Virtual Rings: An Introduction to Concurrency

Kay A. Robbins, Neal R. Wagner, and Dennis J. Wenzel*

Division of Mathematics, Computer Science and Statistics
University of Texas at San Antonio
San Antonio, TX 78285

*also of Southwest Research Institute
San Antonio, TX 78284

Introduction

Concurrency and synchronization are difficult topics, and our operating systems students find a strictly theoretical presentation of these ideas to be unsatisfying. Many textbooks advocate a major course project in which the students write a simple operating system or modify an existing operating system [1, 2].

While there is little doubt that students learn a great deal from such an approach, they write a lot of code which is peripheral to the fundamental ideas of mutual exclusion and resource sharing: a large scale implementation may not give students enough practice in a simple setting for mastery of these difficult concepts. The problem is worse if the students are not already very familiar with the base operating system and the language (often a UNIX derivative and C) or if the course is offered on the quarter system so implementation time is short.

In this paper we offer some ideas for compromise between the theoretical and the implementation approaches to teaching operating systems. We have found that a ring of communicating processes generated in a UNIX environment can be the basis of many interesting projects for operating systems students in an upper division or beginning graduate class.

The ring can be used as a warmup on synchronization before a major implementation assignment, or it can be extended to fill an entire course. Students can directly explore the major ideas of synchronization and other aspects of concurrency and resource sharing without writing large amounts of code. In this paper we describe such a ring along with course project ideas. We assume a knowledge of C and of UNIX system calls particularly pipe creation and the fork [3, 4, 5].

The basic ring

Figure 1 shows a ring of 5 processes generated by a C program running under UNIX. Each process in the ring has its standard input connected to its predecessor's standard output and its standard output connected to its successor's standard input. Process 0 is the first parent. It is the process which

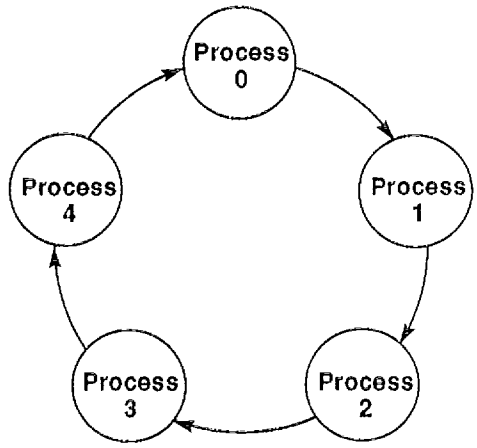

Figure 1: A ring of 5 processes

is created first. The other processes are successive generations of children of Process 0. The children inherit the environment of their parents.

In the sample ring generator program shown in Figure 2, the number of processes in the ring is a parameter furnished by the programmer on the command line invoking the ring program. Process 0 creates a pipe, connects its standard input to the read end of this pipe, and it connects its standard output to the write end of the pipe (Figure 3a).

The program then performs a loop for processes 1 to N-1, first creating a pipe and then creating the next process by a fork. Processes 0 and 1 and their respective connections to the pipes are shown in figure 3b. Each process knows its relative position on the ring by the value of the **iproc** which is set during its generation. The inheritance properties of UNIX assure that each process shares the same standard error file.

```
#include   <stdio.h>

/*****************************************************************************/
/*****************************************************************************/
/*      Sample C program for generating a unidirectional ring of processes.  */
/*      Invoke this program with a command line argument indicating the      */
/*      number of processes on the ring.  Communication is done via pipes    */
/*      which connect the standard output of a process to the standard input */
/*      of its successor on the ring.                                        */
/*****************************************************************************/
/*****************************************************************************/

main  (argc,argv)        /* generate a ring of processes connected with pipes */
int  argc;                                    /* number of command arguments */
char **argv;                                       /* command argument list */

{
      int   i;                  /* loop index                                 */
      int   iproc;              /* number of this process (starting with 1)   */
      int   child;              /* process id of the spawned child            */
      int   ischild;            /* indicates process should spawn another     */
      int   pid;                /* process id work variable                   */
      int   nprocs;             /* total number of processes in ring          */
      int   filedes[2];         /* file descriptors returned by 'pipe'        */

            /* check command line for a valid number of processes to generate */
      if (argc < 2 || 0 >= (nprocs = atoi (argv[1]))) {
            fprintf (stderr, "usage: %s nprocs\n", argv[0]);
            exit(1);
      }
            /* create a pipe with process stdin connected to stdout via pipe */
      if (-1 == pipe (filedes))     serr("create pipe");
      if (-1 == close (0))          serr("close stdin");
      if (-1 == dup (filedes[0]))   serr("stdin pts to pipe");
      if (-1 == close (1))          serr("close stdout");
      if (-1 == dup (filedes[1]))   serr("stdout pts to pipe");
      if (-1 == close (filedes[0])) serr("close old fids to pipe");
      if (-1 == close (filedes[1])) serr("close old fids to pipe");
```

```
            /* create the remaining processes with their connecting pipes */
      ischild = 1;
      for (iproc = 1; (iproc < nprocs) && ischild;  ++iproc)       {

          if (-1 == pipe (filedes))  serr ("next pipe");       /* create pipe */
          if (-1 == (child = fork ( ))) serr ("fork");   /* create next child */

          if (child > 0)  {            /* for parent process, reassign stdout */
             if (-1 == close (1)) serr ("reassign stdout");
             if (-1 == dup(filedes[1])) serr ("stdout pts to pipe");
             ischild = 0;                 /*parent will not continue to spawn */

          }  else  {                     /* for child process, reassign stdin */
             if (-1 == close (0)) serr ("reassign stdin");
             if (-1 == dup (filedes[0])) serr ("stdin pts from pipe");
             ischild = iproc;                /* save child's creation number */
          }  /* end if */

          if (-1 == close (filedes[0])) serr ("release extra fids");
          if (-1 == close (filedes[1])) serr ("release extra fids");

      }  /*  end for */
                                          /*  <----Point A is here ******** */
       exit (0);
}     /* end of main program here */


serr (msg)              /* report a fatal system error message and terminate */
char  *msg;                                                /* message prefix */
{
     perror (msg);
     exit (1);
}   /* end serr */
```

