Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

LINDA AROUSES A SLEEPING BARBER

John H. Reynolds

Computer Science Department
Mary Washington College
1301 College Avenue
Fredericksburg, VA 22401, U.S.A.

ABSTRACT

This paper presents an approach that gives students insights into parallelism and exposure to discrete-event simulation techniques without requiring that they have formal courses in either. I apply the rather curious Linda coordination model to the classic Sleeping Barber Problem used frequently to illustrate inter-process communication activities in operating system courses. Normally, customers seeking haircuts are represented as processes spawned as faceless entities with no regard to inter-arrival times or proper ordering of departures for those who get cuts. This paper uses elementary discrete-event simulation techniques to introduce this sought for realism while preserving the original motivation of using the Sleeping Barber to demonstrate process concurrency.

1 INTRODUCTION

Computer Science majors at MWC may choose as electives **Simulation Techniques** and/or **Parallel Processing**. Since these electives are not offered every year, students may miss the opportunity to take them or defer to more appealing electives. But required courses like **Operating Systems** can provide opportunities to illustrate discrete-event simulation techniques normally seen in specialized courses.

I use the C instantiation of Linda, known as **C-Linda**, in my parallel processing course as a coordination model that implements parallelism via a logically shared memory across a network of workstations. One can teach the Linda concept quickly by confining discussion to just four basic operations. This makes it an ideal supplement for teaching and exploring concurrency concepts in operating systems.

One can introduce inter-arrival delays separating customers entering the barbershop as well as service delays by using OS sleep functions like **usleep** available in UNIX.

In this paper I provide an overview of Linda and highlight the implementation details I used to give the Sleeping Barber example a more realistic feel.

2 WHAT IS LINDA?

In a recent paper (Reynolds 2000) I discuss my initial experiences with Linda after using it to implement the Bucket Sort algorithm suggested by Wilkinson and Allen (1999). (Code at <http://www1.mwc.edu/~bigjr>) In that paper I provide a thumbnail sketch of the Linda paradigm, which I repeat here.

Linda is a coordination language used to develop parallel applications (Carriero and Gelernter 1991). When combined with C one has **C-Linda**, a complete parallel programming language. In 1997 MWC purchased C-Linda for $1,000 from Scientific Computing Associates (<http://www.lindaspaces.com/>) who offer a FORTRAN version as well. There are also free implementations available based on variations involving Lisp, Prolog, and Pascal.

C-Linda runs on MWC's computer science network of five HP workstation/servers under the control of HP-UX. They host 31 Entria/Envizex terminals while one server distributes file systems to the other nodes under the control of the NFS (Network File System). Since all architectures are of HP's proprietary PA-RISC design, a program compiled on one machine will execute on all of the others, albeit at different speeds. This network is ideal for parallel processing; there is no need to do separate compilations to accommodate disparate architectures.

The Linda model provides a ***virtual shared memory*** (VSM) whose basic addressable unit is the ***tuple***. The VSM, often called ***tuple space***, is logically shared by the processes (frequently called ***workers***). A tuple is a sequence of up to 16 typed fields enclosed in parentheses. For example:

```
("College Avenue", 1301, "campus")
("CollegeAveue",?street_number,?edifice_type)
("worker", id, SortMyBucket(id))
```

It is standard practice to use the first field in a tuple for "documentation." A field preceded by a "?" is known as a

placeholder or ***formal*** field, while those with values are called ***actuals***. The first two tuples are examples of ***data*** tuples. The last one could be a ***data*** or ***live*** tuple depending on how it is created. Processes can interface with VSM using four basic operations:

- **out**(tuple): Inserts data into VSM after it resolves all fields of its tuple to actual values. Thus, **out("worker", id, SortMyBucket(id))** results in a data tuple placed in VSM after SortMyBucket is "replaced" by the value it returns. On the other hand, **out("data", my_id, MyBucketSize)** results in a data tuple whose fields contain the current values of the variables.
- **in**(tuple): Does destructive read of data from VSM by removing, atomically, the first tuple found that matches the operation's tuple (sometimes referred to as the ***anti-tuple*** or ***template***). If more than one tuple satisfies a request, an arbitrary, non-deterministic selection is made. If no match is found, the requesting process blocks until a matching tuple is placed in VSM. Thus, **in("data", my_id, MyBucketSize)** is looking for a data tuple whose field values match the current field values of the requesting tuple. The request, **in("data", ?my_id, ?MyBucketSize)**, implies that the first data tuple found in VSM whose first field is "data", and the last two matching the field types of the template, will be removed.
- **rd**(tuple): Functions identically to **in** except that it does not remove the matching tuple from tuple space. That is, it does a non-destructive read of data from VSM.
- **eval**(tuple): Creates a process tuple consisting of the fields specified as its argument and then returns. A child process performs the evaluation for each field. During the evaluation process the tuple is not accessible until it becomes a passive **data tuple**. Thus, it is a **live tuple** until the field-evaluating processes terminate. Within a loop, one could spawn many **eval("worker", id, SortMyBucket(id))** tuples, which will execute in parallel. The difference between out and eval is that the former returns *after* the data tuple is placed in VSM while the latter returns *before* the data tuple materializes in VSM.

