14

Concurrent and Distributed Programming

A concurrent program defines two or more sequences of actions that may be executed simultaneously. Concurrent programs may be executed in two general ways:

- *Multiprogramming.* A single physical processor may run several processes simultaneously by interleaving the steps of one process with steps of another. Each individual process will proceed sequentially, but actions of one process may occur between two adjacent steps of another.
- *Multiprocessing.* Two or more processors may share memory or be connected by a network, allowing processes on one processor to interact with processes running simultaneously on another.

Concurrency is important for a number of reasons. Concurrency allows different tasks to proceed at different speeds. For example, multiprogramming allows one program to do useful work while another is waiting for input. This makes more efficient use of a single processor. Concurrency also provides programming concepts that are important in user interfaces, such as window systems that display independent windows simultaneously and for networked systems that need to send and receive data to other computers at different times. Multiprocessing makes more raw processing power available to solve a computational problem and introduces additional issues such as unreliability in network communication and the possibility of one processor proceeding while another crashes. Interaction between sequential program segments, whether they are on the same processor or different processors, raises significant programming challenges.

Concurrent programming languages provide control and communication abstractions for writing concurrent programs. In this chapter, we look at some general issues in concurrent programming and three language examples: the actor model, Concurrent ML, and Java concurrency. Some constructs we consider are more appropriate for multiprogramming, some for multiprocessing, and some are useful for both.

Historically, many concurrent systems have been written in languages that do not support concurrency. For example, concurrent processes used to implement a network router can be written in a language like C, with system calls used in place of concurrency constructs. Concurrently executed sequential programs can

TONY HOARE

A buoyant, inquisitive person with a twinkle in his eye, Charles Anthony Richard Hoare began his career in computing in 1960 as a programmer for Elliott Brothers, a small scientific computer manufacturer. He became Professor of Computer Science at the Queen's University in Belfast in 1968 and moved to Oxford University in 1977. Professor Hoare retired from Oxford at the mandatory age in 1999 but continues to be active in computing at Microsoft Research Labs in Cambridge, U.K.

Midway through his career, Hoare received the ACM Turing Award in 1980, "For his fundamental contributions to the definition and design of programming languages." At the time he was recognized for his work on methods for reasoning about programs, clever algorithms such as Quicksort, data structuring techniques, and the study of monitors. In 1980, he was also beginning his study of concurrency through the development of Communicating Sequential Processes (CSP). Along with Milner's Calculus of Communicating Systems (CCS), CSP has been a highly influential system for specifying and analyzing certain types of concurrent systems. The standard method for proving properties of imperative programs by using loop invariants, defined in his 1969 journal article, is now commonly called Hoare Logic.

The picture on this page shows Tony Hoare and his wife Jill standing outside Buckingham Palace, holding the medal he received when he was knighted by Queen Elizabeth II.

communicate by reading and writing to a shared structure or by using abstractions provided by the operating system. Programs can send or receive data from a Unix pipe, for example, by function calls that are handled by the operating system. C programs can also create processes and terminate them by using system calls.

There are several reasons why a concurrent programming language can be a better problem-solving tool for concurrent programming than a sequential language. One is that a concurrent programming language can provide abstractions and control structures that are designed specifically for concurrent programming. We look at several examples in this chapter. Another is that a programming language may provide "lighter-weight" processes. Operating systems generally give each process its own address space, with a separate run-time stack, program counter, and heap. Although this protects one process from another, there are significant costs associated with setting up a process and in switching the flow of control from one process to another. Programming language implementations may provide lightweight processes that run in the same operating system address space but are protected by programming language properties. With lightweight processes, there is less cost associated with concurrency, allowing a programmer to use concurrency more freely. Finally, there is the issue of portability across operating systems. Although it is possible to do concurrent programming with C and Unix system calls, for example, programs written for one version of Unix may not run correctly under other versions of Unix and may not be easy to port to other operating systems.

14.1 BASIC CONCEPTS IN CONCURRENCY

Before looking at some specific language designs, we discuss some of the basic issues in concurrent programming and some of the traditional operating system mechanisms used to allow concurrent processes to cooperate effectively.

A sequence of actions may be called a *process*, *thread*, or *task*. Although some authors use these terms interchangeably, they can have different connotations. The word process is often used to refer to an operating system process, which generally runs in its own address space. Threads, in specific languages like Concurrent ML and Java, may run under control of the language run-time system, sharing the same operating system address space. Some authors define thread to mean "lightweight process," which means a process that does not run in a separate operating system address space. The term process is used in the rest of Section 14.1, as many of the mechanisms we discuss are used in operating systems. In discussing Concurrent ML and Java, we use the term thread because this is the standard term for these languages and because threads defined in a Concurrent ML or Java program are lightweight processes that all run in the same operating system address space.

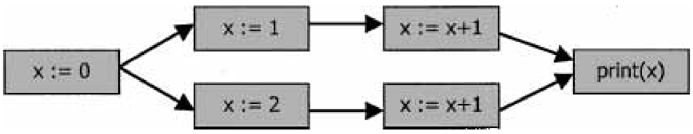
14.1.1 Execution Order and Nondeterminism

Concurrent programs may have many possible execution orders. An elementary and historical concurrent programming construct is the cobegin/coend form used in Concurrent Pascal. This language was developed by P. Brinch Hansen at Caltech in the 1970s. Here is an example Concurrent Pascal program that has several execution orders:

```
x := 0;
cobegin
    begin x := 1; x := x+1 end;
    begin x := 2; x := x+1 end;
coend;
print(x);
```

This program executes sequentially, as usual in Pascal, except that, within the cobegin/coend statement, all of the independent blocks are executed concurrently. Concurrent blocks may interact by reading or assigning to global variables. The cobegin/coend statement terminates when all of the concurrent blocks inside it terminate. After the cobegin/coend statement terminates, the program continues sequentially with the statement following coend.

Here is a diagram showing how our sample Concurrent Pascal program can be executed on a single processor. Each assignment statement is executed *atomically*, meaning that even though an assignment statement might be carried out with a sequence of more elementary machine-language steps, once the processor starts executing an assignment statement, the processor continues and completes the assignment



before allowing another process to execute. In other words, only one assignment statement runs at a time. However, the order between assignments may be different on different executions of the program. Each arrow in this figure illustrates a constraint on execution order: The statement at the left end of the arrow must finish before the statement on the right can start. However, there is no necessary ordering between statements that are not connected by a sequence of arrows. In particular, the assignment x:=1 could be executed before or after the assignment x:=2. Here are some possible execution orders for this program:

```
x:=0; x:=1; x:=2; x:=x+1; x:=x+1; print(x) // Output 4
x:=0; x:=2; x:=1; x:=x+1; x:=x+1; print(x) // Output 3
x:=0; x:=1; x:=x+1; x:=2; x:=x+1; print(x) // Output 3
x:=0; x:=2; x:=x+1; x:=1; x:=x+1; print(x) // Output 2
```

This program illustrates one of the main difficulties in designing and testing concurrent systems: nondeterminism. A program is *deterministic* if, for each sequence of program inputs, there one sequence of program actions and resulting outputs. In contrast, a program is *nondeterministic* if there is more than one possible sequence of actions corresponding to a single input sequence. Our sample Concurrent Pascal program is nondeterministic because the order of execution of the statements inside the cobegin/coend construct is not determined. There are several possible execution orders and, if the program is run several times on the same computer, it is possible for the program to produce different outputs on different runs.

Nondeterminism makes it difficult to design and debug programs. In a complex software system, there may be many possible execution orders. It is difficult for program designers to think carefully about millions of possible execution orders and identify errors that may occur in only a small number of them. It also may be impractical to test program behavior under all possible execution orders. Even if there is some way to cause the system to try all possible orders (a difficult task for distributed systems involving many computers connected by a network), there could be so many orders that it would take years to try them all.

One example that illustrates the complexity of nondeterminism is the design of cache coherency protocols for multiprocessors. In a modern shared-memory multiprocessor, several processors share common memory. Because access to shared memory takes time and can be a bottleneck when one processor needs to wait for another process, each processor maintains a local cache. Each cache is a small amount of memory that duplicates values stored in main memory. The job of the cache coherence protocol is to maintain the processor caches and to guarantee some form of memory consistency, e.g., the value returned by every `load`/`store` sequence must be consistent with what could have happened if there were no caches and all `load`/`store` instructions operated directly on shared memory in some order. Cache coherence protocols are notoriously difficult to design and many careful designs contain flaws that surface only under certain conditions. Because of their subtlety, many cache coherence protocols are analyzed by automated tools that are designed to detect errors in nondeterministic systems. In Subsection 14.4.3 we look at the Java virtual machine memory model, which involves some of the same issues as cache coherency protocols for multiprocessors.

14.1.2 Communication, Coordination, and Atomicity

Every programming language for explicit concurrency provides some mechanism to initiate and terminate independent sequential processes. The programming languages we consider in this chapter also contain mechanisms for some or all of the following general purposes:

- *communication* between processes, achieved by mechanisms such as buffered or synchronous communication channels, broadcast, or shared variables or objects,
- *coordination* between processes, which may explicitly or implicitly cause one process to wait for another process before continuing,
- *atomicity*, which affects both interaction between processes and the handling of error conditions.

When we compare concurrent programming languages, we always look at the way processes are defined, how they communicate and coordinate activities, and how programmers may define atomic actions within the language (if the language provides explicit support for atomicity).

The most elementary form of interprocess communication, as illustrated in the Concurrent Pascal example in Subsection 14.1.1, is through shared variables. Processes may also communicate through shared data structures or files. Another form of interprocess communication is called *message passing*. Here are some of the main distinctions among various forms of message-passing mechanisms:

- *Buffering.* If communication is *buffered*, then every data item that is sent remains available until it is received. In *unbuffered* communication, a data item sent before the receiver is ready to accept that it may be lost.
- *Synchronicity.* In *synchronous* communication, the sender cannot transmit a data item unless the receiver is ready to receive it. With *asynchronous* communication,

the sending process may transmit a data item and continue executing even if the receiver is not ready to receive the data.

- *Message Order*. A communication mechanism may preserve transmission order or it may not. If a mechanism preserves transmission order, then a sequence of messages will be received in the order that they are sent.

Coordination mechanisms allow one process to wait for another or notify a waiting process that it may proceed. Concurrent Pascal cobegin/coend provides a rudimentary form of process coordination as all processes started at the same cobegin must finish before the statement following coend may proceed. Locking and semaphores, discussed in Subsections 14.1.3 and 14.1.4, provide more sophisticated forms of concurrency control.

An action is *atomic* if every execution will either complete successfully or terminate in a state that is indistinguishable from the state in which the action was initiated. A nonatomic action may involve intermediate states that are observable by other processes. A nonatomic action may also halt in error before the entire action is complete, leaving some trace of its partial progress. Generally speaking, any concurrent programming language must provide some atomic actions, because, without some guarantee of atomicity, it is extremely difficult to predict the behavior of any program.

