12 Concurrency

12.1 What is concurrency?

Computers with multiple CPU's can execute a set of programs or program components in *parallel*. A computation can thus be completed in less *elapsed* (clock) time than on a computer with a single CPU, at the expense of additional CPU time for synchronization and communication. It is also possible for several programs to share a computer with a single CPU by switching the CPU so fast from one program to another that it appears that they are executing simultaneously. Even though CPU-switching does not implement true parallelism, it is convenient to design and program software for these systems as if the execution of the programs is actually in parallel. *Concurrency* is a term used for the simultaneous execution of more than one program without specifying if the computation is truly or only apparently parallel.

Concurrency is familiar to most programmers in one or more of these forms:

- *Multi-programming* operating systems enable more than one person to use the computer at the same time. Such *time-sharing* systems implemented on mainframe computers and minicomputers were for many years the only way of supplying computational facilities to a large community like a university.
- *Multi-tasking* operating systems enable components of a single program (or programs of a single user) to proceed concurrently. With the advent of personal computers, multi-programming computers are less common, but even a single person will often need multi-tasking to perform different tasks simultaneously, such as printing in the background while interactively writing a paper.
- *Embedded systems* in factories, transportation systems and medical facilities control sets of sensors and actuators in "real-time". These systems are characterized by the requirement that they perform relatively small amounts of computations at very frequent intervals: each sensor needs to be read and interpreted, then the program must decide upon an appropriate action, and finally items of data are formatted and sent to the actuators. To implement embedded systems, multi-tasking operating systems are used to coordinate among dozens of separate computations.

Designing and programming a concurrent system is extremely difficult, and entire textbooks are devoted to various aspects of the problem: system architecture, task scheduling, hardware interfaces, etc. This purpose of this section is to give an overview of *programming language* support for concurrency, which traditionally has been provided by hardware and operating system functions.

A *concurrent program* consists of one or more program components called *processes* that can potentially be executed in parallel. Concurrent programs have to deal with two problems:

Synchronization Even though the processes execute concurrently, occasionally one process will have to synchronize execution with other processes. The most important form of synchronization is mutual exclusion: two processes must not access the same resource (such as a disk or a shared table) simultaneously.

Communication Processes are not totally independent; they must send data to each other. For example, in a flight control program the process that reads the altitude sensor must send the result to the process that computes the autopilot algorithms.

12.2 Shared memory

The simplest model for concurrent programming is the *shared-memory* model (Figure 12.1). Two

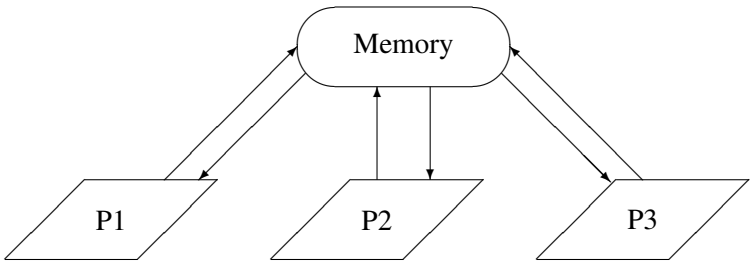

Figure 12.1: Shared-memory model for concurrency

or more processes can access the same area in memory (though they may also have their own private memory). Suppose that we have two processes that are trying to update the same variable in shared-memory:[1]

Ada

```
procedure Main is
   N: Integer := 0;
   task T1;
   task T2;

   task body T1 is
   begin
      for I in 1..100 loop N := N + 1; end loop;
```

[1] In Ada processes are called *tasks*.

```
    end T1;

    task body T2 is
    begin
       for I in 1..100 loop N := N + 1; end loop;
    end T2;

begin
    null;
end Main;
```

Consider now the implementation of the assignment statement:

```
load    R1,N      Load from memory
add     R1,#1     Increment register
store   R1,N      Store to memory
```

If each execution of the loop body in T1 is completed before T2 executes its loop body, N will be incremented 200 times. However, each task may be executed on a separate computer with a separate set of registers. In this case, the following sequence of events may take place:

- T1 loads N into its register R1 (value is n).
- T2 loads N into its register R1 (value is n).
- T1 increments R1 (value is $n+1$).
- T2 increments R1 (value is $n+1$).
- T1 stores the contents of its R1 into N (value is $n+1$).
- T2 stores the contents of its R1 into N (value is $n+1$).

