEDURE MemtoBS(memframe, bsframe: Word.T ) =
  (* Copy a page from backing store to main memory frame *)
  VAR
    memaddr, bsaddr : Word.T ;
  BEGIN

    memaddr := memframe * PAGESIZE ;
    bsaddr  := bsframe * PAGESIZE ;

    FOR i := 0 TO PAGESIZE - 1 DO
      BackStore[bsaddr] := MainMem[memaddr] ;
      INC(memaddr) ;
      INC(bsaddr) ;
    END ;

  END MemtoBS;

PROCEDURE SetDebug(flag: BOOLEAN) =
  BEGIN
    debugflag := flag ;
  END SetDebug;

PROCEDURE Debug(str: TEXT) =
  BEGIN
    IF debugflag THEN
      IO.Put("Machine: " & str & "\n");
    END;
  END Debug;

BEGIN

  (* Put Zero in all registers, and main memory locations *)
  FOR i:=1 TO NREGS DO
    Registers[i] := 0;
  END;

  FOR i:=0 TO MEMSIZE-1 DO
    MainMem[i] := 0;
  END;

  FOR i:=0 TO BACKSTORESIZE-1 DO
    BackStore[i] := 0;
  END;

END Machine.
```

B.3. MachineOS.i3

```
INTERFACE MachineOS ;
(* This interface provides a shortcut to hardware (registers, memory) to *)
(* the operating system code; thereby simplifying our overall design. It *)
(* also contains any specific operations that require priviliged mode.  *)

IMPORT Word;
IMPORT Machine;

CONST

 (* for identifying interrupts *)

 CLOCK = 1 ;
 DMA  = 2 ;
 PAGEFAULT = 3 ;
 ILLEGALADDRESS = 4;
 SOFTWARE = 5 ;

 (* for DMA Transfer direction *)
 MEMTOBS = 1 ;
 BSTOMEM = 2 ;

 BACKSTORESIZE = 2048 ; (* holds 32 processes *)

TYPE

 (* The machine emulation dictates the type of page table entries. *)
 (* This corresponds to the real life situation, since the memory  *)
 (* management unit uses page table entries during address translation *)

 (* page table entry type *)
 PageEntryT = RECORD
  memframe : Word.T ;   (* frame number in main memory *)
  valid   : BOOLEAN ; (* TRUE if mapping is valid *)
  dirty   : BOOLEAN ; (* TRUE if modified *)
  ref     : BOOLEAN ; (* TRUE if referenced *)
 END ;

 (* This provides the type of requests handled for DMA transfer requests  *)
 (* Again, this is provided by the hardware emulation. Any additional info *)
 (* needed by the OS must be maintained in a different structure, which   *)
 (* can include a DMAReqT as a field. *)

 DMAReqT = RECORD
  memframe  : Word.T ;  (* frame in memory *)
  bsframe   : Word.T ;  (* frame in backing store *)
  direction : INTEGER; (* direction of transfer *)
 END ;

VAR

 (* special purpose registers *)
 PTBR: Word.T ; (* page table base register *)
 FaultingPage : Word.T ; (* used to store the faulting page *)

 (* The Page Table is a large array, which should contain the page tables  *)
 (* for all processes. The OS should set the PTBR appropriately to provide *)
 (* the starting point of the page table for the currently running process *)
 (* That is, PageTable[PTBR] corresponds to page 0. *)

 PageTable : ARRAY[0..511] OF PageEntryT ; (* allows 32 processes *)

 (* ---------- Kernel Mode Operations ---------- *)
```

```
PROCEDURE Idle() ;
  (* Idle the cpu for one clock tick. *)

PROCEDURE SetClockCounter(value: INTEGER) ;
  (* Set Clock Counter. The counter counts down to zero, and then generates *)
  (* a clock interrupt. It must be reset after each quantum. *)

PROCEDURE DMASetup(req: DMAReqT) ;
  (* Initiate a DMA transfer. The call returns immediately. When the DMA  *)
  (* terminates (after some time delay), a DMA interrupt is generated. In *)
  (* other words, this simulates an asynchronous I/O operation.          *)
  (* Must not be called if another DMA operation is in progress.         *)

END MachineOS.
```

B.4. Main.m3

```
MODULE Main;

IMPORT Text ;
IMPORT IO, Fmt ;
IMPORT OS, Machine, MemMgr, MyProcess, MyScheduler ;
IMPORT SysCall;
IMPORT User ;


BEGIN


 (* Set appropriate debugging flags *)

 Machine.SetDebug(TRUE) ;
 MemMgr.SetDebug(TRUE) ;
 MyProcess.SetDebug(TRUE) ;
 MyScheduler.SetDebug(TRUE) ;
 OS.SetDebug(TRUE);
 SysCall.SetDebug(TRUE) ;

 (* Do test runs *)

 FOR i := 1 TO 2 DO
  IO.Put("Starting OS Simulation\n") ;

  (* set up the initial set of processes *)
  User.Test(i) ;

  (* start the simulation *)
  OS.RunSimulation() ;

  (* all processes have terminated *)

  IO.Put("Simulation Over\n") ;
 END ;


END Main.
```

B.5. MemMgr.i3

```
INTERFACE MemMgr;
IMPORT Word ;
FROM MyProcess IMPORT PCB ;


PROCEDURE Alloc(proc: REF PCB; nFrames: Word.T) : INTEGER ;
  (* Allocate the given number of frames to the process. *)
  (* Assumes that these correspond to page 0, 1, 2, etc. *)
  (* Called only once per process in the beginning.     *)