14.1.3 Mutual Exclusion and Locking

There are many programming situations in which two or more processes share some data structure, and some coordination is needed to maintain consistency of the data structure. Here is a simple example that illustrates some of the issues.

Example 14.1

Suppose we have a system that allows people to sign up for mailing lists. This system may involve a number of lists of people, and some of these lists may be accessed by more than one process. Here is a procedure to add a person to a list:

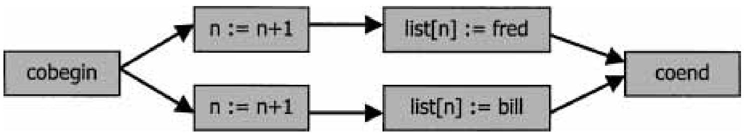
```
procedure sign_up(person)
    begin
        n := n + 1;
        list[n] := person;
    end;
```

This procedure uses an array to store the names in the list and an integer variable to indicate the position of the last person added to the list. There may be other procedures to read names from the list or remove them.

The sign_up procedure works correctly if only one process adds to the list at a time, but not if two processes add to the list concurrently. For example, consider the following program section:

```
cobegin
    sign_up(fred);
    sign_up(bill);
end;
```

As discussed in Subsection 14.1.1, there are several possible execution orders for the assignments to number and list. The ordering between assignments is shown in this illustration:



One possible execution order increments n twice, then writes names fred and bill into the same location. This is not correct: Instead of adding both names to the list, we end up with an empty location, and only one of the two names in the list. The state resulting from the sequence of assignments

```
n := n+1; n := n+1; list[n] := fred; list[n] := bill;
```

is called an *inconsistent state* because it is not consistent with the intended behavior of the operations on lists. An invariant for list and integer variable n is that for every integer i up to n there are data stored in list[i]. This invariant is preserved by sign_up, if it is called sequentially. However, if it is called concurrently, the data structure invariant may be destroyed.

Mutual Exclusion

Example 14.1 shows that, for certain operations, it is important to restrict concurrency so that only one process proceeds at a time. This leads to the notion of *critical section*. A critical section is a section of a program that accesses shared resources. Because a process in a critical section may disrupt other processes, it may be important to allow only one process into a critical section at a time. Mechanisms for preventing conflicting actions in independent processes are called *mutual-exclusion* mechanisms.

Mutual-exclusion mechanisms may be designed to satisfy some or all of the following criteria:

- *Mutual exclusion*: Only one process at a time may be in its critical section.
- *Progress:* If no processes are in their critical section and some process wants to enter a critical section, it becomes possible for one waiting process to enter its critical section.
- *Bounded waiting:* If one process is waiting, there must be a bound on the number of times that other processes are allowed to enter the critical section before this process is allowed to enter. In other words, no waiting process should have to wait indefinitely.

It is also desirable to ensure that, if one process halts in a critical section, other processes will eventually be able to enter the critical section.

Locks and Busy Waiting

A general approach to mutual exclusion is shown in the following pseudocode, which uses the sign_up procedure from Example 14.1. The main idea is that each process executes some kind of *wait* action before calling sign_up and executes some kind of *signal* action afterward. The wait action should check to see if any other process is calling sign_up or executing any other procedure that might interfere with sign_up. If so, the action should make the process wait, only allowing the call to sign_up to occur when this is acceptable. The signal action allows a waiting process to proceed.

```
<initialze concurrency control>
cobegin
    begin
       <wait>          // wait for access
       sign_up(fred);  // critical section
       <signal>        // allow waiting process to proceed
    end;
    begin
       <wait>          // wait for access
       sign_up(bill);  // critical section
       <signal>        // allow waiting process to proceed
    end;
end;
```

Although the ideas behind *wait* and *signal* actions may seem straightforward, implementing these actions is tricky. In fact, in the early days of concurrent programming, several incorrect implementations were proposed before a good solution was finally discovered. We can understand some of the subtlety by examining a plausible but imperfect implementation.

Example 14.2

Let us try to implement wait and signal by using an ordinary integer variable as a "lock." Each process will test the variable before it enters its critical section. If another process is already in its critical section, then the lock will indicate that the process must wait. If the value of the lock indicates that the process may enter, then the process sets the lock to keep other processes out. This approach is shown in the following code:

```
lock := 0;
cobegin
    begin
        while lock=1 do end;   // loop doing nothing until lock is 0
```

```
            lock:=1;                 // set lock to enter critical section
            sign_up(fred);           // critical section
            lock:=0;                 // release lock
        end;
        begin
            while lock=1 do end;     // loop doing nothing until lock is 0
            lock:=1;                 // set lock to enter critical section
            sign_up(bill);           // critical section
            lock := 0;               // release lock
        end;
    end;
```

Each process (each of the two blocks) tests the lock before entering the critical section (i.e., before calling sign_up). If the lock is 1, indicating that another process is using sign_up, then the process loops until the lock is 0. When the lock is 0, the process enters the critical section, setting lock:=1 on entry and lock:=0 on exit. Using a loop to wait for some condition is called *busy waiting*.

The problem with using a shared variable for mutual exclusion is that the operation that reads the value of the variable is different from the operation that sets the variable. With the method shown in this code, it is possible for one process to test the variable and see that the lock is open, but before the process can lock other processes out, another process can also see that the lock is open and enter its critical section. This allows two processes to call sign_up at the same time.

A fundamental idea is concurrency control is atomic test-and-set. An atomic test-and-set, sometimes called TSL for Test and Set Lock, copies the value of the lock variable and sets it to a nonzero (locked) value, all in one atomic step. Although the value of the lock variable is being tested, no other process can enter its critical section. Most modern processors provide some form of test-and-set in their instruction set.

Deadlock

Deadlock occurs if a process can never continue because the state of another process. Deadlock can easily occur if there are two processes and two locks. Suppose that Process 1 first sets Lock 1 and then wait for Lock 2. If Process 2 first sets Lock 2 and then waits for Lock 1, then both processes are waiting for the lock held by the other process. In this situation, neither process can proceed and deadlock has occurred.

One technique that prevents deadlock is called *two-phase locking*. In two-phase locking, a process is viewed as a sequence of independent tasks. For each task, the process must first acquire all locks that are needed (or could be needed) to perform the task. Before proceeding from one task to the next, a process must release all locks. Thus there are two phases in the use of locks, a locking phase in which all locks are acquired and a release phase in which all locks are released. To avoid deadlock in the locking phase, all processes agree on an ordering of the locks and acquire locks in that order. You may find more information about locking policies and deadlock in books on operating systems and database transactions.

14.1.4 Semaphores

Semaphores, first proposed by Edsger W. Dijkstra in 1968, provide a way for an operating system to guarantee mutual exclusion without busy waiting. There are several different detailed implementations of semaphores. All use a counter or Boolean flag and require the operating system to execute the entire wait or signal procedure atomically. Atomicity prevents individual statements of one wait procedure from being interleaved with individual statements of another wait on the same semaphore.

A standard *semaphore* is represented by an integer variable, an integer maximum, and a queue of waiting processes. Initially, the integer variable is set to the maximum. The maximum indicates the number of processes that may enter a critical section at the same time; in many situations the maximum is one. If a program contains several shared resources, the program may use a separate semaphore for each resource.

When a process waits on a semaphore, the wait operation checks the integer value of the semaphore. If the value is greater than 0, this indicates that a process may proceed. The value is decremented before proceeding to limit the number of processes that are allowed to proceed. (If the maximum is 1, then the semaphore is decremented to 0 and no other process can proceed.)

If a wait is executed on a semaphore whose integer value is 0, then the process is suspended and placed on a queue of waiting processes. Suspending a process is an operating system operation that keeps the process from continuing until the process is resumed. Here is a pseudocode showing how semaphore wait may be implemented:

```
procedure wait (s : Semaphore)
begin                            // This procedure must be executed atomically
    if s.value > 0 then
         s.value:= s.value - 1;   // Enter section and decrement counter
    else
         suspend_on (s.queue); // Wait for other processes to finish
end;
```

When a process leaves a critical section, it must allow other processes waiting on the associated semaphore to proceed. The signal operation checks the semaphore queue to see if any process is suspended. If the queue is not empty, one of the suspended waiting processes is allowed to run. If no process is waiting, then the integer value of the semaphore is incremented. Here is a pseudocode showing how semaphore signal might be implemented:

```
procedure signal (s : Semaphore)
begin                              // This procedure must be executed atomically
    if length (s.queue) = 0 then
        s.value := s.value + 1;   // Increase count allowing other processes to enter
    else
        allow_one_process (s.queue);   // Wake up one suspended process
end
```

If allow_one_process wakes up the first process on the queue, then each waiting process will have to wait only for the processes ahead of it in the queue. This achieves bounded waiting, as defined in Subsection 14.1.3.

Originally, Dijkstra, used P() for Wait() because P is the first letter of the Dutch word *passeren*, which means to pass, and V() for Signal() because V is the first letter of the Dutch word *vrijgeven*, which means to release.

14.1.5 Monitors

Semaphores provide mutual exclusion, but only if used properly. If a process calls wait before entering a critical section, but fails to call signal afterwards, this error may cause the program to deadlock. This kind of program error can be hard to find. Monitors, developed in the early 1970s, place the responsibility for synchronization on the operations that access data. Monitors are similar to abstract data types, with all synchronization placed in the operations of the data type. This makes it easier to write correct client code.

A *monitor* consists of one or more procedures, an initialization sequence, and local data. The local data are accessible only by the monitor procedures, which are responsible for all synchronization associated with concurrent access to the data. In traditional terminology, a process *enters* the monitor by invoking one of its procedures. The synchronization code is generally designed so that only one process may be in the monitor at a time; any other process that calls a monitor procedure is suspended and waits for the monitor to become available.

In modern terminology, a monitor might be called a *synchronized object*. In an important early paper (Monitors: An Operating System Structuring Concept, Comm. ACM 17(10) 1974, pp. 549–557), C.A.R. Hoare showed that monitors can be implemented by use of semaphores and, conversely, semaphores can be implemented by use of monitors. In this sense, the two constructs are equally powerful synchronization constructs. However, monitors have proven to be a more useful program-structuring concept. We look at the main programming issues associated with monitors in Section 14.4 in our discussion of Java synchronized objects.

14.2 THE ACTOR MODEL

Actors are a general approach to concurrent computation. Each actor is a form of reactive object, executing some computation in response to a message and sending out a reply when the computation is done. Actors do not have any shared state, but use buffered asynchronous message passing for all communication. Although pure actor systems have not been widely successful, actors are an appealing metaphor and an interesting point on the spectrum of concurrent language design alternatives.

