A UNIX Concurrent I/O Simulator

Steven Robbins
Department of Computer Science
University of Texas at San Antonio
srobbins@cs.utsa.edu

ABSTRACT

This paper describes a simulator that allows users to explore concurrent I/O in UNIX. UNIX I/O provides an interesting example of how a shared variable, in this case the file offset, can be affected by concurrent access. The examples given can run on the simulator or a real UNIX-like system such as Linux, Solaris for Mac OS X. The simulator can run programs written by the user and display pictorially the relationship among various data structures involved in I/O, including the process file descriptor table, the system open file table, the inodes, and the data stored on disk. The user can run the program slowly, or step forward or back through the program to examine the data structures in detail. The simulator supports the creation of both child processes and threads as well as open, close, read, write, wait, join and detach instructions. The simulator is freely available for download. It can be also be used directly from a browser without the need for installation.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*

General Terms

Concurrent I/O

Keywords

operating systems, systems programming

1. INTRODUCTION

Teaching concurrency in undergraduate operating systems courses is always a challenge. Most of the early examples illustrating concurrency given in books involve access to shared variables. Unfortunately, unless students are prepared to write programs involving shared memory or threads, it is difficult to come up with realistic examples that the students can experiment with.

One example that does not require advanced programming techniques involves concurrent I/O. When a UNIX process forks a child after a file is open, the parent and child share the file offset and care must be taken when both processes do I/O to this file.

This paper describes a simulator that allows users to explore the mechanism and consequences of concurrent I/O. The simulator supports both read and write instructions. The read instructions read into memory buffers which are assumed to be sufficiently large arrays. The simulator assumes that users want to load a buffer from the contents of a disk file. In C, a call to read is considered successful if any number (greater than 0) of bytes is read, even if this number is less than the requested number. A read, even from an ordinary file, is not guaranteed to always read the number of bytes requested, even if there are sufficient bytes in the file to satisfy the request. More often, these partial reads occur when reading from a pipe or network, but since programs should be written in a device-independent manner, we cannot assume that all read system calls return the number of bytes requested. While Java has a **readFully** method for some input streams, C does not. Therefore, it is usually up to the programmer to perform read operations in a loop until all desired bytes are read. If all of the bytes must be read before any are processed, this requires multiple reads, each appending to the end of a buffer. We do this by keeping track of the number of bytes already read (the variable **total** in the simulator) and passing to the read function a pointer to the position for the next read. Then **total** must be incremented by the number of bytes read to prepare for the next read operation. This leads to a read instruction in the form:

```
total += read(fd, buf+total, n);
```

To simplify the types of programs handled by the simulator, we assume that all read instructions return the number of bytes requested or the number of bytes remaining in the file, whichever is less. However, the format of the read instruction allows us to simulate the type of code that would not make this assumption.

While writing to a pipe or FIFO is atomic when the number of bytes written is small enough, writing to another device, such as a file or network, is not. Also, a write may return fewer bytes than requested for a number of reasons. The write instructions handled by the simulator looks like:

```
write(fd,"ABCDEFG",n);
```

where the second argument can be any constant string. This will always write the number of bytes requested (**n**, unlike in a real system) and will write garbage into the file if the string is too short (just as in a real system).

The rest of the paper is organized as follows. Section two reviews the I/O mechanism in UNIX. Section 3 describes the basic operation of the simulator. In section 4 we look at using the simulator to explore concurrency among processes. Section 5 deals with concurrency among threads. Section 6 discusses non-atomic I/O. Section 7 discusses using the simulator for classroom demonstrations and assignments and discusses availability.

SIGCSE '06 March 1-5, Houston, Texas, USA

2. UNIX I/O

In traditional UNIX systems, at least three tables are involved when I/O is performed [1]. The *file descriptor table* (FDT) (at least conceptually) resides in each process and can be thought of as an array of pointers indexed by an integer file descriptor. This FDT is sometimes referred to as the *process-specific open file table* [3] or the *table of open files (per process)* [5] and contains pointers to entries in a kernel structure. This kernel structure is sometimes called the *file structure table* [3, 5], the *open file table* [2], the *open file description* [7], or just the *file table* [1, 6]. We will call it the *system file table* (SFT) so that it is clear that it resides in the kernel. This table contains one entry for each open file. Each time *open* is called by a process, a new entry is created. The entries of the SFT contain the file offset, a count of the number of FDT entries pointing to it (so the entry can be freed when it is no longer needed) and a pointer to an in-memory copy of the inode for this file. We assume that these copies of the inodes are kept in a table which we will call the *in-memory inode table*. The information in the inode that we are concerned about here includes the file permissions and the location (on disk) of the data blocks of the file. Our files are all small so we assume that each file contains only a single block.

