Exploration of Process Interaction in Operating Systems: A Pipe-Fork Simulator

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

Abstract This paper examines the use of a simulator to explore process interaction in Unix. The simulator allows instructors to trace through a variety of programs and to show how the processes and pipes are connected. Students can create C language programs and see how changes in their code or changes in process scheduling affect the configuration of the processes and pipes as well as the output of the program. Students can also see the consequences of not protecting critical sections in an executing program. The simulator is flexible enough to allow the creation of process fans, chains and trees as well as unidirectional and bidirectional rings. The program is written in Java and can be run as a standalone application or as an applet from a browser.

1 Introduction

Teaching about the interaction of processes and resource sharing in operating systems is difficult, partly because of the scarcity of concrete examples. One of the simplest examples is a process that forks a child after creating a pipe. The pipe is shared by the original process and the child. One of the two processes can write to the pipe and the other can read from it. Figure 1 shows the situation after a new process has created a pipe and then forks a child. Processes are represented by ovals and pipes are represented by rectangles. The arrows represent communication paths and are labeled by the corresponding file descriptor. Each of the two processes, A and B, can write to the pipe using file descriptor 4 and read from it using file descriptor 3.

Two types of interaction are possible. The most obvious interaction is through communication. Processes can also interact by accessing a shared resource in a critical section.

SIGCSE'02, February 27- March 3, 2002, Covington, Kentucky, USA.

Although small writes to a pipe are atomic, writes to other shared resources, such as standard error, may not be. A write to standard error, particularly if it is in a loop, would be a critical section that needs to be protected.

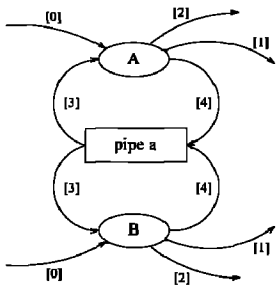

Figure 1: The result of creating a pipe followed by forking a child.

Once a process has written to the pipe in Figure 1, either process can read that information. In order to establish reliable communication in which a process will only read what the other process has written, two pipes are necessary. The code below illustrates this and also redirects standard input and output so that they access the pipes. Figure 2 shows the result.

```
#include <unistd.h>
int fd[2];
pid_t haschild;

pipe(fd);
dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
pipe(fd);
if (haschild = fork())
   dup2(fd[1], STDOUT_FILENO);
else
   dup2(fd[0], STDIN_FILENO);
close(fd[0]);
close(fd[1]);
```

Whatever either process writes to standard output can be read by the other process from standard input. This is a ring of

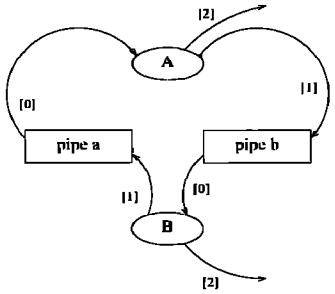

Figure 2: A ring of two processes.

two processes. It can be extended to more processes just by inserting part of the code in a loop [1]. The simple code can form a basis for a number of examples and student projects involving interprocess communication. See Chapter 4 of [2] for details.

When teaching this material I usually trace through the ring of two or three processes drawing diagrams similar to Figure 2. Each time a dup2 or `close` is executed, lines need to be erased. This makes it difficult for students to take notes, and most students do not really understand what is going on until they try it themselves. The example is a worthwhile exercise, as it enhances student understanding of file descriptor tables and forking a process. It is good for teaching the distinction between resources that are part of the process space and get duplicated by a `fork` and those that are part of the kernel space and are shared.

The ring code is simple, but the processes form a ring only in the sense of communication. The ring structure can be exploited only after additional code to communicate among the processes is added. It is not obvious how minor modifications of the code change the communication topology, and it is not simple to devise a procedure for determining the topology once the code it written. Tracing the program for more than 3 processes is difficult. If a mistake is made during tracing, it is difficult to recover without starting over.

2 The Simulator

All of these issues led to the development of a simple simulator to allow experimentation with `pipe`, dup2, `fork` and `close`. The simulator is a Java program that can be run either as an applet from a browser or as an application. It is available to be run from any machine connected to the Internet [5]. Figure 3 shows the main simulator window after it has run a program to create a ring of 5 processes. The upper left corner shows a diagram of the processes, pipes and file descriptors. The file descriptors for standard error are not shown. Initially, there is one process with process ID 100. As processes are created, process IDs are assigned sequentially. The upper right corner shows the code to be executed, with an arrow indicating the current program counter of the active process. The rest of the window consists of buttons