PROCEDURE Free(proc: REF PCB);
  (* free the frames allocated to a process *)


PROCEDURE SetDebug(flag: BOOLEAN) ;


END MemMgr.
```

B.6. MemMgr.m3

```
MODULE MemMgr;

IMPORT IO, Fmt ;
IMPORT Word ;
IMPORT MyList;

FROM MyProcess IMPORT PCB ;
IMPORT Machine ;
FROM MachineOS IMPORT PageTable ;
IMPORT SysCall ;

TYPE

 FrameT = RECORD
  free : BOOLEAN ;
  memframe : Word.T ;
  vpage : Word.T ;
  proc  : REF PCB ;
 END ;

VAR

 FreeFrameList : MyList.T ;

 debugflag : BOOLEAN ;

 frame : REF FrameT ;

(* Allocate the given number of frames to the process. *)
(* Assumes that these correspond to page 0, 1, 2, etc. *)
(* Called only once per process in the beginning.     *)
PROCEDURE Alloc(proc: REF PCB; nFrames: Word.T) : INTEGER =
 VAR frame : REF FrameT ;
 BEGIN

  (* first check if there are enough free frames *)
  (* if not return an error *)
  IF MyList.Length(FreeFrameList) < nFrames THEN
   Debug("Not enough frames (free, need) (" & Fmt.Int(MyList.Length(FreeFrameList)) & "," & Fmt.Int(nFrames) & ")") ;
   RETURN SysCall.ERROR ;
  END;

  (* okay -- enough free frames exist. *)

  (* create frame list for process *)
  proc^.FrameList := MyList.Create() ;

  (* set up PTBR *)

  proc^.PTBR := proc^.id * Machine.NUMPAGES ;

  Debug("Allocating " & Fmt.Int(nFrames) ) ;
  (* get the frames and allocate them to the process *)
  FOR i:= 0 TO nFrames-1 DO
   (* get a free frame *)
   frame := MyList.RemoveFirst(FreeFrameList) ;
   frame^.free := FALSE ; (* mark it allocated *)
   frame^.vpage := i ;  (* mark the page it corresponds to *)
   frame^.proc := proc ;  (* to identify the process using it *)
   PageTable[proc^.PTBR+i].valid := TRUE ;
   PageTable[proc^.PTBR+i].memframe := frame^.memframe ;
   MyList.InsertLast(frame, proc^.FrameList) ; (* add to process list *)
  END;

  RETURN 1 ;
```

```
 END Alloc;

(* free the frames allocated to a process *)
PROCEDURE Free(proc: REF PCB) =
 VAR frame : REF FrameT ;
 BEGIN

  FOR i := 1 TO MyList.Length(proc^.FrameList) DO
   frame := MyList.RemoveFirst(proc^.FrameList) ;
   frame^.free := TRUE ;
   MyList.InsertLast(frame,FreeFrameList) ;
  END ;

  FOR i:= 0 TO Machine.NUMPAGES-1 DO
   PageTable[proc^.PTBR+i].valid := FALSE ;
  END ;

 END Free;


PROCEDURE SetDebug(flag: BOOLEAN) =
 BEGIN
  debugflag := flag ;
 END SetDebug;

PROCEDURE Debug(str: TEXT) =
 BEGIN
  IF debugflag THEN
   IO.Put("MemMgr: " & str & "\n");
  END;
 END Debug;

BEGIN


 FreeFrameList := MyList.Create() ;

 FOR i := 0 TO (Machine.MEMSIZE DIV Machine.PAGESIZE - 1) DO
  frame := NEW(REF FrameT) ;
  frame^.memframe := i ;
  frame^.free := TRUE ;
  MyList.InsertLast(frame, FreeFrameList) ;
 END;

 (* Initialize Page Table Array *)
 FOR i := 0 TO 511 DO
  PageTable[i].valid := FALSE ;
  PageTable[i].dirty := FALSE ;
  PageTable[i].ref  := FALSE ;
 END;

END MemMgr.
```

B.7. MyList.i3

```
INTERFACE MyList;

(* Implements a Generic List of Pointer-Type Items. This allows us to  *)
(* use the same code in a number of situations. The List Implementation *)
(* does not need to know, and does not touch, the internal structure of *)
(* the Items. It is the responsibility of the application to define the *)
(* internal structure of the items.                                    *)


TYPE

  (* Hide the List Type from the Clients of List Interface *)
  (* That is, let the List Type be an unspecified sub-type of REFANY *)
  (* The real type structure is given in the implementation module *)

  T <: REFANY ;

PROCEDURE Create () : T  ;
  (* Create and initialize an Empty List. Return a handle to the List *)

PROCEDURE IsEmpty (list: T) : BOOLEAN ;
  (* Return TRUE if the List is Empty, FALSE otherwise *)

PROCEDURE Length (list: T) : CARDINAL ;
  (* Return the number of items in the list  *)

PROCEDURE First (list: T) : REFANY ;
  (* Return the first item in the list, but do not delete it. *)
  (* Return NIL if the list is empty *)

PROCEDURE InsertLast (item: REFANY;  VAR list: T);
  (* Insert a new item at the tail of the List. *)

PROCEDURE InsertFirst (item: REFANY;  VAR list: T);
  (* Insert a new item at the head of the List. *)

PROCEDURE RemoveFirst (VAR list: T) : REFANY ;
  (* Remove an element from the head of the List, and return the item *)