Actors originated through Carl Hewitt's work on the artificial intelligence system Planner in the early 1970s. The concept evolved as a result of approximately 20 years of subsequent effort by Hewitt and many others, influencing a range of concurrent object-oriented systems. There is no one actor system that embodies all of the concepts that have been developed in the literature. As a result, our discussion of actor systems is more a discussion of a point of view, with possible alternatives and embellishments, than an evaluation of a specific, concrete system.

An *actor* is an object that carries out its actions in response to communications it receives. There are three basic actions that an actor may perform:

- It may send communication to itself or other actors,
- It may create actors,
- It may specify a replacement behavior, which is essentially another actor that takes the place of the actor that creates it for the purpose of responding to later communication.

Actor computation is reactive, which means that computation is performed only in response to communication. An actor program does not explicitly create new processes and control their execution. Instead, an actor program creates some number of actors and sends them messages. All of these actors can react to messages concurrently, but there is no explicit concept of thread.

An actor is dormant until it receives communication. When an actor receives a message, the script of the actor may specify subsequent communication and a set of new actors to create. After executing its script, the actor returns to its dormant state. The replacement behavior specifies how the actor will behave when it receives another message. For example, if an actor representing the number 3 receives a communication *increment by* 1, its replacement behavior will be to represent the number 4. There is no assignment to local variables in the actor model; the only side effect of an actor is to specify a replacement behavior, which is an atomic state change that takes effect only after the other activities are complete.

In any computation, each actor receives a linearly ordered sequence of messages. However, messages are not guaranteed to arrive in the order in which they are sent. If one actor, A, sends two communications to another, B, then B may not receive them in the same order that A sent them. This is consistent with the idea that actors may be physically located in different places, connected by a communication network that might route different messages along different paths.

An important part of the actor model is the *mail system*, which routes and buffers messages between actors. Every message must be sent to a *mail address*; one actor may communicate with another if it knows its mail address. When an actor A specifies a replacement behavior, the replacement behavior will be the script for a new actor that receives all messages addressed to the mail address of A.

A message from one actor to another is called a *task*. A task has three parts:

- A unique tag, distinguishing it from other tasks in the system,
- A target, which is the mail address of the intended receiver,
- A communication, which is the data contained in the message.

Figure 14.1 shows a finite set represented by an actor. The actor in the upper left of the figure represents the set with elements 1, 4, and 7. This actor receives a task `Insert 2`. In response, the actor becomes the actor on the right that has elements 1, 2, 4, and 7. This actor receives the task `Get_min`. As a result, the actor sends the message 1, which is the minimum element in the set, and becomes an actor that represents the set with elements 2, 4, and 7.

The configuration of an actor system consists of a finite set of actors and a finite set of pending or undelivered tasks. Because each task has a single, specified destination,

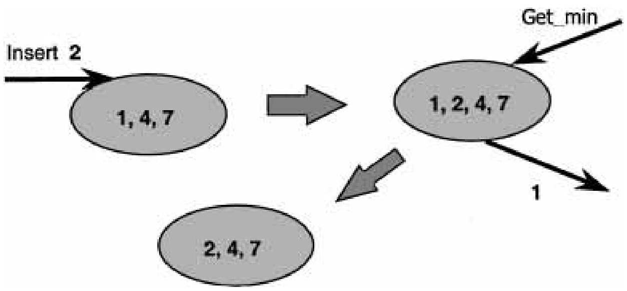


Figure 14.1. Messages and state change for a finite-set actor.

we may visualize the system as a set of actors and task queues, with one queue per actor and each task on exactly one queue.

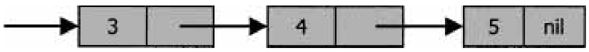
Although we will not look at the syntactic structure of actor programs in any detail, here is a short example to give some feel for the language. The code subsequent is something called a behavior definition, which defines a behavior and gives it a name but does not create an actor. A program that uses this behavior definition creates one or more stack_node actors and sends them tasks. As this example illustrates, a behavior may be defined as a function of parameters, which are called *acquaintances:*

```
a stack_node with acquaintances content and link
    if operation requested is push(new_content) then
        let P=new stack_node with current acquaintances
        become stack_node with acquaintances new_content and P
    if operation requested is a pop and content != nil then
        become forwarder to link
        send content to customer
```

A stack_node actor with the behavior defined by this code has two data fields (determined by acquaintances), content and link. This actor sits and waits for a task. The two tasks that the actor will process are push and pop. If the operation requested is a push, then the push task will include a content value called new_content. The actor creates a copy P of itself, as indicated by the expression new stack_node with current acquaintances and becomes a stack node with data new_content and link to the copy P of the actor's previous state. The reason why the actor "becomes" the new actor, instead of keeping its original state and creating a new actor linked to it, is that future tasks sent to this actor should reach the new stack with new_content added, not the old stack without these data.

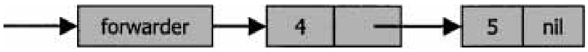
The pop operation uses an actor-specific concept called a forwarder. A forwarder actor does nothing itself, except forward all tasks to another actor. The reason for having forwarders is illustrated by the pop operation, in which the actor that receives the pop task would like to disappear and let another actor take its place. Instead of "disappearing," the actor becomes a forwarded to the actor it is linked to.

We can see how forwarders work by looking at some pictures. Suppose that three actors represent a stack, with data 3, 4, and 5, as shown here:



If the actor on the left, representing the top of the stack, receives a pop task, then the actor sends content 3 to the actor that sent the pop task and becomes a forwarder. This produces a stack of three actors, as shown here.

The forwarder has the same mail address as that of the actor with content 3, so any task sent to the top of the stack will now be forwarded to the actor with content 4. In effect, the forwarder node is invisible to any user of the stack. In fact, an actor implementation may garbage-collect forwarders and handle forwarding in the mail system.



Two interesting general aspects of actors are the concepts of replacement behavior and the use of buffered asynchronous communication. An actor changes state only by becoming another actor after it completes processing input. This is a very clean idea that eliminates state change within an actor while computing a response to a task.

The motivation for using asynchronous communication is that it is easily implemented on a wide-area network. When actors are on different computers connected by a slow network, it is easier to implement asynchronous communication by means of standard IP-based protocols than it is to devise some form a synchronous communication. With asynchronous communication, it is important either to use some form of buffering or to use some kind of resend protocol to handle lost messages. Although buffers may overflow in practice, buffered communication is a convenient abstraction for many programming situations.

The actor design does not guarantee the order of message delivery. This is inconvenient in many situations. For example, consider an actor traversing a graph, using another stack actor to keep track of graph nodes to revisit. If there is no guarantee on the order of messages that push graph nodes onto the stack, the algorithm will not necessarily traverse the graph in depth-first order. The design rationale for basing actors on communication without guaranteed order is that order can be imposed by adding sequence numbers to messages.

There are also some pragmatic issues associated with programming when message delivery may take arbitrarily long. Returning to the example of searching a graph, using a stack to keep track of nodes that must be revisited, it seems difficult to tell whether the graph search is finished. The usual algorithm continues to visit nodes of the graph until the stack of remaining nodes is empty. However, if an actor representing a stack sends a message saying that the stack is empty, this may mean only that all push messages are still in the mail system somewhere and have not yet been delivered to the stack.

Although the simplicity of the actor model is appealing, these problems with message order, message delivery, and coordination between sequences of concurrent actions also help us appreciate the programming value of more complex concurrent languages.

14.3 CONCURRENT ML

Concurrent ML is a concurrent extension of Standard ML that provides dynamic thread creation and synchronous message passing on typed channels. The language was designed by John Reppy, who was interested in concurrency in user interfaces and distributed systems. A motivating application was Exene, a multithreaded *X Window System* toolkit written by Emden Gansner and Reppy. The language implementation, as an extension of Standard ML of New Jersey, is available on the web from several sources and a book describing the language has been published (J. H. Reppy, *Concurrent Programming in ML*, Cambridge Univ. Press, New York, 1999). In the uniprocessor implementation, Concurrent ML threads are implemented with continuations.

The most interesting aspects of Concurrent ML, also referred to as CML, are the ways that communication and coordination are combined and the way that CML allows programmers to define their own communication and synchronization abstractions. The main programming language concept that gives CML this power is a first-class value representing a synchronization called an *event*. We look at some aspects of CML in some detail to study the way that synchronous communication can work as a basic language primitive. Whereas Actors provide too little synchronization, CML, with only threads and channels, provides too much synchronization. This problem is solved with an interesting event mechanism.

14.3.1 Threads and Channels

A CML process is called a *thread*. When a CML program begins, it consists of a single thread. This thread may create additional threads by using the spawn primitive:

```
spawn : (unit → unit) → thread_id
```

where unit is the SML type with only one element (similar to C void, as discussed in Subsection 5.4.2), and thread_id is the CML type used to identify and refer to threads.

When spawn f is evaluated, the function call f() is executed as a separate thread. The return value of the function is discarded, but the function may communicate with other threads by using channels. The thread that evaluates spawn f is called the *parent thread* and the thread running f() is called the *child thread*. In CML, the parent–child relation does not have any impact on thread execution – either thread may terminate without affecting the other.

Here is an expression that spawns two child threads, each printing a message by using the CIO.print function for concurrent input/output:

```
let  val pr = CIO.print
in   pr "begin parent\n";
     spawn (fn () => (pr "child 1\n";));
     spawn (fn () => (pr "child 2\n";));
     pr "end parent\n"
end;
```

When evaluated, this expression will always print "begin parent" first and then print the other three strings, "child 1", "child 2", and "end parent" in arbitrary order. Unlike cobegin/coend, each child thread may finish before or after the parent thread.

Another CML primitive, which we can define by combining spawn with looping or tail recursion, is forever. This function takes an initial value and a function that can be repeatedly applied to this value:

```
forever : 'a → ('a → 'a) → unit
```

The expression forever init f computes x1 = f(init), followed by x2 = f(x1), x3 = f(x2), and so on, indefinitely. The values x1, x2, x3, . . . , are discarded, but the function f may communicate with other threads. Other features of CML can be used to terminate a thread that loops indefinitely.

An important communication mechanism in CML is called a *channel*. For each CML type 'a, there is a type 'a chan of channels that communicate values of type 'a. Two useful operations on channels are send and receive:

```
recv : 'a chan  →  'a
send : ('a chan * 'a)  →  unit
```

As you would expect, if c is an int chan, for example, then recv c returns an int. The type of send is a little more complicated, but makes sense if you think about it: If c is an int channel, then send(c,3) sends the integer 3 on channel c. There is no obvious value for send(c,3) to return, so the type of send(c,3) is unit.

