CONCURRENT/DISTRIBUTED PROGRAMMING ILLUSTRATED USING THE DINING PHILOSOPHERS PROBLEM*

S. Krishnaprasad
Mathematical, Computing, and Information Sciences
Jacksonville State University
Jacksonville, AL 36265
Email: skp@jsucc.jsu.edu

ABSTRACT

Many paradigms and techniques of distributed computing and concurrent processing have matured and are popular. A quick and effective way to assimilate the basics of these principles is to illustrate them using a single but simple example, avoiding language-specific and tool-specific details and syntax. In this paper we consider the famous dining philosophers problem and indicate several solutions to it based on the various concurrent/distributed computing concepts.

1. INTRODUCTION

Advances in hardware, networking, and software have led to widespread use of concurrent and distributed processing techniques. Since many of these technologies have matured it is important for software professionals and computer science students to have basic understanding of these techniques. One of the ways to learn about a variety of techniques is to see them all in a given context. We may consider this as a "one stop shopping" approach of presenting the concepts. In concurrent and distributed computing the following are popular and mature techniques for synchronization and mutual exclusion: semaphores, monitors, message passing, remote procedure call and rendezvous. An excellent introduction and discussion of

---

these topics can be found in the text by Andrews [1]. The intricacies of these methods can be quickly assimilated if they are all presented with a single application example. In this paper we illustrate these techniques as applied to the dining philosophers problem.

The dining philosophers problem (DPP) [4] is one of the popular examples used in operating systems course to illustrate the basic concepts of mutual exclusion, synchronization, and deadlock. This problem was first introduced by Dijkstra [3] in which five philosophers are seated around a dinner table. In front of each philosopher is a plate of spaghetti. A philosopher needs two forks (left and right) to eat. But there are only five forks on the table placed in between each of the five plates. Each philosopher can only do two things: think or eat. No forks are needed for thinking. A philosopher has to acquire both the forks before the eating process. After finishing eating, a philosopher places forks back on the table and resumes thinking. Note that at most two philosophers may eat concurrently. Chandy and Misra [2] have a given more formal treatment of this problem.

Section 2 illustrates a solution to the DPP using semaphores and section 3 shows a solution based on monitors. A solution based on message passing paradigm is presented in section 4. Section 5 shows how to use remote procedure call technique to implement the DPP. Finally, section 6 presents a solution based on the rendezvous method. Conclusions follow in section 7. We highly recommend the book by Andrews [1] for the detailed and lucid exposition of the concepts and techniques related to multithreaded, parallel, and distributed computing.

2. USING SEMAPHORES

Semaphore is a kind of shared variable which is manipulated by two special operations known as P and V. Semaphore variables assume non-negative integer values. Suppose x is a semaphore variable. Then, the operation V(x) increments the value of x by one *atomically*. P(x) operation, also executed atomically, has the following effect: the process executing the P(x) operation waits until the value of x is positive; then it decrements x by one and continues. Semaphore is an important low-level abstraction to implement mutual exclusion and synchronization amongst concurrent processes.

In the DPP each fork is a shared resource. We may represent the forks by an array of five semaphores. Acquiring a fork uses a P operation and releasing a fork uses a V operation on appropriate semaphores. Philosophers 0 to 3 grab the left fork first and then the right fork. Philosopher 4 grabs the right fork first and then the left fork. This is done to avoid deadlock. The necessary mutual exclusion and synchronization actions based on semaphores are shown in the following solution:

```
// DPP solution based on semaphores

Semaphore fork[5] = { 1, 1,1 ,1, 1};
// an array of five semaphore variables, all initialized to 1

process philosopher [ k = 0 to 3 ]   // concurrent processes of first four philosophers
Begin
  Do
```

```
        Think;
        //now hungry, ready to eat
            P(fork[ k ]);            // grab the left fork
        P(fork[ k+1 ]);              // grab the right fork
        Eat the spaghetti;
        V(fork[ k ]);                // release the left fork
        V(fork[ k+1 ]);  //release the right fork
            Until death
    End

    process philosopher [ 4 ]   // concurrent process of the last philosopher
            Begin
        Do
            Think;
            //now hungry, ready to eat
                P(fork[ 0 ]);    // grab the right fork
            P(fork[ 4 ]);        // grab the left fork
            Eat the spaghetti;
            V(fork[ 0 ]);        // release the right fork
            V(fork[ 4 ]);        //release the left fork
                Until death
        End
```