PROCEDURE RemoveItem (item: REFANY; VAR list: T) : REFANY ;
  (* Remove an element from the List, if it exists, and return the item. *)
  (* Return NIL if the element does not exist in the list. *)

END MyList.
```

B.8. MyList.m3

```
MODULE MyList;

TYPE

 ListEntry = RECORD
  item : REFANY ;        (* the real data *)
  next : REF ListEntry ;  (* pointer to next entry *)
 END ;


 (* Reveal the real structure of the List. *)

REVEAL
 T = BRANDED "List" REF RECORD
  head, tail : REF ListEntry ; (* pointers to head and tail *)
  length : CARDINAL ;        (* number of entries *)
 END;


PROCEDURE Create ():T =
  (* Create and initialize an Empty List. Return a handle to the List *)
 BEGIN
  RETURN NEW (T, head := NIL, tail := NIL, length := 0) ;
 END Create;


PROCEDURE IsEmpty (list: T) : BOOLEAN =
 (* Return TRUE if the List is Empty, FALSE otherwise *)
 BEGIN
  RETURN (list.length = 0)
 END IsEmpty;

PROCEDURE Length (list: T) : CARDINAL =
 (* Return the number of items in the list  *)
 BEGIN
  RETURN list.length
 END Length;

PROCEDURE First (list: T) : REFANY =
 (* Return the first item in the list, but do not delete it *)
 (* Return NIL if the list is empty *)
 BEGIN
  RETURN list.head^.item
 END First;

PROCEDURE InsertLast (item: REFANY;  VAR list: T)=
 (* Insert a new item at the tail of the List. *)
 VAR entry : REF ListEntry;
 BEGIN
  entry := NEW(REF ListEntry) ;
  entry^.item := item ;
  entry^.next := NIL ;

  IF IsEmpty(list) THEN
   list.head := entry ;
   list.tail := entry ;
  ELSE
   list.tail^.next := entry ;
   list.tail := entry;
  END;
  INC(list.length) ;
 END InsertLast;

PROCEDURE InsertFirst (item: REFANY;  VAR list: T)=
 (* Insert a new item at the head of the List. *)
```

```
  VAR entry : REF ListEntry;
  BEGIN
   entry := NEW(REF ListEntry) ;
   entry^.item := item ;
   entry^.next := NIL ;

   IF IsEmpty(list) THEN
    list.head := entry ;
    list.tail := entry ;
   ELSE
    entry^.next := list.head ;
    list.head := entry ;
   END;
   INC(list.length) ;
 END InsertFirst;

 PROCEDURE RemoveFirst (VAR list: T) : REFANY =
  (* Remove an element from the head of the List, and return the item *)
  VAR
   item : REFANY;
  BEGIN
   IF IsEmpty(list) THEN
    RETURN(NIL)
   ELSE
    item := list.head^.item ;
    list.head := list.head^.next ;
    IF (list.head = NIL) THEN list.tail := NIL END ;
    DEC(list.length) ;
    RETURN item ;
   END ;
  END RemoveFirst;

 PROCEDURE RemoveItem (item: REFANY; VAR list: T) : REFANY =
  (* Remove an element from the List, if it exists, and return the item. *)
  (* Return NIL if the element does not exist in the list. *)
  VAR
   p, q : REF ListEntry ;
   done : BOOLEAN ;
  BEGIN
   p := list.head ;
   q := NIL ;
   done := FALSE ;

   WHILE (p # NIL) AND (NOT done) DO
    IF p^.item = item THEN
     (* Found it; delete the item from the list *)
     done := TRUE
    ELSE
     (* advance to next entry *)
     q := p ;
     p := p^.next ;
    END ;
   END ;

   IF p = NIL THEN  (* Not Found *)
    RETURN NIL ;
   ELSE
    (* Remove p and adjust list *)
    IF q # NIL THEN
     q^.next := p^.next
    ELSE (* p must be the head of the list *)
     list.head := p^.next
    END ;
    DEC(list.length) ;
    IF p = list.tail THEN
     list.tail := q ;
    END ;
    (*RETURN p ;*)
    RETURN p^.item;  (*97.05.12*)
   END
```

```
  END RemoveItem;

BEGIN
END MyList.
```

B.9. MyProcess.i3

```
INTERFACE MyProcess;

(*
  Interface for Process Abstraction. The Interface is used internally
  within the OS modules.
*)

IMPORT Word ;
IMPORT Thread;
IMPORT MyList;
IMPORT SysCall;
IMPORT Machine;


TYPE

 (* Process States *)

 ProcState = {Running, Ready, Terminated, Blocked} ;

 (* The Process Conurol Block *)

 PCB = RECORD
  id     : INTEGER ;          (* process identifier *)
  ppid   : INTEGER ;          (* parent process id *)
  free   : BOOLEAN;           (* Free Status of PCB  *)
  state  : ProcState;         (* Process State *)
  thread : Thread.T;          (* Handle on the thread *)
  schedcv : Thread.Condition;   (* Scheduling Condition *)
  Registers : ARRAY[1..Machine.NREGS] OF Word.T ;
                        (* Register copies *)
  PTBR   : Word.T;            (* Copy of Page Table Base Reg. *)
  FrameList : MyList.T ;
 END;

VAR

 idleproc : REF PCB ; (* PCB of the idle process *)
 nprocess : INTEGER;  (* bumber of processes still in system *)

PROCEDURE Dispatch(newproc: REF PCB) ;
 (* Dispatch a new process, saving the context for the current one *)

PROCEDURE Exit() ;
 (* Terminate the current process *)

PROCEDURE Fork(proc: SysCall.UserProgramT; arg1,arg2: Word.T): Word.T ;
 (* Fork a new process to execute 'proc(arg1,arg2)' *)
 (* Control stays with the parent. *)

PROCEDURE GetPid() : INTEGER ;
 (* get the process id *)

PROCEDURE Yield() ;
 (* yield cpu to another runnable process, is there is one *)

PROCEDURE GetCurProc() : REF PCB ;
 (* Get the pcb of the current process *)

PROCEDURE SetDebug(flag: BOOLEAN) ;

END MyProcess.
```

B.10. MyProcess.m3

```
MODULE MyProcess ;

(*
  Process Management Module. Implements the Internal Interface MyProcess.

