LINDA

Linda is different, which is why we've put it into a separate chapter. Linda is not a programming language, but a way of extending (in principle) any language to include parallelism[IMP13, IMP14]. It was originally intended for parallel programming rather than operating systems as such, but the topics are obviously closely related. It will work with any language for which one can write procedures which implement its (rather simple) interface, and is most usefully thought of as an extension to the application programmer interface. Its authors have aimed for machine independence, portability, automatically handled interactions between processes, automatic scheduling, and efficient implementation. Processes communicate through a shared memory called tuple space, which we discussed briefly in the *MEMORY MODELS* chapter. By ingenious and elegant trickery in the implementation of access to the tuple space, the Linda manager guarantees mutual exclusion and synchronisation where required. This goes quite some way towards "factoring out" the concurrency issues, as envisaged by Wirth[IMP10] in his intentions for Modula.

Some details are given in the excerpt which follows[IMP15]. Notice the appearance of "Kernel Linda", intended for lower level use than Linda proper.

Linda is a parallel communication mechanism developed by David Gelernter at Yale University. Rather than communicating with messages or through shared memory locations, processes communicate in Linda via a shared data space called tuple space. Tuple space acts something like an associative memory, since tuples are identified by matching on a key rather than using a specific address. There are four fundamental operations on tuple space:

out	place a tuple in tuple space
in	match a tuple and remove it from tuple space
rd	match a tuple and return a copy of it
eval	create an active tuple (a new process)

A tuple is an ordered collection of data items. For example, the tuple

("hello", 42, 3.14)

contains three data items: a string, an integer, and a float. The operation

```
out("hello", 42, 3.14);
```

places this tuple into tuple space. The *out* operation never blocks, and there can be any number of copies of the same tuple in tuple space.

Tuples in tuple space are accessed by matching against a template. For example. the operation

```
int i; float f;
in("hello", ?i, ?f);
```

will match any tuple in tuple space with three fields whose first field is the string "hello," and whose second and third fields are an integer and a float, respectively. In this case, the string "hello" is the key used to find the matching tuple. If a matching tuple is found, the variables *i* and *f* are assigned the corresponding values from the matching tuple, and the tuple is removed from tuple space. If no matching tuple is found, the current process blocks. If the template matches more than one tuple, an arbitrary one is picked. The *rd* operation is like *in*, except that the matched tuple is not removed from tuple space.

The *eval* operation resembles *out*, except that it creates an active tuple. For example, if *fcn* is a function, then

```
eval("hello", fcn(z), true);
```

creates a new process to evaluate *fcn*, which proceeds concurrently with the process that made the *eval* call. When the new process completes, it leaves the resulting passive data tuple in tuple space, just like an *out* operation.

Linda is a high-level tool for parallel programming in that it is not directly based or dependent on a specific hardware mechanism. As with any higher level tool, there will always be some overhead associated with Linda, but it is not excessive (see the sections entitled "Implementing Linda" and "Kernel Linda"). This is similar to higher level languages compared with assembly languages, where any overhead is usually outweighed by the benefits of easier programming and portability. Likewise, the flexibility of Linda can more than make up for any overhead. For example, dynamic load balancing, where the processing load is dynamically adjusted between multiple processors, is rarely used in parallel programs because it is difficult to implement using message passing or semaphores, despite its potential performance improvement; but it is relatively easy to implement using Linda (an example of dynamic load balancing is given in a later section). Linda also makes it easy to experiment with the parallel structure of a program to increase its performance.

Using Linda like a shared-memory model. Linda can be used to model shared memory, but with the advantage that it is easy to avoid unwanted nondeterminism. In the example above, where the shared variable *x* was incremented by multiple processes, a semaphore was used to protect the shared variable from simultaneous access. Using Linda, we can represent a shared variable *x* as a single tuple whose key is the name *x* (we assume that no one else is using *x* as a key); to increment the variable *x* in tuple space, the following operations would be used:

```
in("x", ?i);
out("x", i+1 );
```

Since the *in* operation removes the matching tuple from tuple space, any other process trying to access the value of *x* will block until the tuple is put back by the *out* operation.

Using tuple space as a shared data space is not limited to systems with shared memory; it will also work with distributed-memory computers. This makes programs written in Linda portable between shared-memory and distributed-memory systems.

Using Linda like a distributed-memory model. Linda can also support the message-passing style of programming. In Figure 2, a message is sent using an *out* operation and received using an *in* operation.