Figure 2: A sample C program to create a ring of processes

Mutual exclusion

Processes compete for resources in a typical computer system. Some resources (such as files opened for READ) may be shared, while other resources (such as files opened for WRITE) can only be held by one process at a time. We will only consider resources which require exclusive access in the following discussion.

A code segment which references a shared resource is called a critical section. We require that when a process executes a critical section, no other process will be executing a critical section of code which references the same resource. This exclusivity during resource sharing is called the principle of mutual exclusion [1, 2, 6]. As an illustration of the importance of this idea, consider the following example:

Example:

What happens if the following code is inserted at point A of Figure 2?

```
for (i = 0; i < 10; i++)
   fprintf(stderr,
           "I am %d\n", getpid());
```

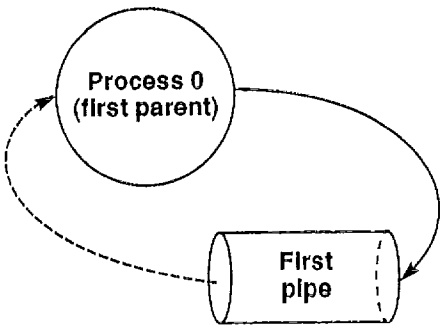

Figure 3a: The first process has its stdin connected to stdout via pipe

Each process attempts to write its process id to the standard error file, **stderr**, 10 times. However, since all processes share the same standard error file, characters in the messages of the different processes will be intermingled as they are written. Each process must therefore acquire exclusive access to the shared resource (in this case the standard error file) before writing its message.

There are a number of mutual exclusion algorithms which can easily be implemented on the ring. The algorithms can be centralized (one process is designated as the resource manager and is in charge of granting access to the resource) or decentralized (all processes have a voice in deciding who gets exclusive access).

Mutual exclusion with tokens

The simplest distributed scheme is based on tokens. The token is a special message which is circulated around the ring. If a process wishes to obtain exclusive access to a resource, it must first acquire the token. Once it has read the token message, it can proceed with its critical section. When the process has completed its access to the shared resource, it releases exclusive access by placing the token back on the ring (writing the token message on its standard output).

A process which does not wish to enter its critical section must pass the token message to the next process. Since the node processes on the ring must perform a great deal of message passing, we will assume that the node processes themselves only handle message passing. User processes with pipes for communication will fork from these nodes to perform other activities such as the execution of user programs.