C-Linda offers a Code Development System for creating and verifying code. Solutions run on a uniprocessor simulator; hence no speedup can be observed. By changing an environment variable and recompiling the source code, one can run the application across a network of workstations. Finally, only C functions can call Linda functions. C++ functions can be intermingled as long as they are devoid of Linda constructs.

3 THE SLEEPING BARBER PROBLEM

A class of problems in the study of operating systems deals with inter-process communication. These problems require techniques that provide cooperative solutions when competing processes want to modify the same variable within a critical section (CS). Tanenbaum (1992) offers extensive coverage of this subject including the approach by Dijkstra (1965) where mutual exclusion is controlled by semaphores that can be raised (V operation) and lowered (P operation) atomically.

Tanenbaum (1992) also provides a pseudo-C solution to the classical Sleeping Barber Problem proposed by Dijkstra (1965) that has customer processes interfacing with a barber process who, when not busy, sleeps in his chair. An arriving customer finding a sleeping barber, wakes him up, and gets a haircut. New arrivals encountering a busy barber take a seat in the n-chair waiting room. If the waiting room is full, the customer simply leaves.

Figures 1 and 2 provide an outline of Tanenbaum's barber and customer processes, respectively. Figure 3 includes the two functions, called in the previous figures, used in a solution by Hartley (1995). The semaphores **mutex** and **barber** are binary with the former "raised (1)" and the latter "lowered (0)" initially. The counting semaphores **customers** and **cutting** are 0 initially.

```
while (1) {
  P(customers); //Sleep if customers = 0
  P(mutex);     //Get access to waiting room
  waiting--;    //Remove a customer
  V(barber);    //Barber ready to cut hair
  V(mutex);     //Release access to waiting
  cut_hair();   //Cut hair (outside of CS)
}
```

Figure 1: Outline of the Barber Process

```
P(mutex);       //Enter critical section
if (waiting < CHAIRS){//Full waiting room?
  waiting++;    //Admit customer
  V(customers); //Wake up barber, possibly
  V(mutex);     //Release access to waiting
  P(barber);    //Sleep if barber busy
  get_haircut;} //Get in chair; be serviced
else {//Shop is full; do not wait
  V(mutex);     //Release access to waiting
}
```

Figure 2: Outline of the Customer Process

```
void cut_hair()
{
    P(cutting);
} //end of cut_hair

void get_haircut()
{
    V(cutting);
} //end of get_haircut
```

Figure 3: Skeletal forms of cut_hair and get_haircut

4 ADDING REALISM

The solution by Hartley (1995) employs the SR programming language, a high-level Pascal-like language for writing concurrent/parallel programs. He also monitors his program's behavior by printing milestones such as the id's of customer processes in the order of arrival as well as the id's of customers in the order of departure. Although I consider his solution correct for the approach taken, his example, involving a waiting room with three chairs, shows six customers arriving in numerical order as one would expect, but the first two customers getting haircuts are the first and fourth arrivals, respectively.

One could argue that when customer processes are spawned they represent templates of customers who become *real* when get_haircut is called in Figure 2 above. Thus, the fourth customer, as a template, is *really* the second customer to receive the haircut. A better approach is to accommodate the fact that preserving first-come first-served ordering is jeopardized when customer processes block before calling the get_haircut function. I see problems like the Sleeping Barber as an opportunity to expose students to discrete-event simulation techniques. In my solution I spawn customers around an exponentially distributed inter-arrival time centered on an input mean. These computed times are used to cause actual delays through calls to the UNIX **usleep** function. In addition, I determine haircut times randomly distributed between minimum and maximum service times. These too are used to build in operating system delays within get_haircut. Before providing a few details about my solution in the next section, I want to illustrate the prior point made when one disregards customer servicing in the order of arrival. The following output was generated by a modified version of my final solution:

```
This program simulates the classic Sleeping Bar-
ber problem. The shop has one barber and a fixed-
size waiting room. Arriving customers finding a
full waiting room simply depart.

Type in mean for customer interarrival times: 10
Type in minimum and maximum haircut times: 10 20
Number of arrivals to attempt (must be > 0)? 10
How many chairs in waiting room (must be > 0)? 3


Summary of Input for This Setup
*******************************
Mean interarrival time = 10 units
Minimum haircut time = 10 units
Maximum haircut time = 20 units
Number of chairs in waiting room = 3
Number of arrivals attempted = 10

Monitoring of transactions follows:

    Barber ready for first customer of the day
    Customer 1 in waiting room. Total waiting = 1
    Barber has next customer. Total waiting = 0
    Customer 2 in waiting room. Total waiting = 1
    Customer 3 in waiting room. Total waiting = 2
    Customer 4 in waiting room. Total waiting = 3
    Customer 5 turned away; full waiting room
    Customer 6 turned away; full waiting room
    Customer 7 turned away; full waiting room
    Customer 8 turned away; full waiting room
    Customer 1 finished haircut in 18 time units
    Barber has next customer. Total waiting = 2
    Customer 9 in waiting room. Total waiting = 3
    Customer 10 turned away; full waiting room
    Customer 4 finished haircut in 12 time units
    Barber has next customer. Total waiting = 2
    Customer 9 finished haircut in 12 time units
    Barber has next customer. Total waiting = 1
    Customer 3 finished haircut in 14 time units
    Barber has next customer. Total waiting = 0
    Customer 2 finished haircut in 12 time units
    Barber is finished for the day
```

