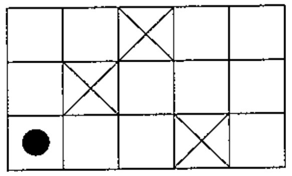
7. (Hoare [32]) Develop an algorithm to simulate the following game. A counter is placed on the lower lefthand square of a rectangular board, some of whose squares are blocked. The counter is required to move to the upper righthand square of the board by a sequence of moves either upward or to the right:



8. Draw a timing diagram similar to the one in Section 8.6 for the case where the accepting task in a rendezvous tries to execute the accept statement before the calling task has called its entry.

9. Suppose that an exception (runtime error) occurs during the execution of the server (accepting) process in a rendezvous. What is a reasonable action to take and why?

10. Compare the monitor solution of the producer–consumer problem in Java given in Section 7.11, with the rendezvous solution in Ada given in Section 8.6. In which solution is there a greater potential for parallel execution?

9 Spaces

Synchronization primitives that are based upon shared memory or synchronous communications have the disadvantage of tight *coupling*, both in time and in space. To communicate using synchronous channels, both the sending and the receiving processes must exist simultaneously, and the identity of the channel must be accessible to both. The Linda model decouples the processes involved in synchronization. It also enables shared data to be *persistent*, that is, data can exist after the termination of the process that created them and used by processes that are activated only later. The properties of loose coupling and persistence exist in the file systems of operating systems, but Linda provides an abstract model that can be implemented in various ways.

Sections 9.1–9.4 present the Linda model. The model is particularly appropriate for the master–worker paradigm, described in Section 9.5. The chapter concludes with a discussion of implementations of the Linda model.

9.1 The Linda model

The model defines a global data structure, for which will we use the metaphor of a giant bulletin board called a *space* on which *notes* can be posted.[1] A note is a sequence of typed elements; for example, ('a',27,false) is a note whose three elements are of types character, integer and boolean, respectively. The first element of a note is a character literal that conveys the intended meaning of a note.

The atomic statements of Linda are:

postnote(v1, v2, ...) This statement creates a note from the values of the parameters and posts it in the space. If there are processes blocked waiting for a note matching this parameter signature, an arbitrary one of them is unblocked.

[1]This text will use terminology consistent with that used in the jBACI concurrency simulator. The original terminology of Linda is described in Section 9.5.

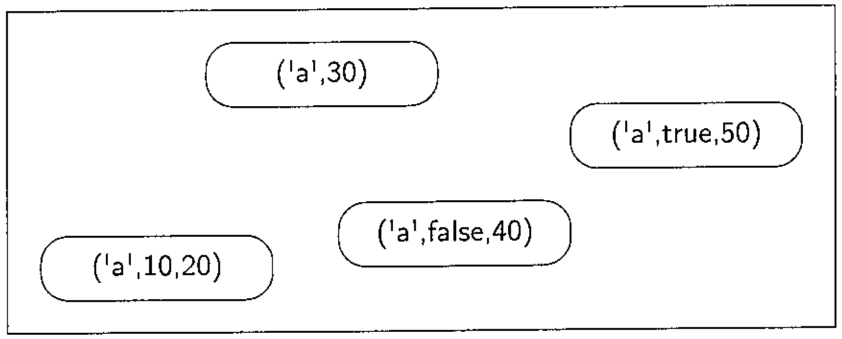
removenote(x1, x2, ...) The parameters must be variables. The statement removes a note that matches the parameter signature from the space and assigns its values to the parameters. If no matching note exists, the process is blocked. If there is more than one matching note, an arbitrary one is removed.

readnote(x1, x2, ...) Like removenote, but it leaves the note in the space.