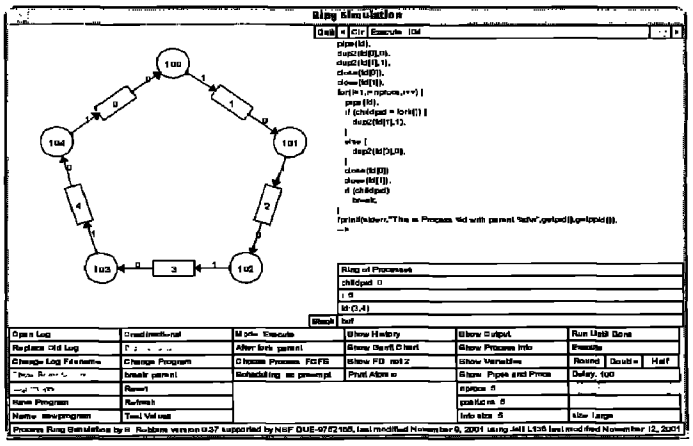

Figure 3: The main window of the simulator after running a program that creates a ring of 5 processes. The program executed is shown in the upper right part of the window.

and labels for controlling the simulator.

The simulation program has two modes of operation. In *Program Mode* the program can be modified. Instructions can be inserted or deleted. In *Execute Mode* the program can be run, either by single stepping through the code or running until completion. The simulator assumes a single CPU and includes several options for controlling the scheduling of multiple processes.

2.1 The Program

The user can modify the program, as long as the general structure is kept. The general structure of the program is as follows:

```
<instructions>
for (i=0;i<nprocs;i++) {
   <instructions>
   if (childpid = fork()) {
      <instructions>
   }
   else {
      <instructions>
   }
   <instructions>
   <break instruction>
}
<instructions>
```

In six places, an arbitrary list of instructions taken from a fixed set are allowed. The *break instruction* allows four options for breaking out of the for loop: parent only, child only, both, and neither.

The program assumes the following declarations:

```
int i;
int nprocs;
int childpid;
int fd[2];
int fd1[2];
char buf[BUFSIZE];
```

where BUFSIZE is sufficiently large to hold any input generated. Some of the instructions allowed include:

```
pipe(fd);
dup2(fd[0],0); dup2(fd[1],1);
close(0); close(1);
close(fd[0]); close(fd[1]);
wait(NULL);
```

There are similar instructions for `fd1` and a number of I/O instructions. There are instructions for filling or appending to `buf` character strings containing values of the variables `i` and `childpid` and process and parent process IDs. It can also be filled or appended with data read from standard input (as read from one of the pipes) and the contents of the buffer can be written to standard output (to write to a pipe) or standard error (to become output of the program). A complete user's guide is available [5] that explains all of the instructions and options for running the simulator.

3 Examples of Using the Simulator

There are many ways to use the simulator either for class demos or assignments. The class demo requires a computer with projection display. The program will run on any platform that supports Java and can be run from a browser, so that the program need not be loaded onto the target machine if the machine has access to the Internet. Students can download the program and run it on their home machines.

3.1 Process fans, chains, and trees

By eliminating the `pipe`, `dup2` and `close` lines from the code we can get a process fan (all forked processes have a common parent), a process chain (each process except the last has a single child) or a process tree (processes have different numbers of children). These examples are gotten by modifying the *break instruction* to have the child break, the parent break, or neither break, respectively. The simulator display can be set to show the parent-child relationships with an arrow from the parent to the child. Figure 4 shows the diagram generated by the simulator for a process fan and a process chain. Figure 5 shows the output generated by the simulator for a process tree on the left. The user can move the processes around to produce the diagram on the right with about a minute's work.

3.2 Tracing the standard ring of processes

The simulator can be used to trace the creation of processes and pipes by single stepping through the program. This has been done successfully in class and is a great improvement over doing it on the blackboard.

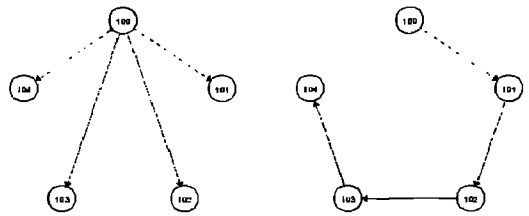

Figure 4: A fan and a chain of processes.

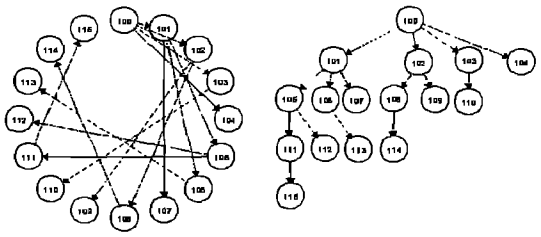

Figure 5: A tree of processes.