CML message passing is synchronous, meaning that communication occurs only when both a sender and a receiver are ready to communicate. If one thread executes a send and no thread is ready to execute recv on the same channel, the sending thread *blocks* (stops and waits) for a thread to execute recv. Similarly, if a thread executes recv and no thread is sending, the receiving thread blocks and waits for a thread to send. Although standard CML channels are point-to-point, meaning that each send reaches one receiver, CML also has multicast channels that allow one send to reach many receivers.

Channels are created by the channel constructor,

```
channel : unit → 'a chan
```

which creates a channel of arbitrary type. Type inference will generally replace the type variable in the type of channel() with the type of values sent or received from the channel. It is not possible to use the same channel to send and receive different types of values, except by using ML datatype to create a type that is a union of several types.

Here is a simple example that uses a channel to synchronize two threads:

```
- fun child_talk () =
    let  val ch = channel ()
         val pr = CIO.print
    in
         spawn (fn () => (pr "begin 1\n";   send(ch,0); pr "end 1\n"));
         spawn (fn () => (pr "begin 2\n";   recv ch; pr "end 2\n"));
    end;
> val child_talk = fn: unit  →  unit
```

This can be run (by RunCML.doit), with the following output:

```
begin 1
begin 2
end 2
end 1
```

The ordering between the two begins is arbitrary – they could appear in either order – and similarly for the ends. However, synchronous communication between the two threads causes the two begins to be printed before either of the ends.

Functions Using Threads and Channels

Because channels are "just another type" in CML, we can define functions that take channel arguments and return channel results. For example, it is easy to write a thread that takes integers off an input channel and sends their squares on an output channel. Because we may want to create threads that do squaring for different channels, it is useful to write a function that creates a squaring thread, given an input channel and an output channel:

```
fun square (inCh, outCh) =
     forever () (fn () =>
          send (outCh, square(recv(inCh))));
```

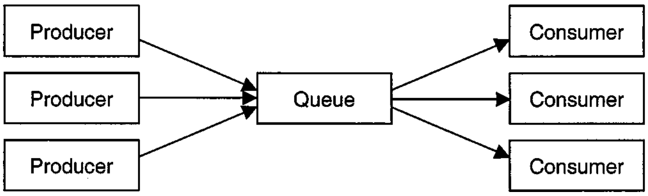
One useful programming technique in CML involves defining functions that create threads, then assembling a network of communicating threads by applying the functions to particular channels. Assuming that numbers(ch) creates a thread that outputs numbers on channel ch, here is a function that creates a thread that outputs squares on a channel and returns that channel:

```
fun mkSquares () =
let
    val outCh = channel()
    and c1 = channel()
in
    numbers(c1);
    square(c1, outCh);
    outCh
end;
```

If a CML thread has the name of a channel, then the thread can send messages on the channel, receive messages on the channel, or both. If a channel is passed to more than one thread, then each thread can send messages and receive messages on the channel.

14.3.2 Selective Communication and Guarded Commands

A basic problem with synchronous communication is that when only one thread is ready to communicate, it blocks and cannot do any other useful work. To see why this is a problem, think about a concurrent system in which several "producer" processes place items onto a queue and several "consumer" processes remove items from the queue.



A natural way to implement the queue is by some thread that runs forever, taking input from a producer channel and sending output to a consumer channel. When the queue is not empty, it should be ready to either send or receive, depending on whether a producer or consumer is ready. We can define the indefinitely looping thread by using forever and build the queue internal data structure by using the nonconcurrent part of CML, but choosing between producer and consumer cannot be done with synchronous channels and sequential control structures. The problem is that, if the queue thread executes a recv and a consumer is ready but no producer is ready, the system will wait for a producer to send, even though we would like the queue to send to the waiting consumer. A similar problem occurs if the queue process tries to send when it would be better to receive.

The solution to this problem is a concurrent control structure that chooses among available actions. A historically important notation for selective choice is Dijkstra's *guarded command language*. A guarded command, which has historically been written in the form

```
Condition ⇒ Command
```

consists of a predicate and an action. The intent of Condition ⇒ Command is that, if the Condition is true, then the Command may be executed. Dijkstra's guarded command language includes constructs such as

```
if
    Condition ⇒ Command
    Condition ⇒ Command
    . . .
    Condition ⇒ Command
fi
```

which executes one of the commands whose condition is true, and

```
do
    Condition ⇒ Command
    Condition ⇒ Command
    . . .
    Condition ⇒ Command
od
```

which loops indefinitely, on each iteration selecting one of the conditions that is true and executing the corresponding command. The if . . . fi construct is *not* the same as Lisp cond. Lisp cond is a sequential conditional. Each condition is tested in turn, starting with the first one, until a true condition is found. In Dijkstra's guarded command language, an if . . . fi statement may nondeterministically choose any guarded command whose condition is true.

The solution to the queue programming problem in CML is to build a form of do . . . od out of CML control structures, in effect writing the queue as

```
do
    Producer is ready to send ⇒ Receive input and store in queue
    Consumer is ready to receive and queue is not empty ⇒ Send to waiting consumer
od
```

We will see how to do this in the next subsection.

14.3.3 First-Class Synchronous Operations: Events

The most innovative and novel feature of CML is a synchonization value that is called an *event*. (The word event generally means "something that happens" in other languages, but in CML the word has a very specific meaning.) Intuitively, we can decompose send into two parts: the code executed when the send occurs and the act of executing this code. In a sense, the code that will cause a send to occur is the *send event*, and doing the send is called *synchronizing on this event.* We can similarly represent the ability to execute recv as an event and synchronize on this event to synchronously receive a value. By grouping send and receive events together in the type of events and defining operations on events CML provides some very interesting programming possibilities.

In case the idea behind CML events seems completely baffling, here is another way to think about events. Suppose that the send command is implemented as a function call. Then, in a language in which we can pass functions as arguments, return them as results, and store them in reference cells, it is possible to do various kinds of computation with the function that implements send, without actually sending. For example, you could build a function that sends twice in parallel out of one send function. You can think of the "send event" as the function that performs a send and "synchronizing on a send event" as calling the function that performs a send.

CML uses the type

```
'a event
```

as the type of all synchronous operations that return a value of type 'a when a thread synchronizes on them. The sync function takes an event and synchronizes on it, returning a value of the appropriate type:

```
sync : 'a event → 'a
```

One kind of synchronous event is a receive event,

```
recvEvt : 'a chan  →  'a event
```

When a thread synchronizes on a receive event by executing sync(recvEvt c) on a channel c of type 'a chan, the result is a value of type 'a. The recv function previously defined for synchronously receiving a value from a channel is definable as

```
fun recv(ch) = sync (recvEvt (ch));
```

In words, a thread can synchronously receive from a channel by synchronizing on a receive event for that channel.

The relation between a send event and send discussed in Subsection 14.3.1 is similar. A send event produces only a value of type unit, so the basic event operator sendEvt has type

```
sendEvt : ('a chan * 'a)  →  unit event
```

The functions sendEvt and recvEvt are called base-event constructors because they create event values that produce a single primitive synchronous operation.

The power of CML events comes from operators that build more complicated events from base events, including an operator select for selective communication, which we will discuss shortly. A simpler operator on events is the wrap function,

```
wrap : ('a event * ('a → 'b)) → 'b event
```

which combines an event that produces an 'a with a function 'a → 'b to get an event that produces a 'b. The idea is that if event e will produce a value x of type 'a and f: 'a → 'b, then wrap(e,f) will produce the value f(x).

Selective Communication with Events

The select operator for selective communication can be defined from a primitive function on events that chooses an event out of a list. The function

```
choose : 'a event list → 'a event
```

takes a list of events and returns one event from the list. (If the list is empty, then a special event that can never be used is returned.) Choose does not have to return the first event in the list or the last event from the list, or the event from any other specific position in the list. However, choose must return an event that can be synchronized on if there is one in the list. We will see how this works by discussing select.

The select function is defined as

```
fun select(evs) = sync(choose (evs))
```

In words, selecting from a list of events chooses an event from the list and synchronizes on it. This is the function we need to solve the programming problem described in Subsection 14.3.2.

Here is a simple example that uses select. The add function creates a thread that repeatedly reads a pair of numbers, one from each of two input channels, and sends the sum of each pair out on an output channel:

```
fun add ( inCh1, inCh2, outCh) =
     forever () (fn () = > let
         val (a,b) = select [
                        wrap (recEvt inCh1, fn a => (a, recv inCh2)),
                        wrap (recEvt inCh2, fn b => (recv inCh1, b))
                   ]
         in
              send (outCh, a+b)
         end
)
```

Let us read through the code and see how this works. Given three integer channels, the add function creates a looping thread by using forever. Each time through the body of the loop, add binds (a, b) to a pair of integers. The way it gets the pair of integers is determined by select. Select is applied to a list containing two events, the first involving recEvt(inCh1) and the second recEvt(inCh2). If some thread is ready to send an integer on inCh1, then the first event may be selected. Similarly, if some thread is ready to send an integer on inCh2, then the second event may be selected. Moreover, if there is a thread ready to send on either channel, then the select will choose a receive that can proceed; it will not choose the event that has to block and wait for another thread to send. If select chooses to receive an integer on inCh1, then the add thread may block and wait for an integer on the other channel, inCh2, as one integer from each channel is needed in order to proceed. Once a pair (a, b) is received, then add sends the sum a+b on the output channel outCh.

The programming technique used in add can also be used to implement a queue, as described in Subsection 14.3.2. The main idea is to use select to choose between send and receive events, depending on whether producer or consumer threads are ready to communicate. Here is the outline of a thread like the queue that selects between sending and receiving:

```
fun queue ( inCh, outCh) =
    forever () (fn () = >let
        val x = . . .
        in
             select [
                 wrap (recvEvt(inCh) , fn a => ( . . .)),
                 . . .
                 wrap (sendEvt(outCh, x) , fn () => ( . . .)),
                 . . .
            ]
        end
)
```

The event mechanism of CML can be used to define and implement other communication abstractions such as a remote procedure call (RPC) style communication mechanism or a multicast channel.

Example 14.3 Synchronized Shared Memory

CML allows threads to share reference cells; because CML is an extension of Standard ML, it would be awkward to prevent this. However, assignment to shared variables can have anomalous results if the threads doing the assignment do not coordinate their effects. A good way to coordinate assignment to shared memory is through the concept of synchronized shared memory. There are several possible forms of synchronized memory. We look at one example both to understand the concept and as an example of how communications abstractions can be defined in CML.

An M-structure is a memory cell that has two states, empty and full. When an M-structure is empty, a value can be placed in the structure. When it is full, a thread can read the value, which leaves the structure empty. (The name M-structure comes from the parallel language ID, designed by R. Nikhil at MIT.) An M-structure is initially empty and can be given a value by calling a put function. If the M-structure is not empty then a take function will get its value. There are two error conditions, one arising if a thread tries to put into a full structure and the other if a thread tries to take from an empty structure. The reason why M-structures cannot be written into when they are full is to avoid losing the value that is in the structure.