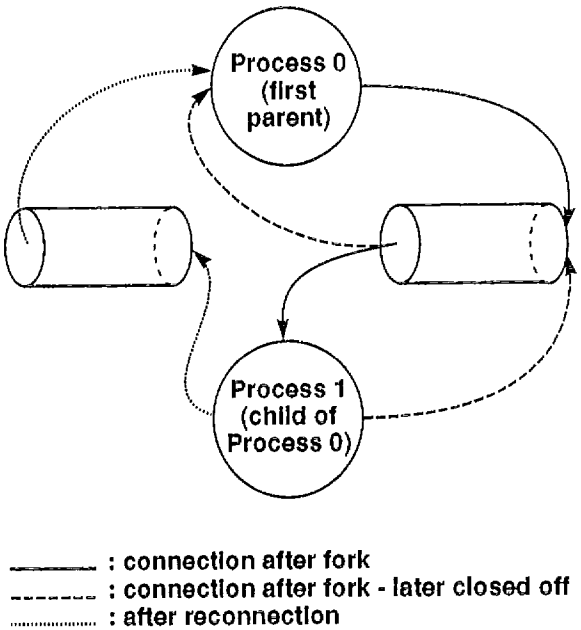

Figure 3b: The ring after two processes have been created

The token method described above involves heavy message traffic when no mutual exclusion is desired because the token keeps circulating on the ring. Also, the order in which processes acquire exclusive access is partly based on their relative positions on the ring. The next section describes a more complicated algorithm based on voting.

Mutual exclusion by voting

The method described in this section is based on an algorithm of Chang and Roberts for extrema finding [7]. When processes contend for exclusive access, they take a vote on the ring to see who gets the access. The method only generates traffic when a process wishes to acquire exclusive access. Furthermore, the voting algorithm can be weighted to allow a variety of priority schemes for allocating exclusive access, including allowing the process

which has been waiting the longest to proceed.

Voting takes place as follows. Processes which are not contending for exclusive access do not participate in the vote. Each voting process generates a voting message with a unique, two-part ID. The first part of the ID is a sequence number; the second part of the ID is the process number (0 to N-1).

The sequence number can be based on the current clock time or on the number of times the process has acquired mutual exclusion in the past. We use the latter method which gives an algorithm that guarantees all processes equal access to the resource.

To vote, the process puts its ID message on the ring. Each process which is not participating in the vote forwards the ID messages as they come in to the next process on the ring. When a process which is voting receives an ID message, it will take action according to one of three cases:

1. The incoming message has a higher ID than its own vote. In this case the process throws away the incoming message.

2. The incoming message has a lower ID than its own vote. In this case the process forwards the message.

3. The incoming message is its own. In this case the process has won the ballot and can begin its critical section.

A process relinquishes access by sending a release message around the ring. If any other processes are waiting for access, the balloting begins again. The winner is the process whose ID message is the lowest for that ballot.

6. Sample assignments for the ring

Once the students have implemented a basic ring, they can extend it to explore many aspects of concurrency. Here is a sample assignment:

> Implement decentralized mutual exclusion for the ring of processes using a voting algorithm. The protected resource will be the standard error file common to all processes on the ring. After the ring is created, each process should loop to write a specified number of messages to the error file in the following manner.
>
> 1. Delay a random amount of time.
>
> 2. Read the time at which the transaction begins.
>
> 3. Acquire exclusive access.
>
> 4. Read the time at which exclusive access is acquired.
>
> 5. Write a message to the error file which includes: the PID, the time at which the transaction began, and the time at which exclusive access was acquired.
>
> 6. Release exclusive access.
>
> Devise a method for determining when all processes have completed their transactions. Kill the ring when completion is detected. The number of error messages to be written by each process to the file and the number of processes in the ring should be passed on the command line which invokes the program.

Extensions to basic assignment

Here are some extensions of the ring assignment.

a. Designate the first process as a monitor and create the ring as a child of the monitor. You can then inject commands from the terminal to have particular nodes EXEC new procedures as children. You can also enter commands to kill processes which are being executed on the ring.

b. Design the ring so that nodes can be added or deleted dynamically.

c. Implement more sophisticated mutual exclusion algorithms [8, 9, 10].

d. Implement the dining philosopher algorithms on a two way ring [11]. Here tokens representing the forks

must be passed back and forth between adjacent processes.

e. Implement sharing of multiple reusable resources [12].

f. Implement centralized or decentralized algorithms for deadlock detection. [13].

g. Implement centralized or decentralized algorithms for process scheduling. [6].

Discussion

The creation of the ring was itself a nontrivial assignment, particularly for students who were not expert C and UNIX programmers. Our courses tended to emphasize system calls and low-level features of UNIX while deemphasizing C and UNIX details (such as shell programming). This was a nice introduction for the inexperienced programmers since it goes to the heart of synchronization and process control without overwhelming students with irrelevant details. Students should be encouraged to study various special features of UNIX, including no-delay reading from pipes and the select and signal system calls.

We used the ring exercise in both undergraduate and graduate operating systems courses. The undergraduate course, which was taught from Tanenbaum [1] without directly using MINIX, had data structures and computer organization as prerequisites. The ring assignments were preceded by an initial fork-exec-wait assignment and a stripped-down shell. In a future version of our undergraduate course we plan to use the ring during the first part of the semester followed by some MINIX implementation at the end.