Suppose that the following statements are executed:

```
postnote('a', 10, 20);
postnote('a', 30);
postnote('a', false , 40);
postnote('a', true, 50);
```

The content of the space is shown in the following diagram:



We must define what it means for the parameter signature of an operation and a node to *match*. The two are defined to match if the following two conditions hold: (a) The sequence of types of the parameters equals the sequence of types of the note elements. (This implies, of course, that the number of parameters is the same as the number of elements of a note.) (b) If a parameter is not a variable, its value must equal the value of the element of the note in the same position. For the space shown in the diagram above, let us trace the result of executing the following statements:

```
    integer m, integer n, boolean b
(1) removenote('b', m, n)
(2) removenote('a', m, n)
(3) readnote('a', m)
(4) removenote('a', n)
(5) removenote('a', m, n)
(6) removenote('a', b, n)
(7) removenote('a', b, m)
(8) postnote('b', 60, 70)
```

The process executing statement (1) blocks, because there is no note in the space matching the first parameter of the statement. Statement (2) matches the note ('a',10,20) and removes it from the space; the value of the second element 10 is assigned to the variable m and 20 is assigned to n. Statement (3) matches, but does not remove, the note ('a',30), assigning 30 to m. Since the note remains in the space it also matches statement (4); the note is removed and 30 is assigned to n. Statement (6) matches both ('a',false,40) and ('a',true,50); one of them is removed, assigning either *false* to b and 40 to n or *true* to b and 50 to n. The other note remains in the space and is removed during the execution of statement (7). Finally, statement (8) posts a note that matches the blocked call in statement (1). The process that executed (1) is unblocked and the note is removed from the space, assigning the values 60 to m and 70 to n.

9.2 Expressiveness of the Linda model

The Linda model combines both data and synchronization in its statements, making it more expressive than semaphores. To implement a general semaphore in Linda, initialize the space by posting a number of notes equal to the initial value of the semaphore:

```
do K times
  postnote('s')
```

removenote is now equivalent to wait and postnote to signal, as can be seen in the following algorithm for the critical section problem:

Algorithm 9.1: Critical section problem in Linda

```
    loop forever
p1:     non-critical section
p2:     removenote('s')
p3:     critical section
p4:     postnote('s')
```

The Linda model only specifies the concepts of spaces, notes and the atomic statements, so there is not much point in asking how to simulate a monitor. The encapsulation aspects of the monitor would have to be supplied by the language in which Linda is embedded.

To simulate sending a value on a channel c, simply post a note with the channel name and the value—postnote('c',value); to simulate receiving the value, remove the note using the same channel name and a variable—removenote('c',var).

What may not be immediately apparent from these examples is the loose coupling and persistence of the system. One process could initialize the system with a few notes and terminate, and only later could another process be initiated and executed to remove the notes and to use the values in the notes in its computation.

9.3 Formal parameters

To motivate the need for formal parameters, let us consider the outline of a client–server algorithm in Linda:

Algorithm 9.2: Client–server algorithm in Linda

	client		server
	constant integer me ← . . .		integer client
	serviceType service		serviceType s
	dataType result, parm		dataType r, p
p1:	service ← // Service requested	q1:	removenote('S', client, s, p)
p2:	postnote('S', me, service, parm)	q2:	r ← do (s, p)
p3:	removenote('R', me, result)	q3:	postnote('R', client, r)

The client process posts a note 'S' containing its ID, the service requested, and the parameters for the service; then it waits to remove a result note 'R' matching its ID and containing the result. The server process waits to remove an 'S' note, performs the service and then posts the result in an 'R' note. It saves the ID of the client to

ensure that the result returns to the client that requested the service. This algorithm works well as long as all servers are able to perform all the services that will be requested. More often, however, different servers provide different services.

The Linda statements described so far are somewhat limited in the way they match parameter signatures with notes; a parameter whose value is a constant expression like a literal matches only that value, while a variable parameter matches *any value* of the same type. More flexibility is obtained with a *formal parameter*, which is a variable parameter that only matches notes containing the *current value* of the variable. We will denote this by appending the symbol = to the variable name.

Suppose that a server can provide one service only; Algorithm 9.2 can be changed so that a server removes only notes requesting that service:

Algorithm 9.3: Specific service

	client		server
	constant integer me ← . . .		integer client
	serviceType service		serviceType s
	dataType result, parm		dataType r, p
p1:	service ← // Service requested	q1:	s ← // Service provided
p2:	postnote('S', me, service, parm)	q2:	removenote('S', client, s=, p)
p3:		q3:	r ← do (s, p)
p4:	removenote('R', me, result)	q4:	postnote('R', client, r)

In the server process, the variable s is specified as a formal parameter so removenote will only match notes whose third element is equal to the current value of the variable.