When a fork is executed in UNIX, the child gets a copy of the parent's file descriptor table. Corresponding entries in the parent and child FDTs point to the same entry in the SFT. SFT entries contain the file offset used by read and write operations, so the parent and the child share the file offset. They can each manipulate this shared resource using read and write operations, and the consequences of this sharing can be immediately apparent. Two processes that share a SFT entry for reading from the file will read from different parts of the file. Alternatively, if each process calls open after the fork occurs, two opens are executed and two separate entries in the SFT are created. The processes use different entries in SFT and therefore each has its own file offset. The first read from each process reads from the start of the file.

3. THE SIMULATOR

This section describes the simplest operation of the simulator. More details can be found in the users guide [4].

Figure 1 shows the initial simulator startup window. The simulator starts by reading a configuration file. Among other things, the configuration file contains the names of the programs that the simulator can access as well as the names and contents of the input files. The user space of Figure 1 contains the program:

```
fork();
fd = open("infile",O_RDONLY);
total += read(fd,buf+total,2);
total += read(fd,buf+total,2);
close(fd);
```

which uses one input file containing **abcdefghijklmnop**.

The screen is divided into sections. The top part of the screen shows a pictorial representation of the computer system. The bottom part has a collection of clickable buttons and other controls that modify the simulator properties and run the simulation. The top part of the screen is divided into three sections. These sections, from left to right are the **User Space** which shows the processes and threads, the **System Space** which shows the system open file table (SFT), the in-memory inodes and the list of processes, and the **Disk Space** which shows the disk blocks.

The **User Space** is divided horizontally into sections for each process. Figure 2 shows the simulator after the main process has executed a **fork** instruction. There are now two processes. Each user process area contains a box representing the process variables, a box containing the program, additional boxes contain the code for threads that have been created (there are no additional threads in Figure 2), and a box representing the process file descriptor table (FDT).

The variables box also shows the process ID and the parent process ID at the top. This parent process ID shown is the parent that created the process, even if that parent terminated before the child does. The simulator uses character arrays in the form **buf***n* and integer variables in the form **total***n*, **child***n*, and **fd***n*, where n is any positive integer. The *n* can also be omitted completely, as in the examples in this paper. All variables are assumed to be declared and appear in the variable list when they are initialized by program execution.

The program appears to the right of the variables. The ▶ symbol indicates the program counter. It changes to ▷ when the process is suspended waiting for a child or joining a thread and it appears in red for the currently running process.

The file descriptor box has an entry for each open file descriptor. File descriptors 0, 1, and 2 (standard input, output, and error) are shown at the top. The simulator programs do not use these so no additional information is given about them. Each additional open file descriptor appears on its own line with an arrow pointing to the corresponding entry in the system open file table (SFT).

The **System Space** area shows three tables. At the top on the left is the SFT. This table has an entry for each open file. The simulator does not show the entries corresponding to standard input, output and error since the simulator programs do not use these. Three lines of information are shown for each open file. The first line indicates whether the file is open for reading or writing. The simulator only supports files open for one or the other, not both. Also shown on this line is the name of the file represented by this entry. This information is not contained in an actual SFT entry, but it is helpful here for understanding the operation of the simulator. The second line gives the file offset to be used by the next I/O operation using this entry. The third line in the entry shows a count of the number of file descriptor table entries that are using this entry. When the count is decremented (a process closes a file descriptor) to 0, the entry is removed. Arrows represent pointers stored at the source of the arrow. There is an arrow from each file descriptor table entry to the corresponding entry in the SFT. Each entry in the SFT has an arrow to an entry in the in-memory inode table.

To the right of the SFT is the *in-memory inode table*. This table contains one entry for each file that is open or has recently been opened. The simulator keeps these entries around even after no more open files refer to them. The first line of each entry gives a count of the number of SFT entries using this inode. This count should equal the number of arrows into the entry from the SFT. The second line indicates the permissions of the file. The simulator only supports two permissions, read only and write only. When the count is 0, the third line shows the name of the file. Otherwise, it indicates if the file is locked. The only lock supported by the simulator is related to the atomic write operations when the file is open with the `O_APPEND` flag.