In our CML definition of M-structures, taken from Reppy's book (mentioned at the beginning of Section 14.3), we use the type 'a mvar for M-variables that hold a value of type 'a. Initially, an M-variable is empty. M-variables have operations mPut, which assign a value to an M-variable, and mTake, which read a value from an M-variable. Following the pattern established for channel send and recv, we find that it is more useful to define an event form of mTake, as this allows us to use select and other event operations and we can define mTake from mTakeEvt in the same way we defined recv from recvEvt. This gives us the following interface for M-variables:

```
type 'a mvar
val mVar : unit ⟶ 'a mvar
exception Put
val mTakeEvt : 'a mvar ⟶ 'a event
val mPut : ('a mvar * 'a) ⟶ unit
```

Here is the implementation of M-variables so you can see how channels may be used to build interesting data structures and events may be used to build interesting control mechanisms:

```
datatype 'a mvar = MV of {
    takeCh : 'a chan,
    putCh : 'a chan
    ackCh : bool chan
};
fun mVar() = let
    val takeCh = channel();
    val putCh = channel();
    val ackCh = channel();
    fun empty() = let val x = recv putCh in send (ackCh, true); full x end;
    and full x = select [
             wrap (sendEvt(takeCh, x) , empty),
             wrap (recvEvt(putCh) , fn _ => (send(ackCh, false); full x))
        ]
    in
        spawn empty;
        MV{ takeCh = takeCh, putCh = putCh, ackCh = ackCh }
    End;
fun mTakeEvt (MV{takeCh, ...}) = recvEvt takeCh;
exception Put;
fun mPut (MV{takeCh, putCh, ...}, x) = (
    send(putCh, x);
    if (recv ackCh) then () else raise Put);
```

The main idea is to spawn a thread for each M-variable and use channels for communicating with this thread to implement the operations mPut and mTake. Because the operations mPut and mTake send messages to the thread representing the M-variable, most of the work is actually done by the M-variable thread, not the functions mPut and mTake.

The most complicated part of the implementation is the function mVar for creating an empty M-variable. Because the representation of an M-variable is a record of three channels, the function mVar creates three new channels at the beginning of the function body and returns the record of them at the end of the function body. The three channels are used to send and receive messages to a thread that is spawned in the body of the function. The function empty, defining the operation of the thread, first waits for a value to be put into the M-variable by synchronously receiving on channel putCh used by the mPut function. Once the M-variable has been given a value, the thread sends an acknowledgement that the variable has a value on channel ackCh, then continues by executing the function call full x, where x is the value stored in the M-variable. The function full, which is executed when the M-variable is full, will either send the value of the M-variable or receive an attempt to put a new value in the variable. However, the mPut function sends a new value and then the thread sends false on ackCh, causing mPut to raise the exception Put.

One question you might want to think about is why full sends a message on ackCh that causes mPut to raise an exception, instead of raising the exception itself. If you understand this from reading the code, then you have really understood this program. The reason full does not raise an exception is that full is executing inside the thread that is used to implement the M-variables. By sending a message to mPut, which is executed in the thread that is attempting to store a value in the M-variable, the exception is raised in the calling thread. This allows the calling thread to handle the exception and proceed with some other task.

14.4 JAVA CONCURRENCY

Java concurrency is an integral part of the Java language and run-time system. We create a Java thread object by defining a class with a run() method, either by extending the Thread class or implementing the Runnable interface directly. When a thread object is activated, the run method runs as a separate thread inside the Java virtual machine (JVM). Java communication and synchronization are provided by separate mechanisms.

Java threads may communicate by calling methods of shared objects. For example, a producer thread may communicate with a consumer thread by using a queue object. When the producer thread wants to send its output to the consumer, the producer calls an enqueue method that places its output in the queue. When the consumer is ready for another item, the consumer thread may call a dequeue method, drawing another item off the queue.

Communication through a shared object does not synchronize the threads involved – each thread may attempt to execute a method independently. To avoid problems associated with shared resources (see Subsection 14.1.3), Java provides various synchronizations primitives. The Java language provides a semaphore primitive (maintaining a queue of waiting processes) and supports monitors directly in the form of synchronized objects, which are objects that allow only one thread to invoke a method at a time.

A second form of Java concurrency, which we discuss briefly in Subsection 14.4.4, arises when Java programs running in one VM communicate with programs running in another. The Java remote method invocation (RMI) mechanism allows an object running in one JVM to invoke methods of an object running in another JVM. Java RMI is used for distributed Java programming, with distributed programming infrastructures such as JINI and JXTA relying on RMI to varying degrees.

The Java thread design is a successful language feature. Java makes concurrent programs portable across a variety of computing platforms, as concurrency is a standardized part of the language and run-time system. Java threads are used in the execution of many Java programs, as the Java garbage collector runs as a separate thread in the JVM.

Although Java has been adopted widely, there remain some difficulties in programming with Java threads. In part, these are just general problems associated with the difference between sequential and concurrent programming. Classes designed for use in sequential programs generally do not use synchronization and may not work well in multithreaded programs. Conversely, code with synchronization may be unnecessarily inefficient in serial programs – synchronization imposes a locking overhead that is wasteful in a single-threaded program.

14.4.1 Threads, Communication, and Synchronization

We can create a Java thread by extending the class Thread, which implements the Runnable interface. When a class extends Thread, it inherits methods to start the thread and throw an exception to the thread:

start: A method that spawns a new thread by causing the virtual machine to execute the run method in parallel with the thread that creates the object.

interrupt: A method that suspends and throws an exception to the thread.

Here is an example subclass of Thread:

```
class PrintMany extends Thread {
    private String msg;
    public PrintMany (String m) {msg = m;}
    public void run() {
            try { for (;;){ System.out.print(msg + " ");
                    sleep(10);
```

```
                        }
                } catch (InterruptedException e) {
                        return;
                }
        }
}
```

Suppose a program creates an object pm = new PrintMany("Hi"). When pm.start is called, a thread is created that prints the string "Hi" over and over. This continues until the thread is interrupted by a call to pm.interrupt.

Originally, Java threads had additional methods that affect their execution, including the following two methods:

suspend: A method that stops execution of the thread until it is later resumed.

stop: A method that forcibly causes the thread to halt.

However, these have now been deprecated because programmers have found that these methods often cause more problems than they solve. The problem with Thread.suspend is that it can contribute to deadlock. If a thread holding a lock on a critical section is suspended, then no other thread can enter the critical section until the target thread is resumed. The problem with Thread.stop is related but almost the opposite. When a thread is stopped by Thread.stop, this causes all the locks it holds to be released. If any of the objects protected by these locks is in an inconsistent state, then other threads may now have access to damaged objects. This can cause the program to behave in unexpected and incorrect ways.

Threads running inside the same virtual machine are scheduled according to their priority. Threads with higher priority are generally executed in preference to threads with lower priority, but there is no guarantee that the highest priority thread will always be run. A Java thread inherits the priority of the thread that creates it and may be changed with the setPriority method.

Communication between threads

In a sense, Java does not make any special provisions for communication: Java threads communicate by assigning to shared variables or calling the methods of shared objects. By themselves, shared variables and shared objects can be problematic, as discussed in Subsection 14.1.3. Java does provide specific synchronization mechanisms that make access to shared objects safe.

Synchronization Primitives

Java synchronization is based on three basic mechanisms:

- Locks: Every object has a lock, used for mutual exclusion.
- Wait sets: Every object has a wait set, providing a form of semaphore.
- Thread termination: A process can pause until another thread terminates.

Locks are tested and set by synchronized blocks and methods, wait sets are used by methods wait, notify, and notifyAll that are defined on all objects, and thread termination is used by the join method on thread objects.

Locks are used by the synchronized statement, which has the form

```
synchronized( object ) {
    statements
}
```

When executed, this statement computes a reference to the indicated object and locks its associated lock before executing the body of the statement. If the lock is already locked by another thread, the thread waits for the lock to become available. After executing the body of the statement, the thread releases the lock. Inside a method, it is common to use this as the lock object. However, it is possible to synchronize on the lock of any object.

Java locks are a *reentrant*, meaning that a thread may acquire a lock multiple times. This may occur if a thread executes one synchronized statement inside another or with synchronized methods that call each other. For example, the synchronized method f in the following class calls the synchronized method g:

```
public class LockTwice {
    public synchronized void f() { g(); }
    public synchronized void g() { System.out.println("Method g() called");}
}
```

When f is called, the calling thread acquires the lock for the LockTwice object. When f calls g, the thread attempts to acquire the same lock again. Because Java locks are reentrant, the calling thread may acquire the LockTwice object's lock again and both methods terminate and return. In systems that do not support reentrant locks, a second call to a synchronized method would deadlock. The semaphore implementation discussed in Subsection 14.1.4 is *not* reentrant.

The methods wait, notify, and notifyAll of class Object use the wait set of an object to provide concurrency control. A thread may suspend itself by using wait. A suspended thread does not execute any statements until another thread awakens it by using notify. The wait/notify combination can be used when threads have a producer–consumer relationship, for example. Because wait notifies the virtual machine scheduler that a process should be suspended, wait and notify are more efficient than busy waiting, which would involve a loop that repeatedly tests to see if a thread should proceed.

The notifyAll method is similar to notify, except that every thread in the wait set for the object is removed and reenabled for thread scheduling. Most Java books recommend calling the notify or notifyAll method of an object only after the calling thread has locked the object's lock.

The third form of synchronization, pausing until another thread terminates, is provided by the join method. Here is a short code example illustrating its use:

```
class Compute_thread extends Thread {
          private int result;
          public void run() { result = f(...); }
          public int getResult() { return result;}
}
...
Compute_thread t = new Compute_thread;
t.start()              // start compute thread
...
t.join(); x = t.getResult();   // wait and get result
...
```

In this code, a Compute_thread object does some kind of computation to determine some result. The program creates a Compute_thread object and starts the thread running so that it will compute a useful result. Later, when the result is needed, the program calls t.join() to wait for the Compute_thread to finish before requesting the result.

14.4.2 Synchronized Methods

Java threads typically communicate by calling methods on shared objects, with synchronization used to provide mutual exclusion. For example, if a transaction manager allows only one thread to commit at a time, the class defining a transaction manager might look like this:

```
class TransactionManager { ...
     public synchronized void commitTransaction(...) {...}
...
}
```

When a method is declared synchronized, the effect is the same as placing the body of the method inside a synchronized statement, using the lock on the object that contains the method. Therefore, even if several threads have access to a TransactionManager object m, only one thread can execute m.commitTransaction(...) at a time. If one thread invokes m.commitTransaction(...) while another thread is in the process of doing so, the second thread will block and wait until the lock on object m is free.