Formal parameters can be used to implement an infinite buffer:

Algorithm 9.4: Buffering in a space

	producer		consumer
	integer count ← 0		integer count ← 0
	dataType d		dataType d
	loop forever		loop forever
p1:	d ← produce	q1:	removenote('B', count=, d)
p2:	postnote('B', count, d)	q2:	consume(d)
p3:	count ← count + 1	q3:	count ← count + 1

The formal parameter in the consumer ensures that it consumes the values in the order they are produced. We leave it as an exercise to implement a bounded buffer in Linda.

Here is a simulation of the matrix multiplication algorithm with channels:

```
Algorithm 9.5: Multiplier process with channels in Linda
                    integer FirstElement
                    integer North, East, South, West
                    integer Sum, integer SecondElement

    loop forever
p1:     removenote('E', North=, SecondElement)
p2:     removenote('S', East=, Sum)
p3:     Sum ← Sum + FirstElement · SecondElement
p4:     postnote('E', South, SecondElement)
p5:     postnote('S', West, Sum)
```

Notes identified as 'E' contain the *E*lements passed from north to south, while notes identified as 'S' contain the partial *S*ums passed from east to west.

9.4 The master–worker paradigm

The channel algorithm for matrix multiplication has a processor structure that is very rigid, with exactly one processor assigned to each matrix element. It makes no sense to remove one processor for repair or to replace a few processors by faster ones. In Linda, we can write a program that is flexible so that it adapts itself to the amount of processing power available. This is called *load balancing* because we ensure that each processor performs as much computation as possible, regardless of its speed. One process called a *master* posts task notes in the space; other processes called *workers* take task notes from the space and perform the required computation. The master–worker paradigm is very useful in concurrent programming, and is particularly easy to implement with spaces because of their freedom from temporal and spatial constraints.

We demonstrate the master–worker paradigm with the matrix multiplication problem. The master process first initializes the space with the n row vectors and the n column vectors:

```
postnote('A', 1, (1,2,3))
postnote('A', 2, (4,5,6))
postnote('A', 3, (7,8,9))
postnote('B', 1, (1,0,1))
postnote('B', 2, (0,1,0))
postnote('B', 3, (2,2,0))
```

Then it posts n^2 notes of the form ('T', i, j), one for each task. Finally, it waits for the n^2 result notes of the form ('R', i, j, result). A worker process removes a task note, performs the (vector) multiplication and posts a note with the result:

```
Algorithm 9.6: Matrix multiplication in Linda
                      constant integer n ← . . .
              master                         |            worker
    integer i, j, result                     |     integer r, c, result
    integer r, c                             |     integer array[1..n] vec1, vec2
                                             |     loop forever
p1: for i from 1 to n                        | q1:     removenote('T', r, c)
p2:     for j from 1 to n                    | q2:     readnote('A', r=, vec1)
p3:         postnote('T', i, j)              | q3:     readnote('B', c=, vec2)
p4: for i from 1 to n                        | q4:     result ← vec1 · vec2
p5:     for j from 1 to n                    | q5:     postnote('R', r, c, result)
p6:         removenote('R',r,c,result)       | q6:
p7:         print r, c, result               | q7:
```

Note the use of variable parameters r= and c= in the worker processes to ensure that the correct vectors are read. In the master process, statement p6 uses variables r and c so that the loop indices i and j are not overwritten; the notes with the results are removed in an arbitrary order as they are posted.

The code of the algorithm is totally independent of the number of worker processes. As long as a process successfully removes a task note, it contributes to the ongoing computation. Nor is the algorithm sensitive to the relative speeds at which the worker processes are executed. A worker executed by a fast processor will simply complete more tasks during a time period than a worker executed by a slow processor. Furthermore, computation can be dynamically speeded up if the computer architecture enables the addition of processors during the execution of the program; the new processors can begin executing the algorithm for the worker process on tasks that are still posted in the space. Conversely, if a specific processor needs to be removed from the system, the worker process it is executing can be stopped at any time after completing the body of the loop.