  Implements the code for process creation, execution, switching, and
  termination.

  Scheduling Control:
  -------------------

  Each process runs on its own Modula-3 thread -- this makes things like
  doing context switch relatively straightforward. In order to control
  the scheduling of processes, we make sure that only one thread is
  runnable at a time. This leaves Modula-3 run-time system with no choice
  but to schedule the thread (our process) that we want. We use conditions
  to achieve this effect -- each process has a condition variable (stored
  in its process control block). When we do a context switch -- say from
  process A to B -- then we first signal B, and then wait on A's condition
  (see procedure Dispatch). This makes B runnable, and A blocked (from
  Modula-3 perspective), and therefore the cpu is given to B. To summarize,
  only the thread corresponding to the currently running process (in our
  simulation world) is runnable as far as Modula-3 run time system is
  concerned; all other threads are blocked on their respective conditions.
  The actual scheduling decisions are taken in the MyScheduler module.

  A tricky situation occurs when a new process is created. Lets say A is
  running and Forks B. When we fork the thread for B, it is runnable and
  there is no way to start it suspended. At this time, the invariant
  mentioned above is violated since we have two runnable threads. The
  solition is quite simple. When B starts, it first blocks itself on its
  condition. A, on the other hand, after it forks B, waits for B to block
  itself -- this avoids race conditions. Then, it inserts B into the ready
  queue.

  Process Mechanics:
  ------------------

  On a Fork, a new process is created by (1) allocating a process control
  block, (2) allocating memory for it, and by (3) forking a new thread.
  Each thread runs the procedure that we want to run -- which is passes
  as an argument to fork. The Closure object for Thread.Fork runs the
  procedure ProgWrapper(), which is a wrapper function within which we
  run the actual procedure. This allows (1) blocking of thread on start-up,
  as mentioned above, and (2) cleanup on termination.

  SystemCalls:
  ------------

  The module implements functions for three system calls: Fork, GetPid,
  and Yield. It also provides the Dispatch function, which implements the
  context switching code, and is called by the Scheduler module.

*)

IMPORT IO, Fmt, Text ;
IMPORT Thread, MyList ;
IMPORT Word ;
IMPORT SysCall;
IMPORT Machine, MachineOS ;
IMPORT MyScheduler;
IMPORT MemMgr ;
FROM Machine IMPORT Registers ;
FROM MachineOS IMPORT PTBR ;
```

```
CONST

 MAXPROCESS = 10 ;


TYPE

 (*
   The Closure Object Type is inherited from Thread.Closure which has a
   predefined method apply, bound to Nil. Here, we bind the method to
   procedure ProgWrapper(), which serves as a wrapper procedure within which
   we execute the procedure desired. We also add three fields to the
   Closure Object: (1) pid: the process id, (2) proc: the procedure to
   be executed, and (3) arg1/arg2: the arguments to the procedures proc.

   The creation of a new process results in the creation of a new Closure
   object, which is passed to the thread.
 *)

 Closure = Thread.Closure OBJECT
  pid  : INTEGER ;          (* process id *)
  proc : SysCall.UserProgramT;   (* procedure to execute *)
  arg1, arg2 : Word.T:           (* arguments to the procedure proc *)
 OVERRIDES
  apply := ProgWrapper;          (* bind method apply to procedure ProgWrapper() *)
 END;

VAR

 (* global variables used for managing processes *)

 ProcTable : ARRAY[0..MAXPROCESS] OF REF PCB ; (* process table *)
 nextpid : INTEGER := 0;  (* Starting point to search for the free PID *)
 curpid  : INTEGER;       (* PID of the currently executing process *)

 debugflag  : BOOLEAN ; (* debugging flag *)


 (*
  The following variables are used to make the parent wait for the
  child to start, and then block. Until then, it is not safe to do
  anything else, since we have two concurrent runnable threads.
 *)

 createflag: BOOLEAN ;
 createcond : Thread.Condition := NEW(Thread.Condition) ;

 mutex : Thread.Mutex := NEW(Thread.Mutex);
 (* to protect createflag. Also used for Dispatching. *)


(*---------- System Calls ---------- *)

(*
  Create a new process, and execute the supplied procedure "proc" in
  it. The parameter "args" is used to pass arguments to the procedure.
  The call returns the process identifier of the newly created process
  on success, and returns SysCall.ERROR on failure.
*)

PROCEDURE Fork(proc: SysCall.UserProgramT; arg1,arg2: Word.T): INTEGER =
 VAR
  pid: INTEGER ;
 BEGIN


  (* Find a free PCB first *)
```

```
    pid := nextpid ;
    WHILE (NOT ProcTable[pid]^.free) DO
     pid := (pid + 1) MOD MAXPROCESS;
     IF (pid = nextpid) THEN
      (* no free slot, return SysCall.ERROR. *)
      RETURN SysCall.ERROR ;
     END ;
    END ;