A class may have some synchronized methods and some methods that are not synchronized. If a method is not declared synchronized, then more than one thread may call the unsynchronized method simultaneously. If two threads call an unsynchronized method at the same time, then each thread will have an activation record for a method call on its own run-time stack. However, the two activation records will point to the same (shared) object fields on the run-time heap, in effect allowing the threads to interact through shared variables. In general, a programmer may wish to leave as many methods unsynchronized as possible, allowing for maximal concurrency. However, care must be taken to synchronize enough methods to avoid placing an object in an inconsistent state.

Here is a simple example of a class with two synchronized methods and an unsynchronized method (similar LinkedCell classes appear in various books and articles on Java threads and on web pages with Java program examples):

```
class LinkedCell {   // Lisp-style cons cell containing
    protected double value;   // value and link to next cell
    protected LinkedCell next;
    public LinkedCell (double v, LinkedCell t) {
            value = v; next = t;
    }
    public synchronized double getValue() {
            return value;
    }
    public synchronized void setValue(double v) {
            value = v;   // assignment not atomic
    }
    public LinkedCell next() {   // no synch needed
            return next;
    }
}
```

This class implements a standard linked cell, as might be used to implement stacks, queues, or singly linked lists, with one data field and a next field that can be used to link one linked cell to the next. Note that the data has type double. This is important (for illustrating the need for synchronization) because assignment of doubles is not atomic. Because doubles are represented by 8 bytes and 32-bit processors normally move 4 bytes at a time, it takes two separate instructions to set or read a double.

If setValue were not synchronized and two threads called value setValue simultaneouosly, this could put the LinkedCell object in an *inconsistent state*. Specifically, consider two calls, c.setValue(x) and c. setValue(y), where x and y are doubles. One possible interleaving of steps in the two assignments to c.value is

Set first half of c.value to first half of x
Set first half of c.value to first half of y
Set second half of c.value to second half of y
Set first half of c.value to second half of x

After execution in this order, c.value does not contain either x or y, but rather contains some strange number it determined by combining some bits from the representation of x with some bits from the representation of y. Because the intended effect of setValue(z) is to place the value z in the cell, interleaved execution is inconsistent with the intended semantics of the method. It is necessary to synchronize calls to setValue.

The reason for synchronizing getValue is similar. If one thread calls getValue while another calls setValue, then the double returned by the call to getValue could contain half of the bits that were in the cell before setValue took effect and half of the bits after. To prevent getValue and setValue from running simultaneously, we synchronize both. More generally, although it is possible for two threads to read a data structure simultaneously, errors can occur if a read occurs during a write. The simplest way to prevent these errors is to lock out reads while a write is occurring. A more sophisticated approach, which may be worth the effort in some applications, is to use separate read and write locks, with one read allowing any number of other reads to proceed, but preventing any write.

The next method is not synchronized because, when two threads call next simultaneously, they both get the same correct answer. There is no method to change the next link of a cell, and pointer assignment is atomic in Java anyway.

Synchronized Methods and Inheritance

The synchronized designation is not part of the method signature, and therefore a subclass may override a synchronized method with an unsynchronized method. This programmer flexibility can be useful. However, it is also possible to destroy completely the locking policy of a class through inheritance and subtyping. Therefore, programmers must be very careful when extending classes in multithreaded programs.

14.4.3 Virtual Machine and Memory Model

General properties of the JVM are discussed in Section 13.4. In the Java architecture, Java source code is compiled to an intermediate form called the Java bytecode, which is then interpreted by the JVM. When a Java program creates a new thread, this thread runs inside the JVM. As mentioned in Section 13.4, each Java thread has its own stack, but all threads share same heap. Therefore, when two threads call a method of the same object, the activation records associated with the method calls are on separate stacks, but the fields of the shared object reside in a single heap accessible to both threads. Although the basic Java virtual machine is designed to run on a single processor, multiprocessor implementations have also been developed and used.

In this subsection, we discuss two aspects of the JVM that are relevant to concurrency: concurrent garbage collection and the memory model used to manage reads and writes to shared locations.

Concurrent Garbage Collection

The JVM uses concurrent garbage collection. One advantage is performance. In sequential garbage collection, the program must stop long enough for the garbage collector to identify and reclaim unused memory. With concurrent garbage collection, the garbage collector is designed to run as other threads execute, keeping accessible memory in a consistent state so that other threads can do useful work. When properly tuned, this approach may keep an interactive program from exhibiting long pauses. Instead of pausing long enough to do a complete collection, the scheduler can switch back and forth between the garbage collector and a thread interacting with a user in a way that keeps the user from noticing any long garbage-collection pauses.

The actual design and implementation of concurrent garbage collectors is complicated and involves concepts that are not covered in this book. If you want to gain some appreciation for the issues involved, imagine running a mark-and-sweep algorithm in parallel with a program that modifies memory. As the garbage collector searches for reachable locations, starting from pointers stored on the stack, the program may modify the stack and modify pointers in reachable data. This makes it difficult to understand whether a cell that has not been marked as reachable is truly garbage. In fact, most garbage-collection algorithms, once started, must either finish the task or abandon all their work. Incremental and concurrent garbage collectors generally involve a concept called a *barrier*, which prevents threads from writing to areas of memory that are in use by the garbage collector.

Java Memory Model

The Java memory model is a specification of how shared memory can be cached and accessed by multiple threads. The specification is called a memory *model* because there are many possible implementations of the JVM. The memory model is a set of properties that every implementation must have, including both single-processor and multiprocessor implementations. The original memory model, presented in the *Java Language Specification*, has proven difficult to understand and unsatisfactory for implementers in various ways. Although it is likely that the memory model will eventually be revised to correct some of the known problems, we discuss the original design briefly in order to understand some of the subtle basic issues involved in concurrent memory access.

In the JVM, all threads share the same heap, which will be referred to as *shared memory* for the rest of this section. The virtual machine specification allows each thread to have a local memory, which will be referred to as the *thread cache*. The reason for each thread cache is to avoid coordinated reads and writes to shared memory when possible. If each thread can proceed independently, reading and writing to its own cache, this reduces the need for synchronization between threads.

Because there are several kinds of reads and writes in the memory model, it is useful to establish some precise terminology:

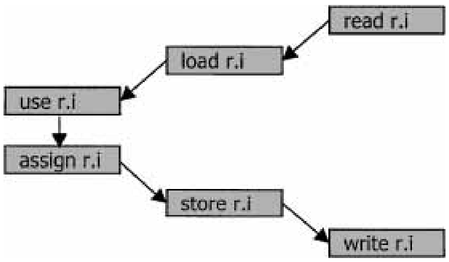
- *use* and *assign* are actions that read and write the thread cache,
- *load* and *store* are actions that transfer values between shared memory and cache,
- *read* and *write* are actions that read and write shared memory.

Bytecode programs execute instructions like getfield, which loads the value stored in a field of an object onto the operand stack (see Subsection 13.4.4) and putfield, which stores a value in the field of an object. In our discussion of the Java memory model, we do not need to distinguish among a getfield instruction, a getstatic instruction (which loads a value from a static field of a class), and array load instructions; we refer to all of these as *use* actions. Similarly, *assign* may be putfield, putstatic, or an array store instruction.

For example, consider a thread executing a compiled bytecode from this source code:

```
r.i = r.i+1;
```

In general, six memory actions will occur as a result of executing this statement. These actions may be interleaved with other actions by the same thread, as long as the order of actions conforms to the memory model. Here is an illustration showing only the six actions directly related to this assignment, in their expected order:



Before the thread computes the sum r.i+1, the value of field i of object r must be read from shared memory, loaded into the local cache of the thread, and used (placed on the operand stack of the thread). Similar steps occur in reverse order to place the value of r.i in shared memory. In the drawing, the left column shows actions that read and write the thread cache, the center column shows actions that transfer values between the thread cache and shared memory, and the right column shows actions that read and write shared memory. Although the effect of *store* r.i, for example, is to place a value in shared memory, the memory model includes separate *store* and *write* actions because the steps that transfer data may not take place atomically. The *store* action indicates the point when the value is read from the thread cache, and the *write* action indicates the point at which the value is actually placed shared memory. If another *assign* occurs after the *store* but before the *write*, it will change the value in the thread cache, but not the value that is placed in shared memory. An important goal in the design of the Java memory model is called *coherence*: For each variable, the uses and assigns to the variable should appear as if they acted directly on shared memory in an order that matches the order within each thread. In other words, threads may proceed independently. In the absence of synchronization, an assignment to a variable in one thread may occur before or after an assignment or use in another thread. However, the cache must be essentially invisible; the program should behave as if the actions operated on shared memory directly. Although coherence is a goal, it is not clear that the goal is actually achieved by the design.

The memory model specifies that each new thread starts with an empty local cache and, when a new variable is created, the variable is created in shared memory, not the thread cache. Here are the main constraints relating *use*, *assign*, *load*, and *store*:

- *Use* and *assign* actions by a thread must occur in the order specified by the program.
- A thread is not permitted to lose its most recent assign (more precisely, a *store*

action by thread *T* on variable *V* must occur between an *assign* by *T* of *V* and a subsequent *load* by *T* of *V*).

- A thread is not permitted to write data from its cache to shared memory for no reason (more precisely, an *assign* action by thread *T* on variable *V* must be scheduled to occur between a *load* or *store* by *T* of *V* and a subsequent *store* by *T* of *V*).

There are similar constraints relating *load*, *store*, *read*, and *write*:

- For every *load*, there must be a preceding *read* action.
- For every *store*, there must be a following *write* action,
- Actions on master copy of a variable are performed by the shared memory in the order requested by thread.

The *Java Language Specification* gives the implementer complete freedom beyond these constraints, stating that "Provided that all the constraints are obeyed, a *load* or *store* action may be issued at any time by any thread on any variable, at the whim of the implementation."

A complication in the Java memory model is a provision for so-called *prescient stores*, which are *store* actions that occur earlier than would otherwise be permitted. The purpose of allowing a prescient store is to allow Java compilers to perform certain optimizations that make properly synchronized programs run faster, but might allow some memory actions to execute out of order in programs that are not properly synchronized. In other words, the memory model lets some compilers choose to execute store actions early under conditions that are intended to preserve coherence for properly structured programs.