Granularity

The master–worker paradigm is quite flexible because we can specify the *granularity* of the task to suit the relative performances of the processors and the communications system. The above algorithm uses a very small granularity where one processor is responsible for computing one result at a time, so the communications overhead is relatively high.

It is easy to modify the algorithm so that it works at any level of granularity. We have added a constant chunk and posted task notes only every chunk'th column. (Assume that n is divisible by chunk.) For example, if n is 100 and *chunk* is 10, then the task notes posted will be: ('T',1,1), ('T',1,11), ..., ('T',1,91), ('T',2,1), ..., ('T',2,91), ..., ('T',100,1), ..., ('T',100,91). The following algorithm shows the changes that must be made in the worker processes:

Algorithm 9.7: Matrix multiplication in Linda with granularity

```
constant integer n ← ...
constant integer chunk ← ...

master
    integer i, j, result
    integer r, c
p1: for i from 1 to n
p2:    for j from 1 to n step by chunk
p3:       postnote('T', i, j)
p4: for i from 1 to n
p5:    for j from 1 to n
p6:       removenote('R',r,c,result)
p7:       print r, c, result

worker
    integer r, c, k, result
    integer array[1..n] vec1, vec2
    loop forever
q1:    removenote('T', r, k)
q2:    readnote('A', r=, vec1)
q3:    for c from k to k+chunk-1
q4:       readnote('B', c=, vec2)
q5:       result ← vec1 · vec2
q6:       postnote('R', r, c, result)
q7:
```

The row vector is read (once per chunk) and then a loop internal to the process reads the column vectors, performs the multiplications and posts the results.

9.5 Implementations of spaces^L

The Linda model has been implemented in both research and commercial settings. For students of concurrency, it is easier to work with implementations that are embedded within one of the languages used in this book. The presentation of the model in this chapter is consistent with its implementation in both the C and Pascal dialects of the jBACI concurrency simulator, and with a package written in Ada. A tuple can contain at most three elements, of which the first is a character, and the other two are integer expressions or variables.

In this section, we briefly describe two commercial systems, C-Linda and JavaSpaces, and then discuss implementations of the model in Java and Promela.

C-Linda

The first implementation of Linda embedded the model in the C language. The resulting system, which has evolved into a commercial product, is called C-Linda. We will briefly review the original presentation of Linda so that you will be able to read the literature on this topic.

A note is called a *tuple* and a space is called a *tuple space.* The names of the statements are out for postnote, in for removenote and rd for readnote. There is an additional statement, eval, that is used primarily for activating processes. There are also non-blocking versions of in and rd, which allow a process to remove or read a tuple and to continue executing if a matching tuple does not exist in the tuple space.

There is no limit on the number or types of parameters; by convention, the first parameter is a string giving the type of the tuple. The notation for formal parameters is ?var.

JavaSpaces

In JavaSpaces, notes are called *entries.* Each type of note is declared in a separate class that implements the Entry interface. For the task notes of the matrix multiplication example, this would be:

```java
public class Task implements Entry {
    public Integer row;
    public Integer col;
    public Task() {
    }
    public Task(int r, int c) {
        row = new Integer(r);
        col = new Integer(c);
    }
}
```

For technical reasons, an entry must have a public constructor with no arguments, even if it has other constructors. All fields must be public so that the JavaSpaces system can access them when comparing entries.

JavaSpaces implements write, take and read operations on the space (or spaces—there may be more than one space in a system). An operation is given a template entry that must match an entry in the space. **null** fields can match any value.

Since an entry is an object of a class, the class may have methods declared within it. A remote processor reading or taking an entry from a space can invoke these methods on the object.

JavaSpaces is integrated into Sun's Jini Network Technology, and it provides features that were not part of the original Linda model. *Leases* enable you to specify a period of time that an entry is to remain in the space; when the time expires, the entry is removed by the space itself. *Distributed events* enable you to specify listener methods that are called when certain events occur in a space. *Transactions* are used to make a system more fault tolerant by grouping space operations so that they either all execute to completion or they leave the space unchanged.