The result will be that the execution of the two loop bodies will only increment N by one. The final value of N can be anywhere between 100 and 200, depending on the relative speeds of the two processors.

It is important to realize that this can occur even on a computer that implements multi-tasking by sharing a single CPU. When the CPU is switched from one process to another, the registers that the blocked process used are saved and then restored when it is allowed to continue.

In the theory of concurrency, the execution of a concurrent program is defined as *any interleaving* of the *atomic instructions* of the tasks. An atomic instruction is simply one that cannot be "partially" executed or interrupted to allow another task to proceed. In the shared-memory model of concurrency, load from memory and store to memory are atomic instructions.

Within the framework of interleaved computations, languages and systems that support concurrency differ in the level of atomic instructions that are defined. The implementation of the instruction must ensure that it is executed atomically. In the case of shared-memory load and store, this

is ensured by the memory interface hardware. Higher-level atomic instructions are implemented within the underlying run-time system, supported by special instructions in the CPU.

12.3 The mutual exclusion problem

The *mutual exclusion problem* for concurrent programs is a generalization of the above example. Each one of a set of tasks is assumed to have its computation divided into a *critical section* and a *non-critical section* which are executed repeatedly:

Ada

```
task body T_i is
begin
   loop
      Prologue;
      Critical_Section;
      Epilogue;
      Non_Critical_Section;
   end loop;
end T_i;
```

To solve the mutual exclusion problem, we must find sequences of code (called the *prologue* and the *epilogue*) such that the program satisfies the following requirements, which must hold for *all* interleavings of the instruction sequences of the set of tasks:

Mutual exclusion At most one task is executing its critical section at any time.

No deadlock There is always at least one task that is able to continue its execution.

Liveness If a task wishes to enter its critical section, *eventually* it will do so.

Fairness Access to the critical section is granted "fairly".

There are solutions to the mutual exclusion problem using just load and store as atomic instructions, but the solutions are complicated and beyond the scope of this book, and we refer the reader to textbooks on concurrent programming.

E.W. Dijkstra defined a higher-level synchronization abstraction called a *semaphore* that trivially solves the problem. A semaphore S is a variable which has an integer value; two *atomic* instructions are defined on semaphores:

```
Wait(S): when S > 0 do S := S - 1;
Signal(S):S := S + 1;
```

Wait(S) instruction blocks the process that executes it until the value of S is positive. Note that since the instruction is atomic, once the process checks that S is positive, it decrements S before any other process executes an instruction. Similarly, Signal(S) is executed atomically with no possibility of another process interrupting between the load and the store of S.

The mutual exclusion problem is solved as follows:

Ada

```
procedure Main is
   S: Semaphore := 1;
   task T_i;                    -- One of many
   task body T_i is
   begin
      loop
         Wait(S);
         Critical_Section;
         Signal(S);
         Non_Critical_Section;
      end loop;
   end T_i;
begin
   null;
end Main;
```

We leave it to the reader to check that this is a correct solution. Of course the trivial solution is achieved by placing the burden on the implementor of the run-time system.

12.4 Monitors and protected objects

The problem with the semaphore and similar facilities supplied by the operating system is that they are not structured. If you forget to match each Wait with a Signal the program can lose synchronization or deadlock. To solve the lack of structuring, a concept called *monitors* was defined and implemented in several programming languages. A monitor is a grouping of data and subprograms which has the following properties:

- The data can only be accessed by the subprograms of the monitor.
- At most one monitor subprogram can be executed at any one time. A process attempting to call a monitor procedure while another process is already executing within the monitor will cause the new process to be suspended.