3. USING MONITORS

Monitors are class-like abstractions used to encapsulate shared data and a set of operations (procedures) to manipulate those data. Procedures inside a monitor are executed mutually exclusively. Thus, when concurrent processes access monitor's procedures, mutual exclusion is implicitly guaranteed. But, condition synchronization need to be explicitly programmed inside the monitor using condition variables. A condition variable is used to delay a process until some monitor's state is met at which time the waiting process will be awakened. Typically a queue of waiting processes is associated with each condition variable. Primitives like wait() and signal() are used for coding synchronization aspects. A monitor-based solution for the DPP follows:

```
// DPP solution based on monitors
Monitor Dining_Controller
    Begin
      Cond ForkReady[5]; // condition variables for synchronization
      Boolean Fork[5] = {true, true, true, true, true}; // availability status of each fork

    Procedure get _forks ( pid ) // pid is the philosopher id #: 0 to 4
    Begin
      int left = pid; int right = (pid +1 ) mod 5;
      //grant the left fork
      if (NOT Fork[left] ) wait(ForkReady[left]); // enqueue in the condition variable queue
      Fork[left] = false;
      //grant the right fork
      if (NOT Fork[right] ) wait(ForkReady[right]); // enqueue in the condition variable queue
      Fork[right] = false;
    End
```

```
        Procedure release_forks(pid)
        Begin
         int left = pid; int right = (pid +1 ) mod 5;
         // release the left fork
         if (Empty(ForkReady[left]) ) // no one is waiting for this fork
             Fork[left] = true;
         else
             Signal (ForkReady[left]); // awaken a  process waiting on this fork
        // release the right fork
        if (Empty(ForkReady[right]) ) // no one is waiting for this fork
             Fork[right] = true;
        else
            Signal (ForkReady[right]); // awaken a  process waiting on this fork
    End

End // Dining_Controller monitor

Process Philosopher [k=0 to 4] // the five philosopher clients
  Begin
    Do
      Think;
      Dining_Controller.get_forks(k); // client requests for forks via the monitor
      Eat spaghetti;
      Dining_Controller.release_forks(k); // client releases its forks via the monitor
    Until death
 End
```

4. USING MESSAGE PASSING

In loosely coupled multiprocessing systems and distributed systems, processes communicate using explicit message passing via a communication network. A channel abstraction is employed to provide this communication path between processes. A channel represents a queue of messages that have been sent but not yet received. A process sends a message to a channel using a send primitive: send channel_ID (parameter list). A process receives a message from a channel using a receive primitive: receive channel_ID (parameter list). A message typically includes client ID, the kind of operation being requested, and any parameters/results. A solution for DPP using message passing is given next.

```
// DPP solution based on message passing

Type op_kind enum (ACQ, REL); // two kinds of operations: acquire and release forks
Channel request (int cid, op_kind kind); // request channel
Channel reply[5]( ); // an array of five reply channels

Process Dining_Server
  Begin
     Boolean fork[5] = {true, true, true, true, true}; // indicates fork availability
     Queue WQ; // queue for clients waiting on forks
     int left, right, cid; op_kind kind;
     DO
       Receive request (cid, kind); // server receives client request via  message passing
       left = cid; right = (cid +1) mod 5;
```

```
      If ( kind == ACQ ) // requested operation is acquire forks
        If (fork[left] && fork[right] )
           Send reply[cid] ( ); // server informs client via  message passing
        Else
           Enqueue(WQ, cid); // put the client in queue until forks are available
      Else // requested operation is release forks
        {
          fork[left] = true; fork[right] = true;
          Send reply[cid] ( ); // server acknowledges client via  message passing
          Assign_Forks_To_A_Waiting_Client():/assumed procedure in the server
        }
    WHILE (true);
  End // Dining_Server

Process Philosopher [k=0 to 4] // the five clients
 Begin
   Do
    Think;
    Send request ( k, ACQ ); // client requests for forks from the server via message passing
    Receive reply[k] ( ); // client receives acknowledgement from the server via message passing
    Eat spaghetti;
    Send request ( k, REL ); // client requests server to release its forks
    Receive reply [k] ( ); // client receives acknowledgement from the server
   Until death
 End
```