Java

The software archive contains an implementation of the Linda primitives in Java that demonstrates how Linda can be embedded into object-oriented programming.

Here is the outline of the definition of a note:

```
public class Note {
    public String id;
    public Object[] p;

    // Constructor for an array of objects
    public Note (String id, Object[] p) {
        this.id = id;
        if (p != null) this.p = p.clone();
    }
    // Constructor for a single integer
    public Note (String id, int p1) {
        this(id, new Object[]{new Integer(p1)});
    }

    // Accessor for a single integer value
    public int get(int i) {
        return ((Integer)p[i]).intValue();
    }
}
```

A note consists of a String value for its type, together with an array of elements of type Object. To make it easy to use in simple examples, constructors taking parameters of type **int** are included; these must be placed within objects of the

wrapper class Integer in order to store them as elements of type Object. For convenience, the method get retrieves a value of type **int** from the wrapper.

The class Space is not shown here. It is implemented as a Java Vector into which notes may be added and removed. The method that searches for a matching note for removenote or readnote checks that the id fields are equal. A match is then declared if either of the arrays of elements of the two notes is **null**. Otherwise, after ensuring that the arrays are of the same length, the elements are matched one by one with a **null** element matching any value. If a matching note is not found, the thread executes wait; any call to postnote executes **notifyAll** and the blocked threads perform the search again in case the new note matches.

For the matrix multiplication example, the class Worker is declared as a private inner class derived from Thread:

```
private class Worker extends Thread {
  public void run() {
    Note task = new Note("task");
    while (true) {
      Note t = space.removenote(task);
      int row = t.get(0);
      int col = t.get(1);
      Note r = space.readnote(match("a", row));
      Note c = space.readnote(match("b", col));
      int sum = 0;
      for (int i = 1; i <= SIZE; i++)
        sum = sum + r.get(i)*c.get(i);
        space.postnote(new Note("result",row,col,sum));
    }
  }
}
```

The code is straightforward except for the calls to the method match. The call match("a",row) creates a note

```
{ "a", new Integer(row), null, null, null }
```

that will match precisely the "a" note whose first element is the value of row. The entire note is then returned by the method readnote.

```
chan space = [25] of {byte, short, short, short, short};

active[WORKERS] proctype Worker() {
    short row, col, sum, r1, r2, r3, c1, c2, c3;
    do
    :: space ?? 't', row, col, _, _;
       space ?? <'a', eval(row), r1, r2, r3>;
       space ?? <'b', eval(col), c1, c2, c3>;
       sum = r1*c1 + r2*c2 + r3*c3;
       space ! 'r', row, col, sum, 0;
    od;
}
```

Listing 9.1: A Promela program for matrix multiplication

Promela

It is interesting to look at a Linda-like program for matrix multiplication (Listing 9.1), if only because it shows some of the features of programming with channels in Promela. The notes are stored in a space implemented as a channel. Normally, the send operation ch ! values and receive operation ch ? variables treat a channel as a FIFO data structure; the send operation places the data on the tail of the queue, while the receive operation removes the data from the head of the queue. This is reasonable since most communications channels are in fact FIFO structures. Here we use the receive operation ch ?? message which removes a message from the channel only if it matches the template given in the operation. Therefore, if the message template contains some constants, only a message with those values will be removed by the receive operation.

In the program, each message in the channel consists of five fields, a byte field and four short integers. The first receive statement is:

```
space ?? 't', row, col, _, _
```

Since the first element is the constant 't', the operation will remove from the channel only those messages whose first element is 't'; these are the notes containing the tasks to be performed. The other fields are variables so they will be filled in with the values that are in the message. In this case, only the first two fields are of interest; the other two are thrown away by reading them into the anonymous variable _.

The next two operations read the row and column vectors, where notes with 'a' are the row vectors and notes with 'b' are the column vectors:

```
space ?? <'a', eval(row), r1, r2, r3>;
```

The word **eval** is used to indicate that we don't want row to be considered as a variable, but rather as a constant whose value is the current value of the variable. This ensures that the correct row is read. The angled brackets < > are used to indicate that the values of the messages should be read, while the message itself remains on the channel. Of course, this is exactly the functionality needed for readnote. After computing the inner product of the vectors, the result note is posted to the space by sending it to the channel.