Since all synchronization and communication is done within the monitor, potential concurrency errors are limited to the programming of the monitor itself; user processes cannot cause additional errors. The interface to a monitor is similar to that of an operating system in that a process calls the monitor to request and receive a service. Synchronization among processes is automatically ensured. The disadvantage of the monitor is that it is a centralized facility.

The original concurrency model in Ada (described in Section 12.8 below) is quite sophisticated and it turned out to have too much overhead for simple mutual exclusion problems. To correct this, in Ada 95 a monitor-like facility was introduced called *protected objects*. For example, a semaphore can be simulated as a protected object. This interface defines the two operations, but

keeps the integer value of the semaphore *private*, which means that it is not accessible to the users of the semaphore:

Ada

```
protected type Semaphore is
    entry Wait;
    procedure Signal;
private
    Value: Integer := 1;
end Semaphore;
```

The implementation of the semaphore is:

Ada

```
protected body Semaphore is
    entry Wait when Value > 0 is
    begin
        Value := Value – 1;
    end Wait;
    procedure Signal is
    begin
        Value := Value + 1;
    end Signal;
end Semaphore;
```

The execution of both the entry and the procedure are mutually exclusive: at most one task will execute an operation of a protected object at any time. An entry also has a *barrier*, which is a Boolean expression. A task trying to execute an entry will be blocked if the expression is false. Whenever a protected operation is completed, all barriers will be re-evaluated and a task whose barrier is now true will be allowed to execute. In the example, when Signal increments Value, the barrier on Wait will now be true and a blocked task will be allowed to execute the entry body.

12.5 Concurrency in Java

Ada is one of the few languages that include support for concurrency within the language, rather than delegating the support to the operating system. The Java language follows the Ada philosophy so that concurrent programs can be ported across operating systems. An important application of concurrency in Java is the programming of servers: each client request causes the server to *spawn* a new process to fulfil that request.

The Java construct for concurrent programming is called *threads*. There is no essential difference between threads and the standard term *processes* in concurrency; the distinction is more implementation oriented in that multiple threads are assumed to execute within the same address space. For purposes of developing and analyzing concurrent algorithms, the model is the same as that described in Chapter 12—interleaved execution of atomic instructions of processes.

A Java class that inherits from the class Thread declares a new thread type. To actually create the thread, an allocator must be called, followed by a call to start. This causes the run method in the thread to begin execution:[2]

Java

```
class My_Thread extends Thread
{
   public void run( ) {...};          // Called by start
}

My_Thread t = new My_Thread( );  // Create thread
t.start( );                      // Activate thread
```

One thread can explicitly create, destroy, block and release another thread.

These constructs are similar to the Ada constructs which allow a task type to be defined followed by dynamic creation of tasks.

Synchronization

Java supports a form of synchronization similar to monitors (see Section 12.4). A class may contain methods specified as synchronized. A lock is associated with each object of the type of this class; the lock ensures that only one thread at a time can execute a synchronized method on the object. The following example shows how to declare a monitor to protect a shared resource from simultaneous access by several threads:[3]

Java

```
class Monitor
{
   synchronized public void seize( ) throws InterruptedException
   {
      while (busy) wait( );
      busy = true;
   }

   synchronized public void release( )
   {
      busy = false;
      notify( );
   }

   private boolean busy = false;
}
```

[2] Java uses the keyword extends in place of the colon used by C++.

[3] Since wait can raise an exception, the exception must either be handled within the method, or the method must declare that it throws the exception. In C++, this clause would be optional.