5. USING REMOTE PROCEDURE CALL

In a truly distributed computing environment, processes residing in different address spaces may communicate. The remote procedure call (RPC) abstraction allows for inter-process communication via remote procedure calls. In this technique, distributed program modules consist of exportable operations (callable from other modules), local operations, and other active processes. For each remote procedure call a new server process is created in the called module. Parameters are passed between the calling and called modules using implicit message passing. The calling process delays until the remote call is serviced and acknowledged by a reply message. At this point the server process terminates. The sending and receiving of parameters using message passing are implicit (that is, no need to explicitly program the message passing). Processes inside the module may execute concurrently and share variables. This requires explicit programming of the mutual exclusion and synchronization aspects. Thus, the module design might get quite complex unlike the case of monitor design. What follows is an RPC-based solution to the DPP.

```
//DPP solution based on RPC

Module Dining_table
Begin

  Operation get_forks (int); // exportable operation
  Operation rel_forks (int); // exportable operation
```

```
  Boolean eating[5] = {false, false, false, false, false};
  Semaphore mutex; // for mutually exclusive access to eating array

  Queue WQ; // queue of waiting clients
  Semaphore MQ = 1; // for mutually exclusive access to WQ queue

 Procedure get_forks (int k)
 Begin
    P(mutex);
    If (NOT eating[k] && NOT eating[ (k+1) mod 5] )
      { eating[k] = true; V(mutex); }
    else
      { P(MQ); Enqueue(WQ, k); V(MQ); V(mutex); }
 End

 Procedure rel_forks(int k)
   Begin
      P(mutex);
      Eating[k] = false;
      V(mutex);
      Assign_Forks-To_A_Waiting_Client();//assumed procedure inside this module
   End

End // Dining_Table module

Process Philosopher [k=0 to 4] // the five clients
 Begin
    Do
     Think;
     CALL Dining_Table.get_forks(k); // client does RPC to acquire forks
     Eat spaghetti;
     CALL Dining_Table.release_forks(k); // client does RPC to release forks
   Until death
 End
```

6. USING RENDEZVOUS

Rendezvous provides implicit communication and synchronization thus greatly simplifying the design of distributed modules. Similar to an RPC module, we define exportable operations inside the rendezvous module. A client process can call an exportable operation of a remote module by a remote call. Unlike the case of RPC, here we do not create a new server process. Instead an existing server process of the remote module actually waits for, accepts, and executes instances of remote calls. Remote operations are executed one at a time. Synchronization is achieved by specifying condition expressions as part of the call-accepting step. We say that the server process of the called module performs a rendezvous with the calling process by executing an accept statement for a specific operation. A solution based on rendezvous follows:

```
// DPP solution based on rendezvous

Module Dining_Table
```

```
  Operation get_forks (int); // exportable operations
  Operation rel_forks (int); // exportable operations

   Process Table_Waiter // active server process
   Begin
     Boolean eating[5] = { false, false, false, false, false };
     DO
      Select
        Accept get_forks(k) && NOT eating[k] && NOT eating[(k+1) mod 5]  // rendezvous
-> eating[k]=true;
       OR
        Accept rel_forks(k) -> eating[k] = false; // rendezvous point
       End Select;
      UNTIL (true);
   End Table_Waiter
End // Dining_Table module

// code for client processes are similar to that given in RPC example
```

7. CONCLUSIONS

In this paper we have considered many of the basic principles behind concurrent/distributed computing and discussed how to apply them in the context of the dining philosophers problem. This kind of "single stop shopping" approach often expedites the learning process by seeing all of the different techniques with a single example. We encourage the readers to substitute the DPP with other practical problems, like bounded-buffer problem, readers-writers problem, and resource sharing problem. To simplify the presentation, we have used pseudo-code to illustrate the programs/procedures without the language or tool-specific details/syntax.

REFERENCES

1. Andrews, G. R., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
2. Chandy, K. M., Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
3. Dijkstra, E. W., *Cooperating sequential processes*, in *Programming Languages*, Academic Press, 1968.
4. Nutt, G., *Operating Systems: A Modern Perspective*, Addison-Wesley, 1990.