The graduate course had an undergraduate operating systems course as a prerequisite. The graduate course used Maekawa[10], Bach [4], and readings from the literature. The graduate students completed a number of demanding theoretical assignments in addition to the ring.

We found that the students were excited by the problem and that better students immmediately devised their own extensions. The graduate students concentrated on implementing various distributed algorithms, while the undergraduates were given more basic process control problems. In both groups, our feedback indicated that the students had a better feeling for concurrency after working on the ring. As instructors we liked the project because it offered many variations.

Bibliography

1. Tanenbaum, A. S., "Operating Systems: Design and Implementation", Prentice Hall, 1987.

2. Comer, D., "Operating Systems Design: The XINU System", Prentice Hall, 1987.

3. Kernighan, B. W., and Ritchie, D. M., "The C Programming Language", Prentice-Hall, 1978.

4. Bach, M. J. "The Design of the Unix Operating System", Prentice-Hall, 1986.

5. Kernighan, B. W., and Pike, R., "The UNIX Programming Environment", Prentice-Hall, 1984.

6. Maekawa, M., Oldehoeft, A. E., and Oldehoeft, R. R., "Operating systems: Advanced Concepts", Benjamin/Commings 1987.

7. Chang, E. J. and Roberts, R., "An Improved Algorithm for Decentralized Extrema-finding in Circular Configurations of Processors", Comm of ACM 22 no 5 (1979), pp 281-283.

8. Hirschberg, D. S. and Sinclair, J. B., "Decentralized Extrema-finding in Circular Configurations of Processors", Comm of ACM 13 no 11 (1980), pp 627-278.

9. Dolev, D., Klawe, M. and Rodeh, M., "An O(n logn) Unidirectional Distributed Algorithm for Extrema Finding in a Circle", J of Algorithms 3 (1982), pp 245-260.

****

RINGS-- continued on page 39

the standard has precisely specified the behavior of the preprocessor for all scanning and replacement, especially where a macro contains an invocation of another macro.

An important portion of the standardization effort for any programming language is to make precise the features of the language which have not previously been painted in black and white. In addition, the standardization activity defines certain *grey* areas where the language implementor has some degree of freedom to solve his local problems and/or to provide more efficient code. The C Standard is not without a broad effort to supply the definitions of these exactions (or inexactions, as the case may be). In a parallel effort, the C Standard defines the issues and features involved in producing a *strictly portable* program in C. The reader is referred to Appendix C of Kernighan and Ritchie [2] for a complete listing of changes in the C language and to Plum [5] for additional dialogue on *all* of the issues in the proposed C Standard.

Summary

Features have been added to the C language in the ANSI C Standard which will allow C programmers to write more reliable, robust, and efficient code. Students of the block-structured paradigm will find that the new C provides for much stronger type-checking in the context of separately-compilable C files. Users of the new C, especially student implementors of large software engineering projects, should find that the language now provides an important measure of additional support for their efforts.

References

1. Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N. J., 1978.

2. Brian Kernighan and Dennis Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, N. J., 1988.

3. Narain Gehani, *C, An Advanced Introduction*, Computer Science Press, Rockville, Maryland, 1984.

4. David Prosser, *Draft Proposed American National Standard for Information Systems — Programming Language C*, X3 Secretariat, CBEMA, Washington, D. C.

5. Thomas Plum, *Notes on The Draft C Standard*, Plum Hall, Cardiff, N. J., 1987.

6. Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, Massachusetts, 1986.

7. Brad Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison Wesley, Reading, Massachusetts, 1986.

8. R. Lessman, "C++ and the Object-Oriented Paradigm," *Proceedings of the 27th Annual Southeast Regional ACM Conference*, April, 1989.

---

*UNIX is a Trademark of AT&T Bell Laboratories.

*Objective-C is a Trademark of The Stepstone Corporation.

*MS-DOS is a Trademark of Microsoft Corporation.

*PDP and VMS are Trademarks of Digital Equipment Corporation.

****

RINGS-- continued from page 28

10. Rotem, D., Korach, E., and Santoro, N., "Analysis of a Distributed Algorithm for Extrema Finding in a Ring", J. of Parallel and Distributed Computing 4 (1987), pp 575-591.

11. Dijkstra, E. W., "Cooperating Sequential Processes", In Programming Languages, F. Geunys (Ed.), Academic Press, 1968, pp 43-112.

12. Ibaraki, T. and Katoh, N., "Resource Allocation Problems: Algorithmic Approaches", MIT Press, 1988.

13. Raynal, M., "Algorithms for Mutual Exclusion", MIT Press, 1986.