Each inode has a single arrow coming out pointing to the disk block containing the contents of the file. The simulator assumes that all files fit in a single block. The disk blocks are shown in the **Disk Space** area.

Lastly, in the **System Space** area, the list of processes is shown at the lower left. One line is shown for each process giving its process ID and current state: running, ready, waiting for a child, joining a thread, zombie, or terminated. The parent of the original process has ID 1000, and we assume that this process is waiting for its child to complete. If a terminated process has its parent terminate before

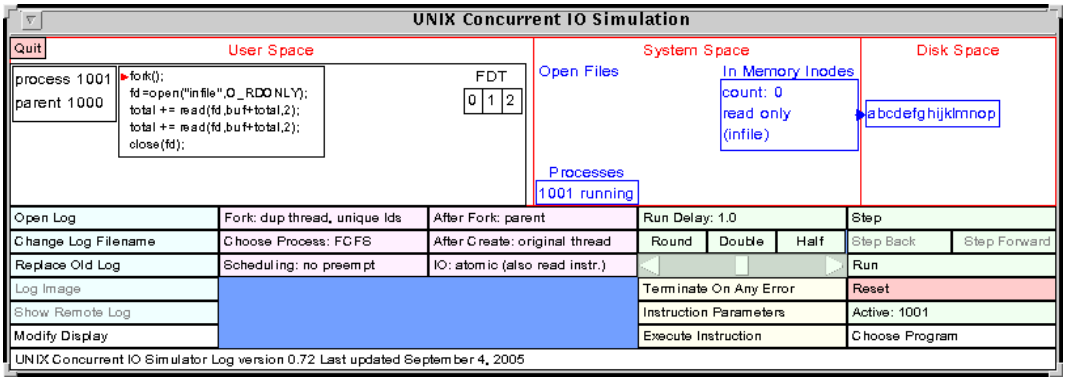

Figure 1: The simulator when it is first started.

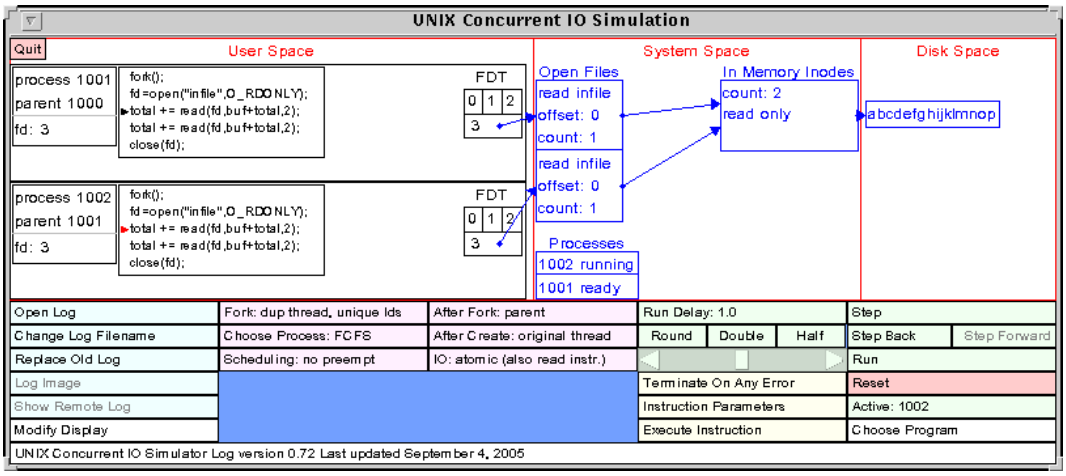

Figure 2: The simulator after a few instruction have been executed.

the parent waits for it, the child process is immediately inherited by the init process which waits for it.

The lower region of the simulator window of Figure 1 has five columns of buttons and controls. The first column controls logging functions and columns 2 through 4 mainly control scheduling. The last column contains buttons for running the simulator. The **Step** button executes one instruction of the running process. The **Run** button runs the program with a delay between instructions so you can watch the execution. You can vary the delay with the slider that is just to the left of the **Run** button. The default is to delay 1 second between instructions. When the program is running, the **Run** button changes to **Pause** which allows you to temporarily pause execution. When pushed, the button changes to **Resume**. The **Reset** button allows you to restart the program from the beginning. After some instructions have been executed, the **Step Back** and **Step Forward** buttons can be used to move back thought previous executed instructions so you can redo the execution and examine the tables at any point in the execution.