    (* initialize the PCB *)
    WITH p = ProcTable[pid] DO

     (* first check if enough memory is available *)
     (* in the absence of demand paging support, we must allocate *)
     (* main memory frames a priori. *)
     IF MemMgr.Alloc(p, Machine.NUMPAGES) = SysCall.ERROR THEN
      (* not enough memory, Fork must fail now *)
      RETURN SysCall.ERROR ;
     END ;

     p^.free  := FALSE ;
     p^.ppid  := curpid ;
     p^.schedcv := NEW(Thread.Condition) ;

     (*
       Create a new Closure Object, initialize it, and pass it to
       Thread.Fork, which in turn creates a new thread and returns
       a handle to the thread. Immediately after this a new thread
       exists concurrently, which will execute the procedure ProgWrapper().
     *)
     createflag := TRUE ;
     Debug("Fork: Id = " & Fmt.Int(pid) & " Ppid " & Fmt.Int(curpid));
     p^.thread := Thread.Fork(NEW(Closure,pid := pid,proc := proc,arg1:= arg1,arg2:=arg2));
    END ;
    nextpid := (pid + 1) MOD MAXPROCESS;  (* the next starting point *)


    (* Wait for the child to block itself, i.e., set createflag to false *)

    LOCK mutex DO
     (* put while, just in case; an if should be enough  *)
     WHILE createflag DO
      (* release lock, and go back to checking the local variable *)
      Thread.Wait(mutex, createcond) ;
     END  ;
    END ;

    (* enter the child in the ready queue *)
    MyScheduler.MakeReady(ProcTable[pid]) ;

    (* there is one more process in the system *)
    INC(nprocess) ;

    (* return back to the parent. note: no re-scheduling done here. *)
    RETURN pid;
  END Fork;

(* Return the process identifier of the process.  *)

PROCEDURE GetPid() : INTEGER =
  BEGIN
   RETURN curpid ;
  END GetPid;


(* Yield the CPU to another process, if possible *)

PROCEDURE Yield() =
  BEGIN
    Debug("Yielding " & Fmt.Int(curpid)) ;
```

```
  MyScheduler.Run() ;
 END Yield;


 (*---------- Support functions for process execution ---------- *)

 (*
  This is the procedure bound to method apply in Closure object. Each
  newly created thread (process in our simulation) executes this procedure.
  By passing a procedure and its arguments to ProgWrapper(), we are able to
  execute different things within each thread (process).
 *)

PROCEDURE ProgWrapper (self: Closure): REFANY =
 VAR pid: INTEGER;
   proc: SysCall.UserProgramT;
   arg1, arg2: Word.T ;
 BEGIN

  (*
    when a process starts here, it is still not under our scheduling
    control. we need to block the process here, and inform the parent
    who then enters the process in the ready queue.
  *)

  pid := self.pid ;
  proc := self.proc ;
  arg1 := self.arg1 ;
  arg2 := self.arg2 ;

  LOCK mutex DO
   createflag := FALSE ; (* Let the parent know that I am blocked *)
   Thread.Signal(createcond) ;
   (* Wait on the condition until the scheduler picks me to run *)
   Thread.Wait (mutex, ProcTable[pid]^.schedcv);
  END;

  proc(arg1,arg2) ;  (* Do the real thing *)
  Exit();      (* Terminate the process -- clean up  *)

  RETURN NIL ;
 END ProgWrapper;

(* Terminate current process, schedule and dispatch another process *)

PROCEDURE Exit() =

 BEGIN

  WITH p = ProcTable[curpid] DO
   Debug("Exit: Id " & Fmt.Int(curpid) & "\n") ;
   p.state := ProcState.Terminated;
   (* do not free the PCB yet, we will do it in when dispatching *)
  END ;


  DEC(nprocess) ; (* one less process *)
  MemMgr.Free(ProcTable[curpid]) ; (* free memory used *)
  MyScheduler.Run() ;  (* give control to another process *)
  (* The thread should never return here *)
  <* ASSERT FALSE *>
 END Exit;


PROCEDURE Dispatch(newproc: REF PCB) =
 (* Dispatch the new process, blocking the current one *)
 VAR
  oldproc : REF PCB ;
 BEGIN
```

```
    (* Simply return if newpid is the same as curpid *)

    oldproc := ProcTable[curpid] ;
    IF newproc = oldproc THEN
     RETURN;
    END;

    (* Force a Context Switch *)
    newproc^.state := ProcState.Running ;
    curpid := newproc^.id ;

    (* save state of old process, and restore state of new process *)
    FOR i:= 1 TO Machine.NREGS DO
       oldproc^.Registers[i] := Registers[i];
       Registers[i]  := newproc^.Registers[i];
    END;
    oldproc^.PTBR := PTBR ;
    PTBR := newproc^.PTBR ;

    Debug("Context Switch: from " & Fmt.Int(oldproc^.id) &
     " to " & Fmt.Int(newproc^.id)) ;
    LOCK mutex DO
     (* Signal the newprocess to run *)
     Thread.Signal (newproc^.schedcv) ;
     (* Block this process *)
     Thread.Wait(mutex,oldproc^.schedcv);
    END
  END Dispatch;

 PROCEDURE GetCurProc() : REF PCB =
  BEGIN
   RETURN ProcTable[curpid] ;
  END GetCurProc;