It is evident that realism goes out the window when customers arriving in numerical order, who get haircuts, leave in the order 1, 4, 9, 3 and 2!

5 C-LINDA SOLUTION

Some of the implementation details are offered here to illustrate how one represents concepts like binary and counting semaphores in C-Linda. Readers will find the complete code at <http://www1.mwc.edu/~bigjr>.

The **eval** operation dispatches the barber and customer processes:

```
eval("Barber", Barber());
eval("Customer", Customer(cut_time);
```

The customers are dispatched within a loop. The UNIX **usleep** function uses exponentially distributed inter-arrival times as an argument to provide delays between dispatches. Notice also that a randomly calculated service time is sent as an argument to the Customer process. Of course, this time becomes moot if a customer can't be admitted because of a full waiting room.

Once the processes die, the **live tuples** become **data tuples** and thus, can be withdrawn from the VSM:

```
for (i = 1; i <= TotalCust; i++)
    in("Customer", ?done);
in("Barber", ?done);
printf(" Barber is finished for the day\n");
```

The P and V operations are easy to implement on binary semaphores. For example:

```
in("mutex", 1);   /* P(mutex) */
out("mutex", 0);

in("mutex", 0);   /* V(mutex) */
out("mutex", 1);
```

Counting semaphores are difficult if not impossible to implement in C-Linda because there is no easy way to simulate atomic updates. However, one can get the job

done by using multiple tuples to represent each value of the count. For example, a literal translation of P(customers) in Figure 1 and V(customers) in Figure 2 would be, in its simplest form:

```
in("customers");   /* P(customers)  */
out("customers");  /* V(customers)  */
```

Thus, if five customers are accepted for haircuts, the V operation will place five copies of the ("customers")-tuple into VSM. On the other hand, the P operation draws, in a non-deterministic way, a tuple representing a "faceless" customer. This approach explains why the out-of-order customer departures demonstrated previously can occur. One must also build in a mechanism that prevents deadlock when the barber gives the last haircut and no more tuples are available to draw. Otherwise, the barber will block forever on its **in("customers")** operation.

To keep track of customers I maintain a counting tuple, CustServed, whose values give unique identities to the "customers" tuples. Likewise, the barber maintains a local variable, my_load, that guarantees a withdrawal of the correct "customers" tuple whose second coordinate matches the current value of my_load. Thus, my P and V operations become:

```
in("customers", my_load);      /* P */
out("customers", CustServed); /* V */
```

I should mention that the value of CustServed is not the same as the id assigned to a customer upon arrival. It is strictly a count of customers admitted to the waiting room. The correlation of waiting room arrival order to the customer identifier, my_id, is handled when the call to get_haircut is made.

The CustServed tuple also provides the values used to control orderly access to the get_haircut function. During initialization activities, the ("FCFS", 1) tuple is placed in VSM. It is used to guarantee that no "line jumping" occurs when customers, who block before calling get_haircut, are usurped by later arriving customers who are not impeded. The code is as follows:

```
in("FCFS", CustServed);
in("barber", 1);  /* P(barber) */
out("barber", 0);
get_haircut(CustServed, my_id, cut_time);
out("FCFS", CustServed+1);
```

To complete the general picture I include my code for the two functions in Figure 3.

```
void cut_hair(CurrentCust)
     int CurrentCust;
{
/* Barber cutting hair */
in("cutting", CurrentCust); / *P(cutting) */
/* Barber finished with this customer */
} /* end of cut_hair */
```

```
void get_haircut(next, CustNum, cut_time)
     int next, CustNum, cut_time;
{
usleep(10000*cut_time);
printf(" Customer %d finished haircut in %d",
          CustNum, cut_time);
printf(" time units\n");
out("cutting", next); /* V(cutting) */
}/* end of get_haircut  */
```