The obstacles to implementing Linda programs in Promela include the lack of true data structuring and even more so the restriction that a channel can hold only one data type.

Transition

The constructs of the Linda model are simple, yet they enable the construction of elegant and flexible programs. Even so, implementations of the model are efficient and are used in distributed systems, especially in those attempting to achieve shorter execution times by dividing large computations into many smaller tasks.

With this chapter we leave the presentation of different models of concurrency and focus on algorithms appropriate for distributed systems that communicate by sending and receiving messages.

Exercises

1. Implement a general semaphore in Linda using just one note.
2. Implement an array in Linda.
3. Suppose that the space contains a large number of notes of the form ('v',n). Develop an algorithm in Linda that prints out the maximum value of n in the space.
4. Implement a bounded buffer in Linda.
5. Modify Algorithm 9.6 so that the worker processes terminate.

6. Compare the following algorithm for matrix multiplication with Algorithm 9.6:

Algorithm 9.8: Matrix multiplication in Linda (exercise)

```
constant integer n ← . . .
```

master	worker
integer i, j, result integer r, c	integer i, r, c, result integer array[1..n] vec1, vec2 loop forever
p1: postnote('T', 0)	q1: removenote('T' i)
p2:	q2: if i < (n · n) − 1
p3:	q3: postnote('T', i+1)
p4:	q4: r ← (i / n) + 1
p5:	q5: c ← (i modulo n) + 1
p6: for i from 1 to n	q6: readnote('A', r=, vec1)
p7: for j from 1 to n	q7: readnote('B', c=, vec2)
p8: removenote('R',r,c,result)	q8: result ← vec1 · vec2
p9: print r, c, result	q9: postnote('R', r, c, result)

7. Suppose that one of the worker processes in Algorithm 9.6 or Algorithm 9.8 terminates prematurely. How does that affect the computation?

8. Introduce additional parallelism into the Linda algorithm for matrix multiplication by dividing the computation of the inner products result ← vec1 · vec2 into several tasks.

9. An $n \times n$ *sparse matrix* is one in which the number of nonzero elements is significantly smaller than n^2. Develop an algorithm in Linda for multiplication of sparse matrices.

10. Write the initializing postnote statements so that Algorithm 9.5 will compute the matrix multiplication on the example in Figure 8.1.

10 Distributed Algorithms

In this chapter and the next two, we present algorithms designed for loosely-connected distributed systems that communicate by sending and receiving messages over a communications network.

The algorithms in this chapter are for the critical section problem. (Traditionally, the problem is called distributed mutual exclusion, although mutual exclusion is just one of the correctness requirements.) The first algorithm is the Ricart–Agrawala algorithm, which is based upon the concept of *permissions*: a node wishing to enter its critical section must obtain permission from each of the other nodes. Algorithms based upon *token-passing* can be more efficient, because permission to enter the critical section resides in a token can that easily be passed from one node to another. We present a second algorithm by Ricart and Agrawala that uses token-passing, followed by a more efficient algorithm by Neilsen and Mizuno.

The variety of models studied in distributed systems is large, because you can independently vary assumptions concerning the topology of the network, and the reliability of the channels and the nodes. Therefore, we start with a section defining our model for distributed systems.

10.1 The distributed systems model

We distinguish between *nodes* and *processes*: A node is intended to represent a physically identifiable object like a computer, while the individual computers may be running multiple processes, either by sharing a single processor or on multiple processors. We assume that the internal synchronization among processes in a node is accomplished using shared-memory primitives, while processes in different nodes can communicate only by sending and receiving messages.

In Chapters 10 and 11, we assume that the nodes do not fail. Actually, a somewhat weaker assumption will suffice: A node may fail in the sense of not performing its "own" computation, as long as it continues to execute the processes that send and receive messages as required by the algorithms in these chapters. In Chapter 12 we will present algorithms that are robust under failure of individual nodes.