3.3 Experiments

There are many ways of making small changes in the standard ring program and most of these have consequences that would be hard to predict. Some changes in the ring code affect the internals of the program without affecting the running of the program or the output generated.

Experiment 1: Figure 6 shows the result of running the simulator omitting the two `close` statements at the beginning of the program. On the left is the result of running the program until the second pipe is generated. The upper diagram is the original program with the two `close` statements. File descriptors 3 and 4 are used for the second pipe (pipe 1). The lower diagram shows the result up to this point the the two close statements removed. Now file descriptors 5 and 6 are used for this pipe. The diagram on the right shows the result after the program completes. While the ring is correct and still uses standard input and output for communication, most of the processes can now access pipe 0, and so process 100 will not detect end-of-file on this pipe until all processes close their connections.

Experiment 2: In the original ring, the parent dups standard output, and the child dups standard input. What would happen if this were reversed? Figure 7 shows the result of running the simulator with this change. The diagram on the left comes from running the simulator with the dups interchanged. The diagram on the right is more clear and shows the result after each pipe has been manually rotated 180 degrees by the user. Now it is easy to see that the data flows in

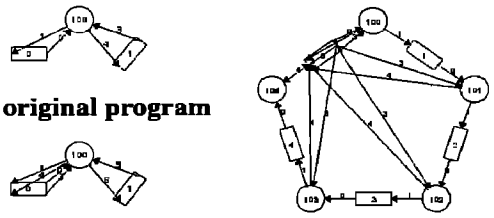

Figure 6: The result of omitting the first two close statements from the ring of processes. After the second pipe is created (left) and the final result (right).

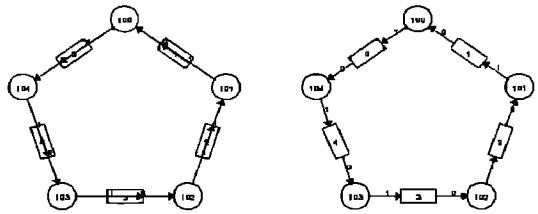

Figure 7: The diagram of the ring in which the parent dups standard input and the child dups standard output.

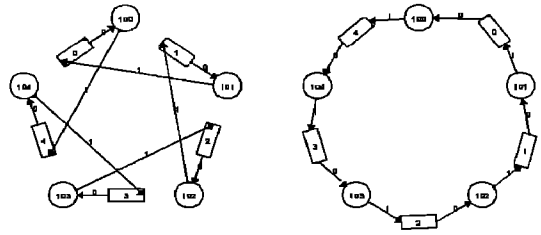

Figure 8: The diagram of the ring with the child breaking instead of the parent.

the opposite direction. That is, data flows toward processes with smaller PID rather than larger ones. This gives a hint as to how to make a bidirectional ring.

Experiment 3: Figure 8 shows the result of having the child break out of the loop instead of the parent. The diagram on the left was produced by the simulator running the program. In the diagram on the right, the user has reorganized the display to show data flow in the opposite direction.

3.4 Scheduling

If a program creates a process chain and each process outputs to the same resource, say standard error, how is the order of the output related to the order of creation of the processes? If such a program were run 100 times and each time the output were the same, can you conclude that the output will always be in the same order? The order of the output is determined by the scheduling of the processes. The simulator lets you control the scheduling in several ways. A process will always leave the CPU when it blocks waiting for input. The scheduling is determined by what happens in each of the following situations:

- A process blocks for I/O or terminates:
  The simulator allows for three possibilities when a process blocks or terminates: first-come/first-served (FCFS), the next highest process ID, or random.
- A process forks:
  The simulator allows the following modes: parent always executes, child always executes, parent or child executes with one randomly selected, any ready process executes with one chosen randomly.
- A process executes a non-blocking instruction:
  The simulator allows no preemption, round robin with a given quantum, where the quantum is a number of instructions, and random scheduling in which a process has a settable probability of losing the CPU after each instruction.
- A process does a non-blocking I/O request:
  The simulator never blocks on writes. Writes to a pipe are always atomic. It blocks on reads when there is no input available and a process exists that can write to the pipe. A read from a pipe will read all data currently in the pipe. The simulator can handle output to standard error either atomically without losing the CPU, or it can lose the CPU with a given probability after each character is output.

3.5 Interleaving of output

One of the common problems in concurrency is the interleaving of data from multiple processes attempting to output to the same device. When I first started giving assignments concerning this about a dozen years ago, my standard example was a program that would fork several times and have each process write a one-line message to standard output. On most machines, the output would be interleaved when about a half dozen processes were involved. At least one of the lines of output would be combined with another one, showing the need to protect the critical sections of this program. The same example no longer works on today's machines, as the probability of having interleaved output has been reduced by the speed of the machines and other factors. This makes the problem even more important as the most difficult problems to solve are the ones that occur infrequently.