The output below illustrates the effects of a one-chair waiting room. As expected, many arrivals go home without haircuts.

```
Monitoring of transactions follows:

    Barber ready for first customer of the day
    Customer 1 in waiting room. Total waiting = 1
    Barber has next customer. Total waiting = 0
    Customer 2 in waiting room. Total waiting = 1
    Customer 3 turned away; full waiting room
    Customer 4 turned away; full waiting room
    Customer 5 turned away; full waiting room
    Customer 6 turned away; full waiting room
    Customer 7 turned away; full waiting room
    Customer 8 turned away; full waiting room
    Customer 1 finished haircut in 18 time units
    Barber has next customer. Total waiting = 0
    Customer 9 in waiting room. Total waiting = 1
    Customer 10 turned away; full waiting room
    Customer 2 finished haircut in 12 time units
    Barber has next customer. Total waiting = 0
    Customer 9 finished haircut in 12 time units
    Barber is finished for the day
```

Finally, a repeat of the out-of-order example under the same input conditions produces serviced customers departing in the correct order.

```
Monitoring of transactions follows:

    Barber ready for first customer of the day
    Customer 1 in waiting room. Total waiting = 1
    Barber has next customer. Total waiting = 0
    Customer 2 in waiting room. Total waiting = 1
    Customer 3 in waiting room. Total waiting = 2
    Customer 4 in waiting room. Total waiting = 3
    Customer 5 turned away; full waiting room
    Customer 6 turned away; full waiting room
    Customer 7 turned away; full waiting room
    Customer 8 turned away; full waiting room
    Customer 1 finished haircut in 18 time units
    Barber has next customer. Total waiting = 2
    Customer 9 in waiting room. Total waiting = 3
    Customer 10 turned away; full waiting room
    Customer 2 finished haircut in 12 time units
    Barber has next customer. Total waiting = 2
    Customer 3 finished haircut in 14 time units
    Barber has next customer. Total waiting = 1
    Customer 4 finished haircut in 12 time units
    Barber has next customer. Total waiting = 0
    Customer 9 finished haircut in 12 time units
    Barber is finished for the day
```

It took several attempts to reproduce the same customers getting haircuts as the out-of-order example previously. For example, one of the iterations had six customers get-

ting haircuts in the order 1, 2, 3, 4, 6, and 8. Variations like this can be expected because one cannot predict when processes exhaust time slices and block. Also, we are not concerned with internal scheduling strategies used to reschedule processes for another time slice or after blocking for I/O. In fact, one can get several different outcomes with the out-of-order example as well.

6 CONCLUSION

Much more could be done with the program I have described. The results shown above were generated on a uniprocessor under C-Linda's Code Development System. However, with some minor modifications and a recompilation, one could run the same program on a large network of workstations where each node represents a single customer. One could also experiment with a shop housing more than one barber. Finally, one could create "departure" tuples of customers that include not only arrival and haircut times but also departure and waiting times. This would allow a tabular printout of all activities after the barber process shuts down.

I have attempted to show that by infusing a standard operating systems course with ideas from other topics in computer science, one can provide students with exposure they could easily miss in a crowded curriculum. This approach could also serve to whet appetites for electives they haven't considered.

REFERENCES

Carriero, N. and D. Gelernter 1991. *How to Write Parallel Programs*. Cambridge, MA: MIT Press.

Dijkstra, E.W. 1965. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands. Reprinted in Genuys (1968), 43-112.

Hartley, S. J. 1995. *Operating Systems Programming*. New York, NY: Oxford University Press.

Reynolds, J. H. 2000. Two Shared-Memory Tools: SMS and Linda. In *Proceedings of the Sixteenth Annual Eastern Small College Computing Conference* 16: 3-16.

Tanenbaum, A. S. 1992. *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice Hall.

Wilkinson, B. and M. Allen 1999. Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers. Upper Saddle River, NJ: Prentice Hall.

AUTHOR BIOGRAPHY

JOHN H. REYNOLDS is a Professor of Computer Science at Mary Washington College where he has taught since 1983. His current interests are parallel processing and operating systems. Prior to coming to MWC he served four years in the U.S. Navy followed by seventeen years of employment at the Naval Surface Warfare Center (Dahlgren, VA) from 1966-1983. At NSWC he worked in geoballistics that required design and implementation of computer simulations, in numerical analysis, in compiler development, in project management, and in personnel training. His e-mail and web addresses are <bigjr@mwc.edu> and <www1.mwc.edu/~bigjr>.