# Systems Projects for a Computer Science Course

**Mohammad B. Dadfar, Sub Ramakrishnan**

**Department of Computer Science**
**Bowling Green State University**
**Bowling Green, Ohio 43403**
**Phone: (419)372-2337 Fax: (419)372-8061**
**email: datacomm@cs.bgsu.edu**

Abstract

In this paper we discuss some practical and useful projects for our operating systems / data communications course. Most of our projects are assigned in a UNIX platform. The projects deal with a multiprogramming environment where there are several processes running concurrently. The operating systems projects use different methods for process synchronization and cooperation including message passing and the use of semaphores. We use different methods for establishing communications between processes, including bi-directional pipes. Other projects are concerned with data communications aspect of the course.

## 1. Introduction

Due to the increasing demand for people with expertise in the area of data communications and networks and the importance of operating systems concepts, our department decided to include a required undergraduate course that covers the fundamental issues in both areas (CS 327). We have been offering a course in each of these two areas for many years. However, none of these two courses are mandatory for our undergraduate students. Many students completed their undergraduate program without taking a course in operating systems or data communications. Since operating systems and computer networks can play an important role in understanding other computer science topics, we feel that students majoring in computer science should have at least one course in these areas. (The ACM/IEEE Computing Curricula 2001 also recommends this approach.)

Our course, CS 327, provides an introduction to both operating systems and networks. It is a one-semester long mandatory course designed for computer science students at the sophomore or junior level. Students wishing to know more of operating systems and/or networking concepts can take our more advanced courses. In CS 327, we cover introductory topics including process management, concurrent processes, process scheduling, protocol architecture, TCP/IP suite, brief overview of broadband services, client-server communication and web enabling applications.

Practical and useful projects in this course is very important. These projects help students to gain an insight into the operating systems concepts and networking and to better understand the topics covered by the instructor. In the past we have assigned some projects that have been very useful[1, 2, 3]. In this paper we discuss some practical projects given to our students. Most of our projects are assigned in a UNIX platform. Our students in CS 327 know UNIX and have done extensive programming in C/C++. We will discuss these projects and share the problem-solving phase with the readers.

The projects deal with a multiprogramming environment where there are several processes running concurrently. We use different methods for process synchronization and cooperation. The first project uses the message passing method to force the processes run in a specified sequence. Each process uses a separate pipe to send its information to another process. In the next project all the children processes use the same pipe to pass their information to the parent process. The third project requires the use of semaphores and semaphore operations to accomplish the same task. At the end we briefly discuss another project dealing with the *client/server* communication concepts. Other projects that are primarily concerned with data communications aspect of the course are not discussed here. We use different methods for establishing communications between processes, including bi-directional pipes. In some projects students use instructor supplied object modules to build an application.

We share our experience and implementation details with readers with a view to possibly help other instructors integrate such ideas in their offerings. Our students feel that completing these projects have helped them to gain a better understanding of networking and inter-process communication issues.

The projects proposed in this paper can also be extended in a number of ways, which may be assigned in follow-up elective courses in data communications and/or operating systems.

## 2. First Project: Using Message Passing

The objective of this project is to understand process creation, understand simple communication between processes, and become familiar with some of the system calls. In this assignment students will study communication between parent and child processes. The parent process creates two children; each child generates a sequence of *n* random numbers in the range 0 to 999. Each number is passed to the parent process. The parent prints the numbers received from children. One child initializes the random number seed to *seed1* and the other child to *seed2*. Each child process runs a loop *n* times and each time generates a random number and sends it to the parent process. The arguments *n*, *seed1*, and *seed2* are given as command line arguments. We provide students with some sample programs that deal with *fork* and *pipe* statements.