The monitor protects the boolean variable which indicates the state of the resource. If two threads try to execute the seize method of a monitor, only one will successfully acquire the lock and execute. This thread will set busy to true and proceed to use the resource. Upon leaving the method, the lock will be released and the other thread will be allowed to execute seize. Now, however, the value of busy is false. Rather than waste CPU time continuously checking the variable, the thread chooses to release the CPU by calling wait. When the first thread completes the use of the shared resource, it calls notify, which will allow the waiting thread to resume execution of the synchronized method.

The Java constructs for concurrency are quite low-level. There is nothing to compare with the sophisticated Ada rendezvous for direct process-to-process communication. Even when compared with Ada 95 protected objects, Java's constructs are relatively weak:

- A barrier of a protected object is automatically re-evaluated whenever its value could change; in Java, you must explicitly program loops.
- Java maintains simple locks for each object; in Ada, a queue is maintained for each entry. Thus if several Java threads are wait'ing for a synchronized object, you cannot know which one will be released by notify, so starvation-free algorithms are difficult to program.

12.6 Message passing

As computer hardware has become less expensive, *distributed programming* has become important. Programs are decomposed into concurrent components which are executed on separate computers. The shared-memory model is no longer appropriate; instead synchronization and communication are based on *synchronous message passing* (Figure 12.2). In this model, a communica-

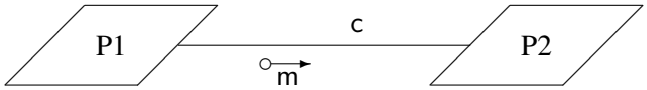

Figure 12.2: Concurrency by message passing (occam)

tions channel c can exist between any two processes. When one process sends a message m on the channel, the process is suspended until the other process is ready to receive. Symmetrically, a process that wishes to receive a message is suspended until the sending process is ready. This suspension is used to synchronize the processes.

The synchronous model of concurrency can be implemented within a programming language, or as an operating system service (pipes, sockets, etc.). Models differ in the ways that processes address each other and in the manner of message passing. We now describe three languages that chose radically different ways to implement synchronous concurrency.

12.7 occam

The synchronous message model was originally developed by C.A.R. Hoare in a formalism called *CSP* (Communicating Sequential Processes). The practical implementation is in the language occam[4] which was developed for programming the *transputer*—a hardware architecture for distributed processing.

In occam, addressing is fixed and message passing is one-way as shown in Figure 12.2. A *channel* has a name and may only be used for sending from one process and receiving on another:

```
CHAN OF INT c :
PAR
  INT m :
  SEQ
    -- Create integer value m
    c ! m
  INT v :
  SEQ
    c ? v
    -- Use integer value in v
```

c is declared as a channel that can pass integers. The channel must be used in exactly two processes: one process contains output instructions (c!) and the other input instructions (c?).

The syntax of occam is interesting. In other languages, the "default" mode of execution is sequential execution of a group of statements and a special notation is used for parallelism. In occam, parallel and sequential computation are considered equally important, so you have to explicitly state using PAR and SEQ how each group of (indented) statements is to be executed.

Even though each channel connects exactly two processes, occam allows a process to simultaneously wait for communications on any one of several channels:

```
[10]CHAN OF INT c :      -- Array of channels
ALT i = 0 FOR 10
  c[i] ? v
  -- Use integer value in v
```

This process waits for communications on any of ten channels and the processing of the received value can depend on the channel index.

The advantage of point-to-point communications is that it is extremely efficient because all addressing information is "compiled-in". No run-time support is required other than to synchronize the processes and move the data; on a transputer this is done in hardware. Of course this efficiency is achieved at the cost of lack of flexibility.

[4] occam is a trademark of INMOS Groups of Companies. Note that the initial letter is not capitalized.