  (*----------------- Auxiliary Support Functions ----------------*)

 PROCEDURE SetDebug(flag: BOOLEAN) =
  BEGIN
   debugflag := flag ;
  END SetDebug;

 PROCEDURE Debug(str: TEXT) =
  BEGIN
   IF debugflag THEN
    IO.Put("Process: " & str & "\n");
   END;
  END Debug;


BEGIN


 (* Initialize the Process Table *)
 FOR i:= 0 TO MAXPROCESS  DO
   ProcTable[i] := NEW(REF PCB); (* pre-allocated storage for PCBs *)
   ProcTable[i]^.free := TRUE ;
   ProcTable[i]^.id := i ; (* to get the pid from the PCB entry *)
 END ;

 idleproc := ProcTable[MAXPROCESS] ;
 idleproc^.schedcv := NEW(Thread.Condition) ;
 curpid := MAXPROCESS ;

 debugflag := TRUE ;  (* Keep the debugging messages on. *)


END MyProcess.
```

B.11. MyScheduler.i3

```
INTERFACE MyScheduler;

FROM MyProcess IMPORT PCB ;

(*
  Scheduler Interface. Used Internally by Other OS Modules. The
  Scheduler manages the Ready (to Run) Processes, and decides
  who gets to run, whenever a scheduling decision has to be made.
*)

PROCEDURE Run() ;
(* Main Entry Point for the Scheduler. This invokes the scheduler,
  which will schedule and dispatch a new thread if required. *)

PROCEDURE MakeReady(proc : REF PCB ) ;
(* Make a Process Ready to Run *)

PROCEDURE SetDebug(flag: BOOLEAN) ;

END MyScheduler.
```

B.12. MyScheduler.m3

```
MODULE MyScheduler;

IMPORT IO, Fmt, Text ;

IMPORT MyList ;
IMPORT Machine;
IMPORT OS ;

FROM MyProcess IMPORT PCB, ProcState, idleproc ;
IMPORT MyProcess ;

VAR

 readyQ : MyList.T ;

 debugflag : BOOLEAN ;

(* ---------- Exported Procedures ---------- *)


PROCEDURE Run() =
 VAR
  newproc : REF PCB ;
 BEGIN

  (* find the next process to run, and then dispatch it *)

  newproc := Pick();

  Debug("New Process: " & Fmt.Int(newproc^.id) ) ;
  MyProcess.Dispatch(newproc);

 END Run;


PROCEDURE MakeReady(proc: REF PCB) =
 BEGIN

  (* Make the process state Ready and insert it in the ready queue *)

  Debug("Process " & Fmt.Int(proc^.id) & " made ready") ;
  proc^.state := ProcState.Ready ;
  MyList.InsertLast(proc,readyQ) ;

 END MakeReady;


(* ---------- Internal Procedures ---------- *)


PROCEDURE Pick (): REF PCB =
 (* Pick the Next Process to Run *)
 VAR
  curproc, newproc  : REF PCB ;
 BEGIN

  curproc := MyProcess.GetCurProc() ;

  IF MyList.IsEmpty(readyQ) THEN
    CASE curproc^.state OF
    | ProcState.Terminated, ProcState.Blocked=>
     (* no runnable process, pick idle process to run *)
     newproc := idleproc ;
     (* free this pcb *)
    | ProcState.Running =>
```

```
        (* continue with the same process *)
        newproc := curproc ;
      ELSE
       (* Should not come here -- program bug *)
       <* ASSERT FALSE *>
      END
    ELSE  (* ready queue is not empty *)
     (* first find the new process to run *)
     newproc := MyList.RemoveFirst(readyQ);
     IF (curproc^.state =  ProcState.Running)
      AND (curproc # idleproc) THEN
      (* Enter the current process on the Ready Queue, *)
      (* unless its the idle process. *)
      curproc^.state := ProcState.Ready;
      MyList.InsertLast(curproc, readyQ);
     END ;
    END ;

    (* Free PCB if the current process has terminated *)
    IF (curproc^.state = ProcState.Terminated) THEN
     curproc^.free := TRUE ;
    END ;
    RETURN newproc ;

  END Pick;


(*------------------ Auxiliary Support Functions --------------------*)


PROCEDURE SetDebug(flag: BOOLEAN) =
  BEGIN
    debugflag := flag ;
  END SetDebug;

PROCEDURE Debug(str: TEXT) =
  BEGIN
    IF debugflag THEN
      IO.Put("Scheduler: " & str & "\n");
    END;
  END Debug;

(*---------------------- Module Initialization ---------------------*)

BEGIN

  readyQ := MyList.Create() ;
  debugflag := TRUE ;

END MyScheduler.
```

B.13. OS.i3

```
INTERFACE OS;

(* This is the OS interface; all OS code execution must go through    *)
(* this interface.                                                    *)

IMPORT Word ;


PROCEDURE InterruptHandler(num: Word.T) RAISES ANY ;
  (* The OS Interrupt Handler. *)

PROCEDURE RunSimulation() ;
  (* Start the OS simulation. Must have created some process before. *)
  (* The call returns when all processes have terminated *)

PROCEDURE SetDebug(flag: BOOLEAN) ;
  (* Allows debugging of the OS Module. *)

END OS.
```

B.14. OS.m3

```
UNSAFE MODULE OS;

IMPORT IO, Fmt, Scan, Text ;

IMPORT MyProcess, MyScheduler ;
IMPORT SysCall ;
IMPORT Machine, MachineOS ;
IMPORT Word ;