Couple of observations are in order. The two child processes are running asynchronously. Thus, it is not possible to predict the interleaving of the two children. In fact, it is quite possible one child runs through its loop sending all of the numbers to the parent before the second child gets a chance to run. However, each time through the parent loop the parent reads a random number from the first child and then from the second child in this order. Thus, no matter how the two children are interleaved, the parent reads the numbers from the two children in strict order. The following is a possible solution to this assignment.

```
/*******************************************************************************
*  Problem:  Parent creates two children.
*     Each child generates n random numbers and send each number to
*     the parent through a pipe.  The parent prints the numbers
*     received from children.
*******************************************************************************/
#include <sys/types.h>
#include <iostream.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
 { // check for the number of arguments
   if (argc != 4)
     {  cout << "You must supply two seeds and the number of random numbers."
             << endl;  exit(1); }
   int pid1, pid2, seed1, seed2, i, count;
   int pipe1[2], pipe2[2], buffer[1];
   count = atoi (argv[3]);
   // create the first pipe, and exit if it fails
   if (pipe(pipe1) < 0 )
      { cout << "Pipe 1 cannot be created" << endl;  exit(1);}
   // create the first child
   pid1 = fork();
   if (pid1 == -1)
      {  cerr << "The child process cannot be created." << endl;  exit(1);}
   if (pid1 == 0)  // child1 process
     {  seed1 = atoi (argv[1]);
        srand(seed1);
        close (pipe1[0]);   // does not read from the pipe
        // generate random numbers and sent them to parent
        for (i=0; i<count; i++)
           {  buffer[0] = rand() % 1000;
              write(pipe1[1], buffer, 4);
           }
        close(pipe1[1]);
        exit(0);
     } // end of child1 process
   // parent process continues
   else
     {   // create the second pipe, and exit if it fails
         if (pipe(pipe2) < 0 )
           {  cout << "Pipe 2 cannot be created" << endl;  exit(1); }
         // create the second child
         pid2 = fork();
         if (pid2 == -1)
            {  cerr << "The child process cannot be created." << endl;
               exit(1); }
         if (pid2 == 0)  // child2 process
           {  seed2 = atoi (argv[2]);
              srand(seed2);
              close (pipe2[0]);   // does not read from the pipe
              // generate random numbers and sent them to parent
              for (i=0; i<count; i++)
                 {  buffer[0] = rand() % 1000;
                    write(pipe2[1], buffer, 4);
                  }
               close(pipe2[1]);
               exit(0);
            } // end of child2 process
           else
            { // Parent process continues.  Close unused ends of pipes.
              close(pipe1[1]);  close(pipe2[1]);
              // read the numbers sent by children
              for (i=0; i<count; i++)
                {  read(pipe1[0], buffer, 4);
                    cout << " The number received from child 1: "
                         << buffer[0] << endl;
                    read(pipe2[0], buffer, 4);
                    cout << " The number received from child 2: "
                         << buffer[0] << endl;
                }          // close the pipes
              close(pipe1[0]);
              close(pipe2[0]);
              exit(0);
            }  /* end of parent process */
       }
     }
```

3. Second Project: Synchronization Between Processes

The objective of this project is to understand the impact of CPU scheduling, understand timing issues, and become familiar with some of the system calls. Recall that the first project relied on two different pipes to pass the random numbers to the parent. The parent reads numbers from the two pipes, in order, so it gets a random number from each of the two children. In this assignment, the two children share the same pipe to pass the random numbers to parent. Yet, during each iteration of the parent loop, the parent gets a random number from each of the two children. Students are asked to modify Project #1 as follows:

- The parent process creates two children. Each child generates a sequence of *n* random numbers in the range 0 to 999. As each number is generated the child passes its ID and the random number to the parent.
- After sending its information, each child sleeps for a while using usleep(…)system call.
- There is only one pipe to pass random numbers and IDs from children to parent. (Both children use the same pipe to pass information.)
- The parent prints the child's information as received, in the form: child ID, random number.
- The printout by the parent is such that odd lines are from first child while even lines are from second child. Use additional pipes to make this happen.
- One child initializes the random number seed to *seed1* and the other child to *seed2*.

Students are unsure how parent interleaves printing of the random numbers when both children use the same pipe to pass numbers to the parent. Eventually they figure out the necessary synchronization mechanisms using extra "control" pipes. Often they come up with different ways to solve the problem. For example, parent signals first child to go (through control pipe 1), reads the random number then signals the second child to proceed (through control pipe 2) before reading random number again. Similarly, the children wait on the control pipe before sending random number to the parent. Another possibility is to synchronize the two children without parent involvement.

Since the second project is a simple extension to the first project, we do not provide the solution.