The simulator assumes that all instructions will execute without error if possible. Errors are classified as either fatal or non-fatal. Fatal errors include accessing an uninitialized variable or attempting to open a file with the incorrect permissions. Non-fatal errors include waiting for a child when no non-waited for children exist or closing a file that is not open. The running of a program always terminates when a fatal error occurs. Optionally, it can also terminate on non-fatal errors.

All I/O operations on valid file descriptors return without error. Writes always return the number of bytes requested to write and reads return the number of bytes requested or the number of bytes left to read in the file. By default, I/O is atomic so that a program does not lose the CPU during execution of an I/O instruction. Alternatively, the simulator can be set for non-atomic I/O. In this case, a slider determines the probability that the process loses the CPU after each byte is processed. Figure 3 shows a process that has lost the CPU after the first of two bytes have been read. One byte has been put in **buf** but the read has not returned so **total** has not been updated.

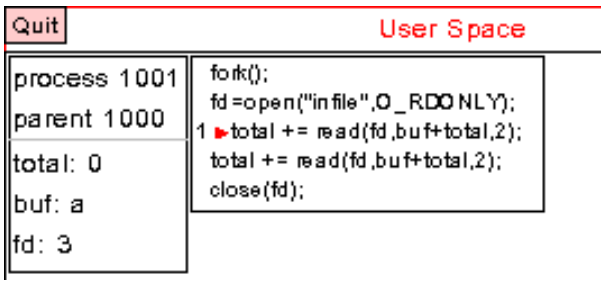

Figure 3: The CPU has been lost during an non-atomic read operation.

When a program is run with the **run** button, the simulator schedules the processes for the CPU using one of several standard algorithms. The user can choose non-preemptive scheduling, round robin with a given quantum, or random scheduling. In round robin scheduling, the quantum determines the number of instructions that can be executed before the process loses the CPU. In random scheduling, after each instruction is executed, the process loses the CPU with a given user-controllable probability. There are 3 options for how the next runnable process is chosen. These are FCFS, next ID, and random. In addition, forks and thread creation are treated in a special way, with 4 possibilities on which happens after each. For a fork, these are parent continues, child continues, either parent or child continues with equal probability, or random. In the last case a random ready process is chosen after a fork.

You can create a log file in HTML format on the fly. At any time you may include the diagram shown at the top of the simulator window along with the scheduling parameters.

The programs used by the simulator are read from ordinary disk files. A number of programs are included in the simulator distribution including the ones discussed in the following sections. Students can create additional programs with an ordinary text editor, and the simulator will find them if their names appear in the configuration file.

4. CONCURRENCY AMONG PROCESSES

A simple way to illustrate the interaction between the FDT and the SFT is to compare the action of these similar programs:

Program 1

```
fork();
fd = open("infile",O_RDONLY);
total += read(fd,buf+total,2);
total += read(fd,buf+total,2);
close(fd);
```

Program 2

```
fd = open("infile",O_RDONLY);
fork();
total += read(fd,buf+total,2);
total += read(fd,buf+total,2);
close(fd);
```

Figure 4 shows a part of the simulator diagram for each of these programs after fork and open have been executed. In Program 1, each process executes open so there are two distinct entries in the SFT, and therefore two file offsets. Each entry shows a count of 1. Both entries point to the same in-memory inode which shows a count of 2, indicating that two SFT entries are referencing it. When both processes have finished executing this code, each will have the first 4 bytes of the file in their respective buffers.

In Program 2, the open is executed before the fork so only one SFT entry exists. It shows a count of 2, indicating that 2 FDT entries are using it. Parent and child are sharing the same file offset, so under normal circumstances, when both processes have executed this code, the first 8 bytes of the file will be distributed between them. It is not possible to predict which bytes will go to each process, illustrating the unpredictability that often occurs when concurrent processes access the same shared variable. In this case the shared variable is the file offset stored in the shared SFT entry.

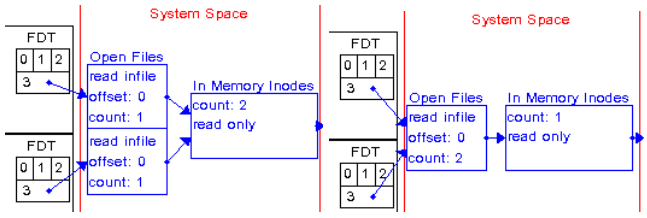

Figure 4: Part of the simulator diagram for Program 1 (left) and Program 2 (right).