12.8 Ada rendezvous

Ada tasks communicate with each other during a *rendezvous*. One task T1 is said to call an *entry* e in another task T2 (Figure 12.3). The called task must execute an accept-statement for the entry:

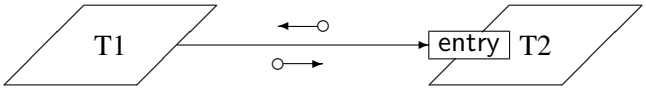

Figure 12.3: Ada tasks and entries

```
accept E(P1: in Integer; P2: out Integer) do
   ...
end E;
```

When a task executes an entry call and there is another task that has executed an accept-statement for the entry, a rendezvous takes place:

- The calling task passes in-parameters to the accepting task and then is blocked.
- The accepting task executes the statements in the accept body.
- The accepting task returns out-parameters to the calling task.
- The calling task is unblocked.

The definition of the rendezvous is symmetrical in that if a task executes an accept-statement and there is no waiting entry call, it will block until some task calls the entry for this accept-statement.

Thus addressing is only in one direction: the calling task must know the name of the accepting task, but the accepting task does not know the name of the calling task. The rationale for this design is to allow the implementation of *servers* which are processes that provide a specific service to any other process. The *client* task must of course know the name of the service it is requesting, while the server task will provide the service to any arbitrary task, and it has no need to know about the client.

A single rendezvous can involve message passing in two directions, because a typical service might be a request for an element from a data structure. The overhead of an additional interaction to return the result would be prohibitive.

The rendezvous mechanism is quite sophisticated: a task can wait simultaneously for different entry calls using the select-statement:

```
select
   accept E1 do ... end E1;
or
   accept E2 do ... end E2;
or
   accept E3 do ... end E3;
end select;
```

Select alternatives can have Boolean expressions called *guards* which enable the task to control which calls it is willing to accept. Timeouts (limiting the time spent waiting for a rendezvous) and polling (checking for an immediate rendezvous) can also be specified. Unlike the ALT construct in occam, an Ada select-statement cannot wait simultaneously on an arbitrary number of entries.

Note the basic difference between protected objects and rendezvous:

- The protected object is a passive mechanism and its operations are executed by other tasks.
- An accept-statement is executed by the task in which it appears, that is, it performs computation on behalf of other tasks.

Rendezvous would be used to program a server if the server had significant processing to do in *addition* to communication with a client:

```
task Server is
begin
   loop
      select
         accept Put(I: in Item) do
            -- Save I in data structure
         end Put;
      or
         accept Get(I: out Item) do
            -- Retrieve I from data structure
         end Get;
      end select;
      -- Maintain data structure
   end loop;
end Server;
```

The server saves and retrieves items from a data structure, but after each operation it perform additional processing on the data structure, for example, logging the modifications. There is no reason to block other tasks while this time-consuming processing is being done.

The Ada concurrency mechanism is extremely flexible, but the flexibility is bought at the price of less efficient communication than with the occam point-to-point channels. On the other hand, it is practically impossible to implement a flexible server process in occam, since each additional client process needs a separate named channel, and this requires modification of the server code.

12.9 Linda

Linda is not a programming language as such but a model for concurrency that can be added to an existing programming language. Instead of one-way (Ada) or two-way (occam) addressing, Linda does not use any addressing between concurrent processes! Instead a process can choose to send

a message to a global *Tuple Space*. The name is derived from the fact that each message is a tuple, that is a sequence of one or more values of possibly different types. For example:

```
(True, 5.6, 'C', False)
```

is a 4-tuple consisting of values that are Boolean, float, character and again Boolean.

There are three operations that access the Tuple Space:

out Put a tuple in tuple space.

in Block until a matching tuple exists, then remove it (Figure 12.4).

read Block until a matching tuple exists (but do not remove it).