## 4. Third Project: Use of Semaphores for Process Synchronization

The objective of this project is to understand the effect of CPU scheduling and understand semaphores and their applications. This project is a further refinement to the second project. As before, both children share the same pipe to pass their random numbers to the parent. However, this project uses semaphores to provide for synchronization between the two children. Note that they cannot use any other pipes for control. Students are expected to enhance their ability to use semaphore and reuse existing C++ classes. Students make sure that they destroy, under program control, all of the semaphores prior to exiting. All processes are to be completed before the main program is finished. Since UNIX semaphore implementation is nontrivial, we provide students with a relocatable module that contains an easy to use, *Semaphore*, class (semaphore.o). Students instantiate the *Semaphore* class. They have access to some sample programs on using semaphores. The following is the project description and a possible solution to this assignment. We also show a sample run of the program.

```
/*******************************************************************************
* Problem: Use of semaphores for process synchronization.
*    Each child generates n random numbers and send each number with
*    its ID to the parent through a single pipe. Using semaphores
*    for process synchronization the two child processes send their
*    information (ID and random number) in an alternate way.
*    The seed for random numbers and the count for random numbers are
*    the command line arguments.
*
*    The parent process performs a signal on each child's semaphore
*    and then reads from the pipe. Each child process first performs
*    a wait on its semaphore then it sends the information to the
*    parent process through the pipe. At the end the parent process
*    deletes the semaphores.
*******************************************************************************/
#include <sys/types.h>
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include "/home/cs/dadfar/cs327/semaphore.h"  // instructor file
int main(int argc, char *argv[])
 { // check for the number of arguments
   if (argc != 4)
     { cout << "You must supply two seeds and the number of "
            << "random numbers." << endl;
       exit(1);
     }
   int pid1, pid2, seed1, seed2, i, count;
   int datapipe[2], buffer[2];
   count = atoi (argv[3]);
   // create the pipe, and exit if it fails
   if (pipe(datapipe) < 0 )
      { cout << "The pipe cannot be created" << endl;
        exit(1);
      }
```

```
// create two semaphores, called 0 and 1 for the children
Semaphore semaphore(12345, 2);
// initialize both semaphores to value 0
semaphore.Init(0, 0);
semaphore.Init(1, 0);
// create the first child
pid1 = fork();
if (pid1 == -1)
  {  cerr << "The child process cannot be created." << endl;
     exit(1);
  }


//************************************************** Child1 ***
if (pid1 == 0)
 {  seed1 = atoi (argv[1]);
    srand(seed1);
    close (datapipe[0]);     // does not read from the pipe
    pid1 = getpid();
    // wait for your turn by performing a wait on your
    // semaphore and then generate a random number
    // and send it to the parent along with your ID
    for (i=0; i<count; i++)
      {  semaphore.Wait(0);
         buffer[0] = pid1;
         buffer[1] = rand() % 1000;
         write(datapipe[1], buffer, 8);
      }
    close (datapipe[1]);
    exit(0);
 }
//**** end of child1 process ************************************

// parent process continues
else
 {   // create the second child
     pid2 = fork();
     if (pid2 == -1)
       {  cerr << "The child process cannot be created." << endl;
          exit(1);
       }
     //********************************************* Child2 ***
     if (pid2 == 0)
      {  seed2 = atoi (argv[2]);
         srand(seed2);
         close (datapipe[0]);  // does not read from the pipe
         pid2 = getpid();
         // wait for your turn and then send the info to the parent
         for (i=0; i<count; i++)
           {  semaphore.Wait(1);
              buffer[0] = pid2;
              buffer[1] = rand() % 1000;
              write(datapipe[1], buffer, 8);
           }
         close (datapipe[1]);
         exit(0);
      }     // end of child2 process
     //**********************************************************

     else
       { // *******  parent process continues  *******
         // close unused end of the pipe
         close(datapipe[1]);
         // Perform a signal on the children's semaphores and
         // read the information sent by children.
```

```
                for (i=0; i<count; i++)
                  { // ******* Signal child 1 and read from child 1.
                    semaphore.Signal(0);
                    read(datapipe[0], buffer, 8);
                    cout << " The child ID is: " << buffer[0]
                        << " and the number received is: " << buffer[1] << endl;
                    // ******* Signal child 2 and read from child 2.
                    semaphore.Signal(1);
                    read(datapipe[0], buffer, 8);
                   co ut << " The child ID is: " << buffer[0]
                        << " and the number received is: " << buffer[1] << endl;
                  }
                // close the pipe and delete the semaphores
                close(datapipe[0]);
                semaphore.Destroy();
                exit (0);
              }  /* end of parent process */
        }
    }

/**************************************************************
$ g++  lab3key.cpp  semaphore.o  -o a.out
$
$ a.out  2000  3000  2
 The child ID is: 17557 and the number received is: 876
 The child ID is: 17558 and the number received is: 431
 The child ID is: 17557 and the number received is: 186
 The child ID is: 17558 and the number received is: 661
$
$ ipcs
**************************************************************/
```

## 5. Fourth Project: Client/Server Communication

The objective of this project is to become familiar with the concept of *client/server* networking and understand how to set up *client/server* connection. In a *client/server* paradigm one application (client) actively initiates communications by sending requests to another application (server). The server application is waiting passively to receive specific type of messages and respond to these incoming requests. You may refer to other articles[3] for a detailed discussion of *client/server* communication concepts.