When independent processes write to the same file, one process may overwrite data written by the other process. Consider the following program:

Program 3

```
child=fork();
fd = open("outfile",O_WRONLY | O_CREAT,0444);
if (child) {
   write(fd,"ab",2);
   write(fd,"cd",2);
}
else {
   write(fd,"AB",2);
   write(fd,"CD",2);
}
close(fd);
```

Parent and child each open the file and try to write 4 bytes. Assuming that each of the 2-byte writes is atomic and each write returns 2, when both processes are done, the file will contain 4 bytes. Figure 5 shows a part of the simulator diagram for this program after the parent has executed the first write followed by the first write of the child and the second write of the parent. By looking at the value of the file offset in the appropriate SFT entry, you can see what will happen next. Since the offset for the child is 2, its write instruction overwrites the **cd** already written by the parent.

Several variations of the program can be explored including interchanging the fork and open instructions (the simulator's Program 4), and adding the `O_TRUNC` and/or `O_APPEND` flags to the open instruction.

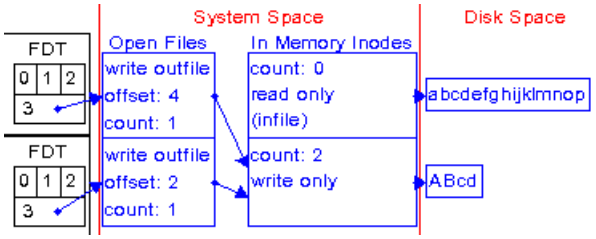

Figure 5: Part of the simulator diagram for Program 3.

5. CONCURRENCY AMONG THREADS

The simulator supports POSIX compatible create, join and detach functions. Unlike when processes are created with fork, creation of threads does not modify the SFT. In Program 5, a file is opened and a thread is created. The main process reads two bytes from the file twice and the thread reads two bytes once. If the instructions are executed atomically, the first 6 bytes of the file are read and it does not matter in what order the main process and the created thread execute.

Program 5

```
fd = open("infile",O_RDONLY)
pthread_create(&tid1,NULL,firstThread,NULL);
total+=read(fd,buf+total,2);
total+=read(fd,buf+total,2);
pthread_join(tid1,NULL);
```

it uses the following thread:

```
void *firstThread(void *args) {
   total+=read(fd,buf+total,2);
   return NULL;
}
```

When the main process exits, so do all of its created threads. With the default scheduling (**non-premptive, after create: original process**), the thread would never get to execute, it it were not for the join instruction. If the scheduling is changed to **after create:**

new thread the thread reads first. Alternatively, you could remove the join instruction and detach the thread so it does not exit when the main process terminates.

By default, the simulator assumes that all instructions are atomic. Even if file I/O is atomic on a given system, a read instruction like the one used by the simulator will rarely be atomic on real systems, and never on RISC hardware. An expression that starts out **total +=** will first perform the add in registers and then set **total** to the sum. If this instruction is executed concurrently with another instruction that modifies **total**, the result may depend on the order in which the operands are moved into the registers and exactly when the process loses the CPU. The simulator allows experimenting with this by allowing the process to lose the CPU at exactly one point during execution of the instruction. The **read** is allowed to complete and its return value is saved (in a register). At this point, the process may lose the CPU. When it gets the CPU back, **total** is moved into another register, the two are added and the result is stored back in **total**. Here is an example of a sequence of events that can lead to an unintended result. Assume that **total** is initially 0 and the file contains **abcdefgh**.

1) The main program executes a read, **buf** contains **ab**. The return value of 2 is saved but before **total** can be accessed, the process loses the CPU. The value of **total** is still 0.
2) The thread executes the read instruction. Since **total** is still 0, the 2 bytes read are put at the beginning of the buffer, overwriting what was previously read. The buffer now contains **cd**.
3) The main process gets the CPU and accesses **total**, whose value is now 2, and adds the return value of the **read** to it, making **total** equal to 4. This is the correct value of total, but only the first two slots in **buf** have been filled.
4) The main process reads 2 more bytes, but puts them in slots 4 and 5 of **buf**. The simulator shows **buf** as containing **cd..ef** where the two dots indicate uninitialized values.