The simulator program supports output of strings to standard error. The strings may contain the process ID of the process sending the string, the process ID of the parent process, and the values of program variables such as `i` and `childpid`. A window can be popped up showing the output generated by the program. The output generated by each process can be shown in a distinct color, allowing the user to easily see which characters are sent by each process. The simulator can be set to either have the output of an `fprintf` be atomic, or have a given probability of losing the CPU after each char-

acter is sent. Students can experiment to see how this probability affects the perceived output.

One way to protect this critical section is to have each process wait for its child to complete before sending output to standard error. Another method is to pass a token around the ring and allowing only the process with the token to produce output. Both of these can be tested with the simulator.

3.6 Fibonacci number calculation

A simple example of using the ring connectivity is the calculation of Fibonacci numbers [2]. The first process sends the first two Fibonacci numbers (1 and 1) to the next process in the ring in the form of an ASCII string. Each process then reads from the ring, decodes the two numbers, `x1` and `x2` and sends the string corresponding to `x2` and `x1+x2` to the next process. This continues for a certain number of iterations or until the calculation fails due to overflow. The simulator supports the call to a function, `fibconvert(char *buf)` that converts the input string to the output string needed for this computation.

3.7 Bidirectional ring

The simulator supports the use of a second array of file descriptors, `fd1[2]` and the corresponding `pipe`, `dup2`, and `close` functions to allow two pipes to be created on each iteration of the loop. Students can then write a program that creates a bidirectional ring in which file descriptor 1 (standard output) is used to send data in one direction and file descriptor 3 is used to send data in the other. Figure 9 shows the figure created by the simulator when a correct bidirectional ring program is run.

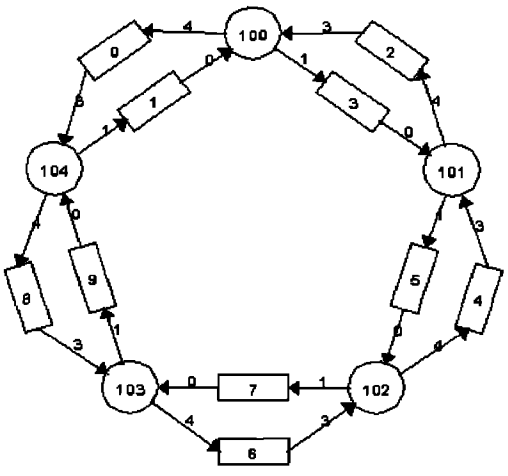

Figure 9: A bidirectional ring of processes.

4 Saving Programs and Logging

The simulator uses the Jeli package for logging [3]. An HTML log file can be created that contains the history of any process (what instructions it executed) and the output generated. The diagrams in the upper left and right corners of the simulator window can also be put in the log file. This allows students to create programs using the simulator and then compile and run the C code generated. Programs to be run by the simulator can either be created on the fly by the simulator or read in from a file.

5 Conclusions

The simulator was used in an undergraduate Operating Systems class in Spring 2001. First the students were asked to compile a version of the ring program written in C and run it with 5 processes. They were to observe the order of the output and how the output was affected by putting a `wait` command in various places. After a quick demo of the simulator, the students were asked to run the simulator on the same code and to experiment with the effect of different scheduling choices. They were to compare these with the results from running the C program. They were also asked to use the simulator to create a bidirectional ring. Most students were able to accomplish this task after the hint of looking at Experiment 2 from Section 3.3. The students enjoyed using the simulator and it generated favorable comments.

6 Acknowledgments

This work has been supported by an NSF grant: *A Web-Based Electronic Laboratory for Operating Systems and Computer Networks*, DUE-9752165 and is one of a collection of simulators designed supplement the teaching of operating systems [4].

References

[1] Robbins, K. A., Wagner, N. and Wenzel, D. J., Virtual rings: an introduction to concurrency. *Proc. 20th SIGCSE Technical Symposium on Computer Science Education*, 1989, pp. 23-28.

[2] Robbins, K. and Robbins, S., *Practical UNIX Programming, A Guide to Concurrency, Communication, and Multithreading*, Prentice Hall, 1996.

[3] Robbins, S., "Remote logging in Java using Jeli: A facility to enhance development of accessible educational software," *Proc. 31st SIGCSE Technical Symposium on Computer Science Education*, 2000, pp. 114-118.

[4] Robbins, S., NSF projects, 1999. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/nsf/**

[5] Robbins, S., Process ring simulator, 2001. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/nsf/ring/**