FROM MyProcess IMPORT PCB ;
FROM Machine IMPORT Registers ;


CONST

 QUANTUM = 5 ;

VAR
 debugflag : BOOLEAN ;

PROCEDURE InterruptHandler(num: Word.T) =
 BEGIN

  CASE num OF
  | MachineOS.CLOCK  => ClockInterruptHandler() ;
  | MachineOS.DMA  => DMAInterruptHandler() ;
  | MachineOS.PAGEFAULT  => PageFaultHandler() ;
  | MachineOS.ILLEGALADDRESS  => IllegalAddressHandler() ;
  | MachineOS.SOFTWARE => SwInterruptHandler() ;
  ELSE
   IO.Put("Unknown Interrupt" & Fmt.Int(num) & "\n") ;
   IO.Put("Crashing....") ;
   <* ASSERT FALSE *>
  END ;
 END InterruptHandler;

PROCEDURE SwInterruptHandler() =
 VAR curproc : REF PCB ;
 BEGIN

  Debug("In Software Interrupt Handler\n") ;

  CASE Registers[1] OF
  | SysCall.FORK =>
   Registers[8] := MyProcess.Fork(LOOPHOLE(Registers[2],SysCall.UserProgramT), Registers[3], Registers[4]) ;
  | SysCall.GETPID =>
   Registers[8] := MyProcess.GetPid() ;
  | SysCall.YIELD =>
   MyProcess.Yield() ;
  ELSE
   IO.Put("Illegal System Call\n") ;
   curproc := MyProcess.GetCurProc() ;
   IO.Put("Crashing Process " & Fmt.Int(curproc^.id) & "\n") ;
   MyProcess.Exit() ;
  END ;

 END SwInterruptHandler;

PROCEDURE ClockInterruptHandler() =
 BEGIN

  Debug("In Clock Interrupt Handler\n") ;
  (* Reset the clock counter *)
  MachineOS.SetClockCounter(QUANTUM) ;
```

```
  END ClockInterruptHandler;

 PROCEDURE IllegalAddressHandler() =
  BEGIN

   Debug("In Illegal Address Interrupt Handler") ;
   (* Terminate the current process *)
   MyProcess.Exit() ;

  END IllegalAddressHandler;

 PROCEDURE PageFaultHandler() =
  BEGIN
   Debug("In Page Fault Handler\n") ;

   MyProcess.Exit() ;
  END PageFaultHandler;

 PROCEDURE DMAInterruptHandler() =
  BEGIN

   Debug("In DMA Interrupt Handler\n") ;

  END DMAInterruptHandler;

 PROCEDURE RunSimulation() =
  BEGIN
   (* set the clock quantum *)
   MachineOS.SetClockCounter(QUANTUM) ;
   (* become the idle process *)

   WHILE TRUE DO
    MyProcess.Yield() ;
    MachineOS.Idle() ;
    IF (MyProcess.nprocess = 0) THEN
     RETURN ;
    END;
   END;

  END RunSimulation;

 PROCEDURE SetDebug(flag: BOOLEAN) =
  BEGIN
   debugflag := flag ;
  END SetDebug;

 PROCEDURE Debug(str: TEXT) =
  BEGIN
   IF debugflag THEN
    IO.Put("OS:" & str);
   END;
  END Debug;

BEGIN


END OS.
```

B.15. SysCall.i3

```
INTERFACE SysCall;

(* This is the Interface provided by the OS to user programs. The      *)
(* interface consists of the system calls, as well as data types       *)
(* and constants needed by user programs.                              *)

IMPORT Word ;

CONST

  ERROR = 0 ;          (* Return Value for Error *)

  (* System Call Numbers *)
  FORK = 1 ;
  GETPID = 2 ;
  YIELD  = 3 ;
TYPE

  (* Procedure type for user programs.  *)
  (* To keep it simple, we only allow two Word.T arguments to be passed *)
  (* to the procedure. *)
  UserProgramT = PROCEDURE(arg1, arg2: Word.T) ;

(* ------------------ System Calls ----------------------------- *)

PROCEDURE Fork(proc: UserProgramT; arg1, arg2: Word.T): Word.T ;

  (* Fork a new process to execute 'proc(arg1,arg2)' *)
  (* Control stays with the parent. returns ERROR on failure,  *)
  (* and the pid of the child on success. *)

PROCEDURE GetPid() : Word.T ;

  (* Return the process identifier of the process making the call *)

PROCEDURE Yield();

  (*
    Yield the CPU to some other runnable process, if possible.
    The call is used for cooperative multi-tasking.
  *)

PROCEDURE SetDebug(flag: BOOLEAN) ;
  (* allow debugging of SysCall module *)

END SysCall.
```

B.16. SysCall.m3

```
UNSAFE MODULE SysCall ;

IMPORT IO, Fmt, Text ;
IMPORT Machine ;
IMPORT Word ;
FROM Machine IMPORT Registers ;

VAR

 debugflag : BOOLEAN ;
 (*
   Create a new process, and execute the supplied procedure "proc" in
   it. The parameter "args" is used to pass arguments to the procedure.
   The call returns the process identifier of the newly created process
   on success, and returns ERROR on failure.
 *)


(* The System calls use registers to pass values to the operating system  *)
(* and to get return values. The general convention is that Register 8 is *)
(* used to get back the return value, and the registers 1, 2, 3, etc. are *)
(* used to pass the system call number and the arguments to the call. The *)
(* program must save the contents of the registers prior to making the    *)
(* system call, as they get overwritten. *)