Here is the outline of a simple program (devised by William Pugh). The Java Memory Model is Futally Flawed, *Concurrency: Practice and Experience*, 12(6) 2000, pp. 445–455), whose behavior is affected by the relaxed prescient store rule (this is obviously not Java syntax; the idea is that we have a program that initializes x and y to 0, and then spawns two threads that each execute two assignments):

```
x = 0; y = 0;
Thread 1:     a = x; y = 1;
Thread 2:     b = y; x = 1;
```

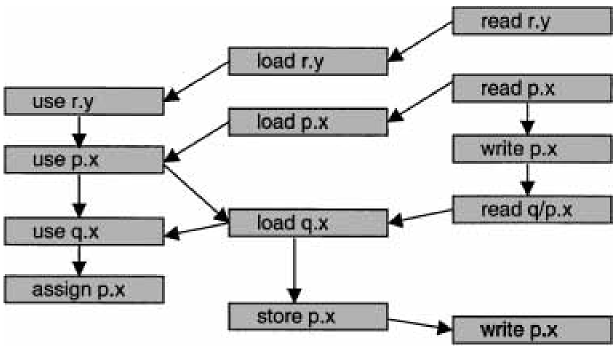
Without prescient stores, the two assignments in each thread must occur in the order shown. As a result, after both threads are executed, variables a and b can both be 0, or one can be 0 and the other 1. With prescient stores, the store actions can be done before the assign actions. This makes it legal for the write actions for both x and y to come before either of the read actions, and the threads can finish with a==b==1. The prescient store rule does not allow the load and store actions for a particular variable to be reordered, but a store involving variable y can be done before an assignment involving a and x.

William Pugh has studied the Java memory model (see paper mentioned above) Here is a simple program that, by his analysis, shows that the memory model is more restrictive than necessary:

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x from another thread occurs here
k = q.x;
p.x = 42;
```

When executed in a manner consistent with the Java memory model, Pugh's analysis shows that the Java memory model *forces* a write to p.x from another thread to change the value of q.x in the thread cache, giving j and k different values in this program.

Here is a drawing showing the ordering of events required by the Java memory model. All of the actions in the drawing come from the preceding code, except for the write action halfway down the right-hand column, which comes from another thread:



Although it takes a little time, it is possible to check that this ordering between actions satisfies the constraints that make up the Java memory model. For example, all use and assign actions, which are in the left-hand column, are in the order corresponding to the assignments in the Java source code. Each use is preceded by a load that initializes the local cache, and similarly each load has a preceding read from shared memory.

The problem with this situation is that it forces a change in a thread cache that does not seem intuitively necessary. A programmer who writes this code and knows that p and q are aliases would be perfectly happy to have j and k get the same value. However, the implementation is forced to do extra work, setting the thread cache after the write from another thread. This requirement makes it more difficult to implement the Java memory model than it needs to be. The problem is particularly significant for multiprocessor implementations, as multiprocessors may reorder instructions for efficiency in situations like this. In fact, common single-processor implementations also appear not to enforce the full Java memory model correctly.

Although the details of the Java memory model are complicated, it should be clear from our brief discussion that it is difficult to design a good memory model. The two problems are that the memory model for any concurrent programming language with shared variables must be easy for programmers to understand so they can write correct and efficient programs that take advantage of concurrency. At the same time, the memory model must be implementable. This is particularly tricky for multiprocessors that may have their own form of processor caches and shared memory. Because Java is the first programming language that integrates concurrency and is widely used for general programming, it is likely that further experience with Java will lead to better understanding of memory models for concurrent languages.

14.4.4 Distributed Programming and Remote Method Invocation

RMI allows a thread running on one JVM to call a method of an object on another virtual machine across the network. This communication mechanism is one of the basic building blocks of distributed Java programming. Because RMI is an asymmetric operation (one thread executes a method call, the other does not), it is convenient to refer to the two threads by two different names. We will refer to the caller as the *client* and the thread providing the remote object as the *server*.

A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. There is a basic chicken-and-egg problem, however. Once we have some established connection between client and server, we can use this connection to pass references to objects. But how do we make the first connection between two systems? A solution used in distributed Java programming is the RMI registry, which is a remotely accessible directory of available objects.

Java provides three general facilities for distributed programming:

- *Locating remote objects*: A server can register its remote objects by using the RMI registry or the server can pass object references through other connections already established by the programs.
- *Communicating with remote objects*: Communication between remote objects is handled by RMI, which looks like standard Java method invocation to the programmer.
- *Transmitting objects across the network*: An object can be sent as an argument to a remote call or as the return value of such a call. When a Java object is transmitted, one virtual machine transmits the bytecode to execute the object and associated data to another. The receiving virtual machine loads the bytecode and creates the object.

Creating Remote Objects

There are several steps involved in creating an object that can be accessed remotely. The first step is to define a class that implements a *remote interface*, which is an interface that extends the interface `java.rmi.Remote`. Each method of the remote interface must declare `java.rmi.RemoteException` in its throws clause, in addition to any other exceptions. This allows a remote client to throw a remote exception to any object of this class.

When an object from a class that implements a remote interface is passed from the server virtual machine to a client virtual machine, RMI passes a remote *stub* for the object. The stub object has the same methods as the original object sitting on the server, but calls to methods on the stub object are sent across the network and executed on the server object. In the client program, all of this looks like regular Java programming: A call to an object that happens to be remote returns the stub of another object, and subsequent calls to methods on that object are transmitted across the network and executed on the server.

Compiling the server code requires two steps. The first step is to use the regular Java compiler to compile the source files, which contain the implementation of the remote interfaces and implementations and the server classes. The second step uses a second compiler to create stubs for the remote objects.

Dynamic Class Loading

An important feature of Java RMI is the way that the bytecode is transmitted across the network. This is possible because Java provides a uniform execution environment, the JVM, that runs standard bytecode regardless of the kind of hardware or operating system used. When RMI is used to pass an object from one virtual machine to another, the bytecode for the class of the object is transmitted and loaded, if the class has not already been installed on the receiving machine. This allows new types to be introduced into a remote virtual machine, extending the behavior of a program dynamically.

The RMI Registry

The RMI registry, running on a server, allows remote clients to get a reference to a remote object. The registry is typically used to locate only the first remote object that a client needs, as methods of the first object can return other remote objects.

The RMI system allows a programmer to bind a URL-formatted name of the form `//host/objectname` to a remote object. Once a remote object is registered on the server, callers can look up the object by name, obtain a remote object reference, and then remotely invoke methods on the object. For example, if `helo` is an object that can be called remotely to say "hello world," the following code binds the name `HelloWorld` to a reference to the helo object:

```
Naming.rebind("//machine.stanford.edu/HelloWorld", helo);
```

This will allow a client program to get a stub for `helo` by a network access to the host (in this case machine.stanford.edu). There are several details involved in setting this all up properly that need not concern us. The main point is that a Java thread can get an object from a server across the network by using a symbolic name for the server and the object, and this is enough to let a client thread get more remote objects from this server.

14.5 CHAPTER SUMMARY

In this chapter, we discussed some general aspects of concurrent and distributed programming and looked at three example languages: Actor systems, CML, and Java features providing threads, synchronization, and RMI. These three languages were chosen to illustrate different general approaches to communication and synchronization. Actors and CML are based on message passing, whereas Java threads communicate by shared variables and shared objects that may have one or more synchronized methods. Threads are implicit in actor systems, arising from concurrently sending more than one message to a set of concurrently reactive actors, whereas CML and Java programs must explicitly create threads.

General Issues in Concurrency

Concurrent programming languages define concurrent actions that may be executed simultaneously on several processors or in an interleaved fashion on a single processor. Concurrent programs generally have nondeterministic behavior, as the order of execution may depend on the speed of communication or the order in which process are scheduled. Independent threads may communicate in order to compute a useful result and coordinate to avoid problems associated with shared resources. When two threads attempt to update a shared data structure, for example, some coordination may be needed to keep the actions of each thread from interfering with the other. Without coordination, assignments by one thread may cancel assignments of another and leave the data structure in a state that is not meaningful to the program.

Locking, semaphores, and monitors are traditional operating systems mechanisms for coordinating the actions of independent processes. A lock is a kind of shared variable that may be set to indicate that a resource is locked or unlocked. When one process locks a resource, other processes should not access the resource. One problem with locking is that testing and setting a lock must be atomic. For this reason, locks with atomic test-and-set are often provided by hardware.

Semaphores structure a set of waiting processes into a list that may be given access to the shared resource when it becomes available. Processes using a semaphore must call *wait* before accessing the protected resource and *signal* when they are done. Both *wait* and *signal* are atomic actions that may be implemented by the operating system (or programming language run-time system) using locks. Monitors group synchronization with operations on data, eliminating the need for the calling process to call *wait* and *signal*.

Concurrent programs communicate by using shared resources or message passing. In principle, message-passing mechanisms can be synchronous or asynchronous, buffered or unbuffered, and preserve or not preserve transmission order. Actors and CML differ in all three ways, illustrating two very different sets of communication assumptions. As explored in the exercises, some of the remaining combinations are sensible language design choices and some are either not sensible or too weak for many programming problems.

	Synchronous		**Asynchronous**	
	Ordered	**Unordered**	**Ordered**	**Unordered**
Buffered				Actors
Unbuffered	CML			

Actor Systems

An actor is a form of reactive object that performs a computation in response to a message and may perform one or more of the following basic actions:

- Sending communication to itself or other actors,
- Creating actors,
- Specifying a replacement behavior.

Actors do not have any shared state, but use buffered asynchronous message passing for all communication. Actors change state atomically. There are no changes in the internal state of an actor while it is responding to a message. Instead, an actor changes state atomically as it completes its computation and specifies a replacement behavior. Some interesting aspects of the actor model are the mail system, which buffers and delivers messages, and the challenges of programming with only asynchronous communication without order guarantees.

Concurrent ML

CML is a concurrent extension of Standard ML that provides dynamic thread creation and synchronous message passing on typed channels. The most interesting aspects of CML, are the ways that communication and coordination are combined and the way that CML allows programmers to define their own communication and synchronization abstractions by using events. Events represent synchronous actions that are performed when a program synchronizes on them. CML event primitives can be used to implement selective computation, as illustrated by Dijkstra's guarded commands, and a variety of communication mechanisms such as multicast channels and various forms of protected shared variables.

Java Threads and Synchronization

Java makes concurrent programs portable across a variety of computing platforms. We create a Java thread object by defining a class with a run() method, either by extending the Thread class or by implementing the Runnable interface directly. When a thread object is activated, the run method runs as a separate thread inside the JVM. Java threads generally communicate by calling methods of shared objects. The Java language provides several synchronization primitives and supports monitors directly through objects with synchronized methods.

Java synchronization is achieved through three basic mechanisms:

- Locks: Every object has a lock, used for mutual exclusion.
- Wait sets: Every object has a wait set, providing a form of semaphore.
- Thread termination: A process can pause until another thread terminates.

Locks are tested and set by synchronized blocks and methods, wait sets are used by methods wait, notify, and notifyAll that are defined on all objects, and thread termination is used by the join method on thread objects.

The Java memory model is a specification of how shared memory can be cached and accessed by multiple threads. The original memory model, presented in *Java Language Specification*, has proven difficult to understand and unsatisfactory for implementers in various ways. Although it is likely that the memory model will eventually be revised to correct some of the known problems, we discussed the memory model briefly in order to understand some of the subtle basic issues involved in concurrent memory access.

Distributed Java Programming and Remote Method Invocation

RMI allows a thread running on one JVM to call a method of an object on another virtual machine across the network. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A server may make objects accessible by using the RMI registry, which is a remotely accessible directory of available objects. When RMI is used to pass an object from one virtual machine to another, the bytecode for the class of the object is transmitted and loaded if the class has not already been installed on the receiving machine. This allows new types to be introduced into a remote virtual machine, extending the behavior of a program dynamically.

EXERCISES

14.1 Mutual Exclusion

Example 14.2 gives an incorrect implementation of *wait* and *signal* by using an ordinary integer variable lock as a "lock." Describe an execution of the code in Example 14.2 that leads to an inconsistent state. Support your description by giving the initial values of the variable n and array list and list in order the assignments and tests that occur in executing the code. Explain why atomic implementations of *wait* and *signal* would prevent this execution sequence.

14.2 Fairness

The guarded-command looping construct