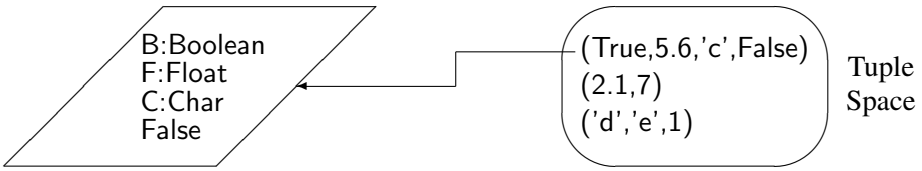

Figure 12.4: Linda tuple-space

Synchronization is achieved because the in- and read-commands must specify a tuple-signature: the number of elements and their types. Only if a tuple exists with a matching signature can the receive operation be executed; otherwise, the process is suspended. In addition, one or more elements of the tuple can be explicitly specified. If a value is given in the signature it must match the value at the same position of a tuple; if a type is given it can match any value of that type in the position. For example, all of the following statements will remove the first tuple in the Tuple Space of Figure 12.4:

```
in(True, 5.6, 'C', False)
in(B: Boolean, 5.6, 'C', False)
in(True, F: Float, 'C', B2: Boolean)
```

The second in-statement will return the value True in the formal parameter B; the third in-statement will return the values 5.6 in F and False in B2.

The Tuple Space can be used for dispatching computational jobs to processes that may reside on different computers. A tuple ("job", J, C) will indicate that job J should be assigned to computer C. Each computer will block waiting for a job:

```
in("job", J: Jobs, 4);          -- Computer 4 waits for a job
```

The dispatching task will "throw" jobs into the tuple space. The out-statement can also use a formal parameter to indicate that it doesn't care which computer does each job:

```
out("job", 6, C: Computers); -- Job 6 to any computer
```

The advantage of the Linda model is its extreme flexibility. Note that a process may place a tuple in the tuple-space and terminate; only later will another process retrieve the tuple. Thus a Linda program is distributed in *time* as well as being distributed in *space* (among processes that may be on separate CPU's). Contrast this with Ada and occam which require processes to directly communicate with each other. The disadvantage of the Linda model is the overhead involved in maintaining the Tuple Space which requires potentially unbounded global memory. Even though the tuple-space is global, sophisticated algorithms have been developed for distributing it among many processors.

12.10 Exercises

1. Study the following attempt to solve the mutual-exclusion problem under the shared memory model where B1 and B2 are global Boolean variables with initial value False:

Ada

```
task body T1 is
begin
   loop
      B1 := True;
      loop
         exit when not B2;
         B1 := False;
         B1 := True;
      end loop;
      Critical_Section;
      B1 := False;
      Non_Critical_Section;
   end loop;
end T1;

task body T2 is
begin
   loop
      B2 := True;
      loop
         exit when not B1;
         B2 := False;
         B2 := True;
      end loop;
      Critical_Section;
      B2 := False;
      Non_Critical_Section;
   end loop;
end T2;
```

What is the intended meaning of the variables B1 and B2? Is it possible for both tasks to be in their critical sections at any time? Can the program deadlock? Is liveness achieved?

2. Check the semaphore solution to the mutual exclusion problem. Show that in *all* instruction interleavings, at most one task is in the critical section at any time. What about deadlock, liveness and fairness?

3. What happens in the solution to the mutual exclusion problem if the semaphore is given an initial value greater than 1?

4. Try to precisely define fairness. What is the connection between fairness and priority?

5. How would you implement a semaphore?

6. How does a Linda job dispatcher ensure that a specific job should go to a specific computer?

7. Write a Linda program for matrix multiplication. Each inner product is a separate "job"; an initial dispatching process fills the tuple space with "jobs"; worker processes remove "jobs" and return results; a final collecting process removes and outputs results.

8. Translate the Linda program for matrix multiplication into Ada. Solve the problem twice: once with separate tasks for the dispatcher and collector, and once with a single task that does both functions within a single select-statement.

9. Compare Java monitors with the classic monitor construct.[5]

[5]See: M. Ben-Ari, *Principles of Concurrent and Distributed Programming (Second Edition)*, Addison-Wesley, 2006.