(* The calls in this module are merely wrapper functions to make the life *)
(* of a programmer easy. *)

PROCEDURE Fork(proc: UserProgramT; arg1, arg2: Word.T): Word.T =
 VAR
   saveregs : ARRAY[1..Machine.NREGS] OF Word.T ;
   retvalue : Word.T ;
 BEGIN

   FOR i := 1 TO Machine.NREGS DO
     saveregs[i] := Registers[i] ;
   END ;
   Registers[1] := FORK ;
   Registers[2] := LOOPHOLE(proc, Word.T) ;
   Registers[3] := arg1 ;
   Registers[4] := arg2 ;

   (* Generate a software interrupt which will pass control *)
   (* to the operating system. *)

   Machine.GenSwInterrupt() ;

   (*
     When we get back to this point, the system call has already
     been handled. However, we must return the appropriate values
     to the process.
   *)

   retvalue := Registers[8] ;

   FOR i := 1 TO Machine.NREGS DO
     Registers[i] := saveregs[i] ;
   END ;

   RETURN retvalue ;

END Fork;
```

```
 (* Return the process identifier of the process making the call *)

PROCEDURE GetPid() : Word.T =
 VAR
  retvalue : Word.T ;
  saveregs : ARRAY[1..Machine.NREGS] OF Word.T ;
 BEGIN


  FOR i := 1 TO Machine.NREGS DO
   saveregs[i] := Registers[i] ;
  END ;

  Registers[1] := GETPID ;

  Machine.GenSwInterrupt() ;

  retvalue := Registers[8] ;
  FOR i := 1 TO Machine.NREGS DO
   Registers[i] := saveregs[i] ;
  END ;

  RETURN retvalue ;

 END GetPid;

 (*
  Yield the CPU to some other process, if there is one. The call is used
  for cooperative multi-tasking.
 *)

PROCEDURE Yield() =
 VAR
  saveregs : ARRAY[1..Machine.NREGS] OF Word.T ;
 BEGIN


  FOR i := 1 TO Machine.NREGS DO
   saveregs[i] := Registers[i] ;
  END ;

  Registers[1] := YIELD ;
  Machine.GenSwInterrupt() ;
  FOR i := 1 TO Machine.NREGS DO
   Registers[i] := saveregs[i] ;
  END ;

 END Yield;

PROCEDURE SetDebug(flag: BOOLEAN) =
 BEGIN
  debugflag := flag ;
 END SetDebug;

PROCEDURE Debug(str: TEXT) =
 BEGIN
  IF debugflag THEN
   IO.Put("SysCall: " & str & "\n");
  END;
 END Debug;


BEGIN
END SysCall.
```

B.17. User.i3

```
INTERFACE User ;

PROCEDURE Test(i: INTEGER) ;

END User.
```

B.18. User.m3

```
(* This Module Contains the User Programs used for testing purposes *)

MODULE User;

IMPORT Text, IO, Scan, Fmt, Rd ;
IMPORT Word ;

IMPORT SysCall;

FROM Machine IMPORT Registers;
FROM Machine IMPORT Load, Store ;

PROCEDURE Test(i: INTEGER) =
 BEGIN

  CASE i OF
  | 1 => Test1() ;
  | 2 => Test2() ;
  END ;

 END Test;

PROCEDURE Test1() =
 BEGIN
  EVAL SysCall.Fork(Dummy2,5,0);
  EVAL SysCall.Fork(Dummy2,10,0);
 END Test1;


PROCEDURE Test2() =
 BEGIN
  EVAL  SysCall.Fork(Dummy,10, 0);
  EVAL SysCall.Fork(Dummy,20, 0);
 END Test2;


PROCEDURE Dummy(nwords, notused: Word.T) =
 BEGIN

  (* The dummy program accesses the memory in the range  *)
  (* [0..nwords-1] in succession storing a number, and  *)
  (* then in reverse, getting the numbers, and computing *)
  (* their sum at the same time. *)

  FOR i := 0 TO nwords-1 DO
   Registers[1] := i ;
   Store(i,1) ; (* Memory[i] := Registers[1] *)
   SysCall.Yield() ;
  END ;

  Registers[2] := 0 ;   (* to keep the running sum *)
  FOR i := 0 TO nwords-1 DO
   Load(i,1) ; (* Registers[1] := Memory[i]  *)
   Registers[2] := Registers[2] + Registers[1] ;
  END ;

  Registers[3] := SysCall.GetPid() ;
  IO.Put("User Process " & Fmt.Int(Registers[3]) & " terminating\n") ;
  IO.Put("The sum is " & Fmt.Int(Registers[2]) & "\n") ;
 END Dummy;


PROCEDURE Dummy2(niter, notused: Word.T) =
 VAR pid : Word.T ;
 BEGIN
```

```
    Registers[1] := 1 ;
    Registers[2] := 2 ;
    pid := SysCall.GetPid() ;
    FOR i := 1 TO niter DO
      Registers[8] := Registers[1] + Registers[2] ;
      IO.Put("Process " & Fmt.Int(pid) & ": The value is register 8 is " & Fmt.Int(Registers[8]) & "\n") ;
      Registers[1] := Registers[2] + 1 ;
      SysCall.Yield() ;
      Registers[2] := Registers[1] + 1 ;
      IO.Put("Process " & Fmt.Int(pid) & ": The value is register 8 is " & Fmt.Int(Registers[8]) & "\n") ;
    END ;
  END Dummy2;
BEGIN
END User.
```