```
do
    Condition ⇒ Command
    . . .
    Condition ⇒ Command
od
```

involves nondeterministic choice, as explained in the text. An important theoretical concept related to potentially nonterminating nondeterministic computation is *fairness*. If a loop repeats indefinitely, then a fair nondeterministic choice must eventually select each command whose guard is true. For example, in the loop

```
do
    true ⇒ x := x+1
    true ⇒ x := x-1
od
```

both commands have guards that are always true. It would be *unfair* to execute x := x+1 repeatedly without ever executing x := x-1. Most language implementations are designed to provide fairness, usually by providing a bounded form. For example, if there are n guarded commands, then the implementation may guarantee that each enabled command will be executed at least once in every $2n$ or $3n$ times through the loop. Because the number $2n$ or $3n$ is implementation dependent, though, programmers should assume only that each command with a true guard will eventually be executed.

(a) Suppose that an integer variable x can contain an integer value with only absolute value less than INTMAX. Will the preceding do . . . od loop cause overflow or underflow under a fair implementation? What about an implementation that is not fair?

(b) What property of the following loop is true under a fair implementation but false under an unfair implementation?

```
go := true;
n := 0;
do
    go => n := n+1
    go => g := false
od
```

(c) Is fairness easier to provide on a single-processor language implementation or on a multiprocessor? Discuss briefly.

14.3 Actor Computing

The actor mail system provides asynchronous buffered communication and does not guarantee that messages (*tasks* in actor terminology) are delivered in the order they are sent. Suppose actor A sends tasks $t_1, t_2, t_3, \ldots,$ to actor B and we want actor B to process tasks in the order A sends them.

(a) What extra information could be added to each task so that B can tell whether it receives a task out of order? What should B do with a task when it first receives it before actually performing the computation associated with the task?

(b) Because the actor model does not impose any constraints on how soon a task must be delivered, a task could be delayed an arbitrary amount of time. For example, suppose actor A sends tasks $t_1, t_2, t_3, \ldots, t_{100}$ and actor B receives the tasks $t_1, t_3, \ldots, t_{50}$ without receiving task t_2. Because B would like to proceed with some of these tasks, it makes sense for B to ask A to resend task t_2. Describe a protocol for A and B that will add resend requests to the approach you described in part (a) of this problem.

(c) Suppose B wants to do a final action when A has finished sending tasks to B. How can A notify B when A is done? Be sure to consider the fact that if A sends *I'm done* to B after sending task t_{100}, the *I'm done* message may arrive before t_{100}.

14.4 Message Passing

There are eight message-passing combinations involving synchronization, buffering, and message order, as shown in the following table.

	Synchronous		**Asynchronous**	
	Ordered	**Unordered**	**Ordered**	**Unordered**
Buffered				
Unbuffered				

For each combination, give a programming language that uses this combination and explain why the combination makes some sense, or explain why you think the combination is either meaningless or too weak for useful concurrent programming.

14.5 CML Events

The CML recv function can be defined from recvEvt by

```
fun recv(ch) = sync (recvEvt (ch));
```

as shown in the text. Give a similar definition of send by using sendEvt and explain your definition in a few sentences.

14.6 Concurrent Access to Objects

This question asks about synchronizing methods for stack and queue objects.

(a) Bounded stacks can be defined as objects, each containing an array of up to *n* items. Here is a pseudocode for one form of stack class.

```
class Stack
  private
    contents : array[1..n] of int
    top : int
  constructor
    stack () = top := 0
  public
    push (x:int) : unit =
      if top < n then
        top := top + 1;
        contents[top] := x
      else raise stack_full;
    pop ( ) : int =
      if top > 0 then
        top := top - 1:
        return contents[top+1]
      else raise stack_empty;
end Stack
```

If stacks are going to be used in a concurrent programming language, what problem might occur if two threads invoke push and pop simultaneously? Explain.

(b) How would you solve this problem by using Java concurrency concepts? Explain.

(c) Suppose that, instead of stacks, we have queues:

```
class Queue
  private
    contents : array[1..n] of int
    front, back : int
  constructor
    queue() = front := back := 1
  public
    insert (x:int) : unit =
      if back+1 mod n != front then
        back := back+1 mod n;
        contents[back] := x
      else raise queue_full;
    remove ( ) : int =
      if front != back then
        front := front+1 mod n;
        return contents[front]
      else raise queue_empty;
end Queue
```

Suppose that five elements have been inserted into a queue object and none of them have been removed. Do we have the same concurrency problem as we did with push and pop when one thread invokes insert and another thread simultaneously invokes remove? Assume that *n* is 10. Explain.

14.7 Java Synchronized Objects

This question asks about the following Java implementation of a bounded buffer. A bounded buffer is a FIFO data structure that can be accessed by multiple threads:

```
class BoundedBuffer {
   // designed for multiple producer threads and
   // multiple consumer threads
   protected int numSlots = 0;
   protected int[] buffer = null;
   protected int putIn = 0, takeOut = 0;
   protected int count = 0;
   public BoundedBuffer(int numSlots) {
       if (numSlots <= 0)
          throw new IllegalArgumentException("numSlots <= 0");
       this.numSlots = numSlots;
       buffer = new int[numSlots];
   }
   public synchronized void put(int value)
          throws InterruptedException {
       while (count == numSlots) wait();
       buffer[putIn] = value;
       putIn = (putIn + 1) % numSlots;
       count++;
       notifyAll();
   }
   public synchronized int get()
          throws InterruptedException {
       int value;
       while (count == 0) wait();
       value = buffer[takeOut];
       takeOut = (takeOut + 1) % numSlots;
       count--;
       notifyAll();
       return value;
   }
}
```

(a) What is the purpose of while (count == numSlots) wait() in put?

(b) What does notifyAll() do in this code?

(c) Describe one way that the buffer would fail to work properly if all synchronization code is removed from put.

(d) Suppose a programmer wants to alter this implementation so that one thread can call put at the same time as another calls get. This causes a problem in some situation but not in others. Assume that some locking may be done at entry to put and get to make sure the concurrent-execution test is satisfied. You may also assume that increment or decrement of an integer variable is atomic and that only one call to get and one call to put may be executed at any given time. What test involving putIn and takeOut can be used to decide whether put and get can proceed concurrently?

(e) The changes in part (d) will improve performance of the buffer. List one reason that leads to this performance advantage. Despite this win, some programmers may choose to use the original method anyway. List one reason why they might make this choice.

14.8 Resources and Java Garbage Collection

Suppose we are writing an application that uses a video camera that is attached to the computer. Our application, written in Java, has multiple threads, which means that separate parts of the application may run concurrently. The camera is a shared resource that can be used by only one thread at a time, and our multithreaded application may try to use the camera concurrently from multiple threads.

The camera library (provided by the camera manufacturer) contains methods that will ensure that only one thread can use the camera at a time. These methods are called

```
camera.AcquireCamera()
camera.ReleaseCamera()
```

A thread that tries to acquire the camera while another object has acquired it will be blocked until camera.ReleaseCamera() has been called. When a thread is blocked, it simply stops without executing any further commands until it becomes unblocked.

You decide to structure your code so that you create a MyCamera object whenever a thread wants to use the camera, and you “delete” the object (by leaving the scope that contains a pointer to it) when that thread is done with the camera. The object calls camera.AcquireCamera() in the constructor and calls camera.ReleaseCamera() in the finalize method:

```
import camera    // imports the camera library
class MyCamera {
    . . .
    MyCamera() {
        . . .
        camera.AcquireCamera();
        . . .
    }
    . . .              // (other methods that use the camera go here)
    finalize() {
        . . .
        camera.ReleaseCamera();
        . . .
    }
}
```

Here is a sample code that would use the Mycamera object:

```
{
...
MyCamera c = new MyCamera();
...  // (code that uses the camera)
}    // end of scope so object is no longer reachable
```

In this question, we will say that a *deadlock* occurs if all threads are waiting to acquire the camera, but camera.ReleaseCamera is never called.

(a) When does camera.ReleaseCamera actually get called?

(b) This code can cause a deadlock situation in some Java implementations. Explain how.

(c) Does calling the garbage collector by using Runtime.getRuntime().gc() after leaving the scope where the camera is reachable solve this problem?

(d) How can you fix this problem by modifying your program (without trying to force garbage collection or by using synchronized) so deadlock will not occur?

(e) Suppose you have a multithreaded Java implementation with the garbage collector running concurrently as a separate thread. Assume the garbage collector is always running, but it may run slowly in the background if the program is active. This will eventually garbage collect every unreachable object, but not necessarily as soon as it becomes unreachable. Does deadlock, as previously defined, occur (in the preceding original code) in this implementation? Why or why not?

14.9 Separate read and write Synchronization

For many data structures, it is possible to allow multiple reads to occur in parallel, but reads cannot be safely performed while a write is in progress and it is not safe to allow multiple writes simultaneously. Rewrite the Java LinkedCell class given in this chapter to allow multiple simultaneous reads, but prevent reads and writes while a write is in progress. You may want to use more than one lock. For example, you could assume objects called ReadLock and WriteLock and use synchronized statements involving these two objects. Explain your approach and why it works.

14.10 Java Memory Model

This program with two threads is discussed in the text:

```
x = 0; y = 0;
Thread 1: a = x;   y = 1;
Thread 2: b = y;   x = 1;
```

Draw a box-and-arrow illustration showing the order constraints on the memory actions (*read*, *load*, *use*, *assign*, *store*, *write*) associated with the four assignments that appear in the two threads (you do not need to show these actions for the two assginments setting x and y to 0):

(a) Without prescient stores.

(b) With prescient stores.