The above scenario will occur with the simulator using Program 5 if instructions are set to be non-atomic, scheduling is RR with a quantum of 3, and after create is set to original process. Clearly, Program 5 needs some synchronization to make it behave correctly. One possibility would be to have the main process join the thread before doing any reads (supported by the simulator) or put mutex locks around the reads in both the main process and the thread. While the simulator does not support mutex locks, this has the same effect as making the instructions execute atomically, which is supported by the simulator.

6. NON-ATOMIC I/O

The POSIX standard does not usually require that I/O be atomic. Two exceptions are small writes to pipes and FIFOs (not supported by the simulator), and writes to a file with the **O_APPEND** flag set. Otherwise concurrent I/O to the same buffer or file needs to be protected. The simulator allows you to experiment with this.

Problems can occur if a process can lose the CPU during an I/O instruction. The POSIX standard does not specify whether the file offset is updated with each byte of I/O, or only after the I/O is complete. For concreteness, the simulator assumes that the file offset is updated with each byte processed.

When I/O is not atomic, the simulator's n-byte read instruction is treated as n+1 instructions, n to read the bytes and 1 to set the value of **total**. If we look at Program 5 again and run it with non-atomic I/O and RR scheduling with a quantum of 3, we can end up with different values in the buffer depending on the initialization of the random number generator. Possibilities include **bdef**, **abdf**, **cdef** and **bd..ef**.

7. CONCLUSIONS

The simulator has its limitations. It does not directly show how the operating system translates the name of the file in the **open** system call to get an inode. File names are kept only in the directory, and showing the directory in the simulator, with appropriate arrows to the inodes would only confuse the diagrams. I like to give this as part of an assignment by posing a question like the following one:

The simulator indicates the name of the file in the SFT entry. This does not actually appear in the entry in a real system. Where is the correspondence between the name of the file and the entry kept in a UNIX system?

The pictorial representation of the data structures in the User, System, and Disk spaces is a compromise between usability and accuracy. File names are included in the SFT and in-memory inode tables to key the entries to the corresponding open instructions. The simulator allows you to use the mouse to drag the various data structure around in their respective spaces to improve the readability of the diagrams. This was done in Figure 4.

There is no limitation on the number of processes or size of the programs handled by the simulator, other than Java memory limitations, but the display becomes unusable when the diagram will not fit on the screen.

A number of choices were made in the simulator design in handling the non-atomic nature of instructions. Rather than allowing a large number of configurable options, I made decided to only allow the I/O instructions to behave non-atomically. Of the instructions handled by the simulator, the read instruction has the greatest opportunity to behave non-atomically. The limited non-atomic behavior discussed in Sections 5 and 6 is sufficient to introduce the topic and allow students to experiment.

This simulator is part of a suite of seven simulators that have been designed to augment the operating system curriculum. All of the simulators are written in Java and can be run as Java applications on any system that supports the Java virtual machine. They are available on line [4] and can be downloaded individually, or in their entirety as a CD image. The CD is also available without charge by contacting the author. The simulators can also be run in demonstration mode as a Java applet directly from a browser by following the the appropriate links on the simulator web page. The web page also contains user guides for each simulator and information about using the simulators for in-class demonstrations and assignments.

This simulator is ideal for classroom demonstrations as it allows for single-stepping through programs and illustrates the dynamic relationship among the various data structures. When students ask questions, the **Step Back** button is useful for examining the various tables at and earlier step in the execution. At any time during the demonstration, the current state of the simulator can be saved so that it can be later restored for reference.

8. REFERENCES

[1] M. J. Bach, *The Design of the UNIX Operating System,* Prentice Hall, 1996.

[2] C. Crowley, *Operating Systems, A Design-Oriented Approach,* Irwin, 1997.

[3] G. Nutt, *Operating Systems, Third Edition,* Addison-Wesley, 2003.

[4] S. Robbins, Simulators for teaching operating systems, 2005. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/nsf/simulators**

[5] A. Silberschatz, P. B. Galvin and G. Gagne, *Operating System Concepts, Seventh Edition,* John Wiley and Sons, Inc, 2005, online Appendix A at **http:www.cs.yale.edu/homes/avi/os-book/os7/index.html**.

[6] W. R. Stevens, *Advanced Programming in the UNIX Encironment,* Addison Wesley, 1992.

[7] A. Tanenbaum, *Modern Operating Systems, Second Edition,* Prentice Hall, 2001.