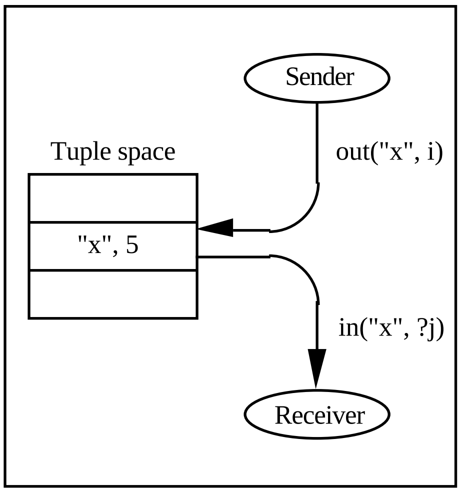

Figure 2. Message passing using Linda.

If the receiver gets ahead of the sender, it will block waiting on the next message. The sender is allowed to run faster than the receiver because the *out* operation is asynchronous (it never blocks). As in any asynchronous message-sending situation, if the sender can get significantly ahead of the receiver, some sort of flow control must be used to avoid overflowing processor memory (an example of using flow control is given in a later section).

In Linda, the ordering of messages is not guaranteed. For example, if one process executes two *out* operations in the following order:

```
out("x", 1);
out("x", 2);
```

then a separate process doing an *in* operation on the key "x" may receive the tuples in either order. The order is often unimportant but if not, an additional field in the tuple can be used to sequence the messages. For example, both the sender and receiver can keep a local sequencing variable, initially set to zero. The *out* and *in* operations in Figure 2 are changed to

```
// in the sender process
out("x", send_sequence++, i);

// in the receiver process
in("x", recv_sequence++, ?j);
```

and the correct order is guaranteed. If there are multiple senders or receivers, then the sequence variables can be shared by storing them in tuple space.

Linda provides decoupling between the sender and receiver of a message in both time and space. Messages are decoupled in time because the *out* operation is asynchronous and because messages travel via tuple space. Since messages can be held in tuple space, sending a message between two programs not running at the same time is even possible. This is crucial in an operating system, where operating system services must be able to communicate with user programs that typically were not written when the operating system was written.

Messages are decoupled in space because individual messages are identified by their contents. Neither the sender nor the receiver needs to know the other's identity or location. This allows communication in Linda to be more dynamic than in systems where the sender and receiver of a message must explicitly rendezvous. In particular, sending and receiving processes can be located on any processor, or even dynamically moved between processors, with no effect on the program. This important property of Linda facilitates writing parallel programs.

Unlike some message-passing systems, Linda is not limited to a single sender and receiver for a message stream. With the number of clients changing over time, this is useful for writing server processes, which receive messages from many different clients. In this case, if the client requires a message in return, it should identify itself in the message to the server so the server can respond to the correct client.

Implementing Linda. At first glance, it might appear that tuple matching requires searching tuple space for each operation, a process that would clearly be too slow for use on a parallel computer. Fortunately, a number of techniques are used to speed up the matching process. The goal of these techniques is to reduce the amount of searching necessary and make the remaining searching as efficient as possible.

The most common approach to making Linda more efficient is to use a preprocessor to reduce or eliminate runtime searching. The preprocessor analyzes a Linda program to determine the patterns of tuple usage. For example, consider a Linda program that contains two processes. The first process contains a single *out* operation:

```
out("x", i);
```

and the second process contains a single *in* operation:

```
in("x", ?j);
```

where *i* and *j* are integer variables. Since the key "*x*" is a constant at compile time, the Linda preprocessor determines that this pair of Linda operations can be implemented as a queue between the variables *i* and *j*. On a shared-memory system, this could be realized as a shared queue protected by a semaphore; on a distributed memory system, this could be realized as an asynchronous message send. Thus, no searching is done, and the most efficient implementation for the target architecture is used without changes to the program's source code. In this case, there is no overhead from the use of Linda.

Using a preprocessor is not always possible if Linda is used, for example, with interpreted languages such as Lisp, Postscript, Smalltalk, or Prolog, since tuple space usage can change while the program is running. Even with compiled languages, determining tuple space usage is not always possible at compile time if the programs that will be communicating cannot be preprocessed at the same time (in particular, for communication between the operating system and a user program).

A common technique used by Linda implementations that do not use a preprocessor is to restrict the key to a single field. In this case, hashing is used so the search can be performed (on average) in constant time. The performance of such an implementation is quite acceptable, since the overhead of a hashing function is reasonably small compared with the cost of data transfer on a typical system. Allowing but a single key in a tuple places some restrictions on Linda, but Linda's main advantages—including portability and decoupling in time and space—remain. For instance, all examples of Linda in this article need only a single key.

A third technique used to make Linda more efficient is to split tuple space into smaller pieces, for example, through the use of multiple tuple spaces; doing so, less space needs to be searched even if searching is required. This technique is also helpful for systems that use hashing, since smaller hash tables can be used.

Kernel Linda

Until now, Linda has been used primarily to support application-level programming. To use Linda for system-level programming, a number of modifications were made. This section discusses those modifications and shows how they benefit system-level programming.