This project is similar to projects discussed in an earlier paper[2]. By completing this project students learn the difference between a client and a server. Students are asked to write the client for *finger* protocol (port # 79) and then finger a user (supplied in command line) and display the response from server. UNIX *client/server* programming primitives are a bit involved. We supply a relocatable module which students link to their program. The instructor-supplied module provides a connectToHost primitive which in turn invokes the standard connect primitive. The following is the project description and a possible solution to this assignment.

```
/*****************************************************************************
* Problem:  Use of client/server networking.
*    In this assignment we write the client for finger protocol.
*    The following C++ client/server routines have been provided by
*    the instructor.
*    int socket (AF_INET, SOCK_STREAM, 0):
*     Allocate a socket and return the socket number.  The returned
*     value is used as the first parameter in the following routines:
*    int connectToHost (int socketNO, char * serverMachineName,
*                       int serverPortNO):
*      Connect to serverMachineName at serverPortNO.  Returns a
*      negative value upon error.
*    int read (socketNO, FromServer, maxLength):
*      Read (up to maxLength bytes) from the server and store it in
*      FromServer.  Actual number bytes read is returned.
*    int write (int socketNO, char *ToServer, int lengthOfToServer):
*      Write to the server lengthOfToServer bytes from the array
*      ToServer.  Returns negative value upon error.
*    int close (int socketNO):
*      Close the connection.  Returns negative value upon error.
*
*    To compile:
*     g++ lab4Client.cpp /home/cs/dadfar/cs327/lab4/clientLibrary.o
*         -o client -lsocket -lnsl
*    Run the program as:
*     client allegro.cs.bgsu.edu 79 anyUserYouWantToFinger
*****************************************************************************/

#include <stdio.h>
#include <iostream.h>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include "/home/cs/dadfar/cs327/lab4/clientServer.h"

int connectToHost(int socketNO, char * serverMachineName, int serverPortNO);
int connect(int s, struct sockaddr *name, int namelen);
int socketNO, n, bufferSize = 1024;

int main(int argc, char *argv[])
{  char FromServer[bufferSize];
  // check for the number of arguments and quit if it is not equal to 4
  if (argc != 4)
    { cout << "Usage: programname host 79 username" << endl;
      exit(1);
    }
  if ((socketNO = socket(AF_INET,SOCK_STREAM,0)) < 0) /* get a mailbox  */
    { perror("socket error \n");
      exit(1);
    }
  if (connectToHost(socketNO, argv[1], atoi(argv[2]) ) < 0 )
    { perror ("connect error \n");
      exit(1);
    }
  // Send the information (request) to the server
  if (write(socketNO, argv[3], strlen(argv[3])) < 0)
    { perror ("write error \n");
      exit(1);
    }
  // send the end of line character
  if (write(socketNO, "\n", 1) < 0)
    { perror ("write error \n");
      exit(1);
    }
    /* receive from server the entire response */
    n = read(socketNO, FromServer, bufferSize);
    while (n > 0)
      {  /* write on terminal one character at a time */
         for (int index = 0; index < n; index++)
                 cout << FromServer[index];

         n = read(socketNO, FromServer, bufferSize);
      }  /* end while */
    cout << endl;

    if ( close(socketNO) < 0)
       cout << "Error closing connection.";
    return (0);
  }
```

## 6. Concluding Remarks

In this paper, we described four projects suitable for a course in operating systems and data communications. We believe that projects like these which provide hands-on experience are necessary to reinforce theoretical concepts. We feel that the overall outcomes of these projects were both interesting and beneficial to the students. The projects are simple but they provide a new experience to the students. For brevity we did not include the complete code for the instructor supplied routines, and the student solutions. They can be obtained from the authors.

## Bibliography

**1.** Dadfar, Mohammad B. , Brachtl, Michael, and Ramakrishnan, Sub, "A Comparison of Common Processor Scheduling Algorithms" ASEE 2002 Annual Conference, 1520-01

**2.** Dadfar, Mohammad B. , Francis, Jeffrey, and Ramakrishnan, Sub, "A Core Computer Science Course to Introduce Innovative Systems Concepts" The American Society for Engineering Education (ASEE) J. Computers in Education, Vol. VIII, No. 1, January-March 1998, pp. 27-33.

**3.** Ramakrishnan, Sub and Dadfar, Mohammad B., "Client/Server Communication Concepts for a Data Communications Course," ASEE 1997 Annual Conference, 2520-02.

**MOHAMMAD B. DADFAR**
Mohammad B. Dadfar is an Associate Professor in the Computer Science Department at Bowling Green State University. His research interests include Computer Extension and Analysis of Perturbation Series, Scheduling Algorithms, and Computers in Education. He currently teaches undergraduate and graduate courses in data communications, operating systems, and computer algorithms. He is a member of ACM and ASEE.

**SUB RAMAKRISHNAN**
Sub Ramakrishnan is a Professor of Computer Science at Bowling Green State University. From 1985-1987, he held a visiting appointment with the Department of Computing Science, University of Alberta, Edmonton, Alberta. Dr. Ramakrishnan's research interests include distributed computing, performance evaluation, parallel simulation, and fault-tolerant systems.