Kernel Linda is a version of Linda designed to be used for system-level programming, as the interprocess communication (IPC) mechanism for a parallel operating system. Kernel Linda was also designed for use as the runtime system for the original Linda. In this case, the Linda preprocessor is used to generate calls to Kernel Linda, rather than to the normal Linda runtime library. This allows programs written in Linda to be integrated into an operating system based on Kernel Linda.

Since Kernel Linda is designed to support communication between operating system processes and user processes, a preprocessor cannot be used to speed up tuple space operations. Consequently, for efficiency, tuples in Kernel Linda are allowed to have only a single key. Except for the single-key restriction, Kernel Linda is a superset of Linda and includes the following extensions:

- multiple tuples spaces,
- the ability to store a tuple space as a field of a tuple, and
- a set of language-independent data types.

In addition, the ordering of tuples in tuple space is guaranteed in Kernel Linda. If one process executes the following *out* operations in the following order:

```
out("x", 1);
out("x", 2);
```

then a separate process doing an *in* on the key "*x*" is guaranteed to receive the values 1 and 2 in that order (assuming that no other Linda operations use the key "*x*" in this tuple space). This avoids the need for extra sequencing keys, and also means that Kernel Linda can be used to transmit a stream of ordered messages from one process to another.

Multiple tuple spaces. Kernel Linda incorporates a unique implementation of multiple tuple spaces that allows the tuple space name to be used somewhat like an additional key in a tuple. In traditional Linda, multiple keys in a tuple are often used to restrict communication to specific data structures; in Kernel Linda, this function can be more efficiently performed using multiple tuple spaces. The provision of multiple tuple spaces removes much of the need for multiple keys.

For example, in the *out* operation

```
// which row in the matrix
int RowNumber;
// data for the row
float data[SIZE];
out("MatrixA", RowNumber, data);
```

the constant "*MatrixA*" identifies a specific data structure, and the identifier *RowNumber* identifies a specific row in the matrix. A specific row of the matrix would be retrieved by searching on the first two keys:

```
int RowNumber = 42;
rd("MatrixA", RowNumber, ?data);
```

In Kernel Linda, the need for multiple keys can be eliminated by placing matrix A in its own tuple space, with the key being the row number. This also reduces the amount of searching required, since the name of the matrix is not used as a key during matching. In Kernel Linda, the above *out* operation would be written (in C) as

```
out(aMatrix, RowNumber, data);
```

where *aMatrix* is the name of a Kernel Linda tuple space. In C++, the special role of the tuple space is syntactically highlighted:

```
aMatrix.out(RowNumber, data);
```

The provision of multiple tuple spaces has other benefits for parallel programming. In a single global tuple space, name conflicts can occur between programs using the same key to identify tuples. Multiple tuple spaces allow communication to be more structured. In a multiapplication environment, some tuple spaces can be local to specific applications, while other tuple spaces can be common to some or all applications. This allows Kernel Linda to support both multiprogramming (separate applications running concurrently) and multiprocessing (single applications running on multiple processors).

Subsequent to our implementation of multiple tuple spaces in Kernel Linda, another design for multiple tuple spaces was done for Linda as part of the design of Linda 3, although it is not the type of system we describe here.

Tuple spaces as tuple fields. Kernel Linda allows a tuple space to be used as a field in a tuple, which allows Kernel Linda to be used as its own name service. This is done by using a name as the key and the tuple space as the value of a tuple in another tuple space. No other naming mechanism is required. A tuple space can be used like a directory in a hierarchical file system, except that arbitrary graphs can be constructed. For example, all tuples spaces associated with an application can be given "names" by storing them in a tuple space, or one process can be given access to a tuple space owned by another application via a common tuple space. This is diagramed in Figure 3, where process P is computing with three matrices: *A*, *B*, and *R*, and a separate process Q has been given access to the *R* (result) tuple space.

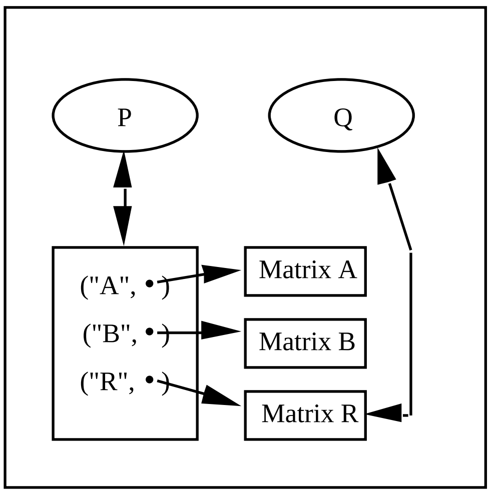

Figure 3. Tuple spaces as fields in a tuple.

Language-independent data types. Linda is traditionally implemented to work within a single language and borrows the data types from that language. Kernel Linda is designed to work with many different languages (both compiled and interpreted), so it provides a set of language-independent data types. For example, this allows a tuple space created by a C++ program to be accessed by a debugger or tuple space browser written in Postscript.

In traditional Linda, all fields in a tuple are passed by value. This is reasonable for small fields, such as numbers or even names. But, for large fields, such as structures or long character strings, this can result in extra copying. Passing some types by reference avoids this extra copying. More importantly, if a tuple space is to be passed as a field in a tuple, it must be passed by reference, because two processes must have references to the same tuple space to communicate. Consequently, the Kernel Linda data types are divided into two groups, simple (pass-by-value) and composite (pass-by-reference). The simple types are

int	a 32-bit integer
real	a 32-bit floating-point number
real64	a 64-bit floating-point number
name	an atomic identifier
struct	a structure containing user-supplied data
null	the error value

The simple types are always passed by value. If a process passes one of these objects to another process (for example, via tuple space), each process will have a separate copy of the object, so modifying the object in one process will not affect the value of the object to the other process. The composite types are

dict	a Kernel Linda tuple space (dictionary)
string	a string of characters
block	an array of bytes
array	a heterogeneous array of other Kernel Linda objects

Composite types are always passed by reference; if a process passes one of these objects to another process (again, typically via tuple space), then both processes will have a reference to the same object, even if the processes are on different processors. Thus, modifying the object in one process will affect the value of the object to the other process . This is the only Kernel Linda communication mechanism: modifying a shared object. In particular, the Linda operations (*in, out,* and *rd*) modify shared dictionaries. Shared objects are implemented on a distributed-memory computer using a mechanism called locators, described in the next section.

Of course, it is possible to make a copy of a Kernel Linda object. Making a local copy of an object that was passed by reference gives the equivalent semantics as pass-by-value.

A dict (dictionary) is the Kernel Linda name for a tuple space. A tuple in a dictionary consists of a single key/value pair. The keys are almost always names or integers but can be of any data type. A value associated with a key in a dictionary can also be of any type, for example, another dictionary. Tuples are only allowed to have a single value, but this value can be a structured object, such as an array, which holds multiple values. As in Linda, multiple copies of the same tuple can be held in tuple space; this corresponds to a key in a Kernel Linda dictionary having multiple values.

To allow the user to choose between value and reference semantics, strings and names are essentially the same type, except that strings are passed by reference and names are passed by value. The same is true of blocks and structs. This allows copying to be avoided, if desired. For example, names are typically used as keys in tuples, while strings are used for long strings of text such as shell scripts. Likewise, an executable load module is normally stored as a block to avoid making extra copies, but simple data structures, corresponding to a struct in C, are normally stored as a Kernel Linda struct.

Implementation. Kernel Linda evolved through several implementations, both on single-processor and distributed-memory computers. On a single-processor computer, and within a single processor on a distributed system, Kernel Linda is implemented as a shared-memory-model program. Internally, composite objects are protected by a semaphore to guard against concurrent updates. This semaphore is controlled by the system and is not visible to the user.

The implementation on a distributed-memory computer, where a distributed-memory-model program is used, is much more interesting. In this case, composite objects are manipulated using a form of distributed pointer called a *locator,* which, in addition to other information, contains the processor ID of the processor on which the object is located. When an operation is to be performed on a remote object, a message is sent to the processor containing that object, and a server process on the remote processor performs the operation. When an operation is to be performed on a local object (including operations by the server process after receiving a request from a remote processor), the shared-memory-model program is used. The implementation of Kernel Linda on a shared-memory computer is more straightforward than the one for the distributed-memory computer because it avoids the remote server process.

Despite the fact that Kernel Linda does not use a preprocessor, the overhead compared to that of message passing is reasonable. This overhead arises mainly from evaluation of the hashing function, any required searching within the hash table, and the locking of a semaphore. For transfers of reasonable amounts of data the overhead is about 10 percent, rising to as much as 50 percent for shorter transfers (a synchronization with no data transfer being the worst case). This is comparable to the overhead of higher level languages such as C, and, of course, when this overhead is unacceptable, the user can drop down into machine primitives, analogous to the way C programmers can drop down into assembly language.

REFERENCES.

IMP13 : S. Ahuja, N. Carriero, D. Gelernter : "Linda and friends", *IEEE Computer* **19#8**, 26 (August 1986).

IMP14 : N. Carriero, D. Gelernter : "Linda in context", *Communications of the ACM* **32**, 444 (1989).

IMP15 : W. Leler : “Linda meets Unix”, *IEEE Computer* **23#2**, 43 (February 1990).

QUESTIONS.

What problems does Linda solve?

What is it that Kernel Linda does that Linda doesn't ? What sort of support do they need from the operating system ?
