1. Introduction to Concurrent Programming

A *concurrent program* contains two or more threads that execute concurrently and work together to perform some task.

When a program is executed, the operating system creates a *process* containing the code and data of the program and manages the process until the program terminates.

Per-process state information:

- the process' state, e.g., ready, running, waiting, or stopped
- the program counter, which contains the address of the next instruction to be executed for this process
- saved CPU register values
- memory management information (page tables and swap files), file descriptors, and outstanding I/O requests

Multiprocessing operating systems enable several programs/processes to execute simultaneously.

A *thread* is a unit of control within a process:

- when a thread runs, it executes a function in the program - the "main thread" executes the "main" function and other threads execute other functions.
- per-thread state information: stack of activation records, copy of CPU registers (including stack pointer, program counter)
- threads in a multithreaded process share the data, code, resources, address space, and per-process state information

The operating system allocates the CPU(s) among the processes/threads in the system:

- the operating systems selects the process and the process selects the thread, or
- the threads are scheduled directly by the operating system.

In general, each ready thread receives a time-slice (called a *quantum*) of the CPU.

- If a thread waits for something, it relinquishes the CPU.
- When a running thread's quantum has completed, the thread is *preempted* to allow another ready thread to run.

Switching the CPU from one process or thread to another is known as a *context switch*.

multiple CPUs $\Rightarrow$ multiple threads can execute at the same time.

single CPU $\Rightarrow$ threads take turns running

The *scheduling policy* may also consider a thread's priority. We assume that the scheduling policy is *fair*, which means that every ready thread eventually gets to execute.

1.2 Advantages of Multithreading

- Multithreading allows a process to overlap IO and computation.
- Multithreading can make a program more responsive (e.g., a GUI thread can respond to button clicks while another thread performs some task in the background).
- Multithreading can speedup performance through parallelism.
- Multithreading has some advantages over multiple processes:
  - threads require less overhead to manage than processes
  - intra-process thread communication is less expensive than inter-process communication (IPC).

Multi-process concurrent programs do have one advantage: each process can execute on a different machine. This type of concurrent program is called a *distributed program*.

Examples of distributed programs: file servers (e.g., NFS), file transfer clients and servers (e.g., FTP), remote login clients and servers (e.g., Telnet), groupware programs, and web browsers and servers.

The main disadvantage of concurrent programs is that they are difficult to develop. Concurrent programs often contain bugs that are notoriously hard to find and fix.

1.3 Threads in Java

(See Chapter1JavaThreads.pdf)

1.4 Threads in Win32

(See Chapter1Win32Threads.pdf)

1.5 Pthreads

(See Chapter1PthreadsThreads.pdf)

1.6 A C++ *Thread* Class

Details about Win32 and POSIX threads can be encapsulated in a C++ *Thread* class.

1.6.1 C++ Class *Thread* for Win32

(See Chapter1Win32ThreadClass.pdf)

1.6.2 C++ Class *Thread* for Pthreads

(See Chapter1PthreadThreadClass.pdf)

1.7 Thread Communication

In order for threads to work together, they must communicate.

One way for threads to communicate is by accessing shared memory. Threads in the same program can reference global variables or call methods on a shared object.

Listing 1.8 shows a C++ program in which the *main* thread creates two *communicatingThreads*. Each *communicatingThread* increments the global shared variable *s* ten million times. The *main* thread uses *join()* to wait for the *communicatingThreads* to complete, then it displays the final value of *s*.

We executed this program fifty times. In forty-nine of the executions, the value 20000000 was displayed, but the value displayed for one of the executions was 19215861.

Concurrent programs exhibit non-deterministic behavior - two executions of the *same* program with the *same* input can produce *different* results.

```
int s=0; // shared variable s

class communicatingThread: public Thread {
public:
   communicatingThread (int ID) : myID(ID) {}
   virtual void* run();
private:
   int myID;
};
void* communicatingThread::run() {
   std::cout << "Thread " << myID << " is running" << std::endl;
   for (int i=0; i<10000000; i++) // increment s ten million times
      s = s + 1;
   return 0;
}

int main() {
   std::auto_ptr<communicatingThread> thread1(new communicatingThread (1));
   std::auto_ptr<communicatingThread> thread2(new communicatingThread (2));
   thread1->start();  thread2->start();
   thread1->join();   thread2->join();
   std::cout << "s: " << s << std::endl; // expected final value of s is 20000000
   return 0;
}
```

Listing 1.8 Shared variable communication.

1.7.1 Non-Deterministic Execution Behavior

Example 1. Assume that integer *x* is initially 0.

```
Thread1   Thread2   Thread3
(1) x = 1;(2) x = 2; (3)   y = x;
```

The final value of *y* is unpredictable, but it is expected to be either 0, 1 or 2. Below are some of the possible interleavings of these three statements.

(3), (1), (2) ⇒ final value of *y* is 0
(2), (1), (3) ⇒ final value of *y* is 1
(1), (2), (3) ⇒ final value of *y* is 2

The memory hardware guarantees that that read and write operations on integer variables do not overlap. (Below, such operations are called "atomic" operations.)

Example 2**.** Assume that *x* is initially 2.

```
Thread1            Thread2            Thread3
while (true) {     while (true) {     while (true) {
  (1) x = 1;         (2) x = 2;         (3) if (x == 2) exit(0);
 }                  }                  }
```

Thread3 will never terminate if statements (1), (2), and (3) are interleaved as follows: (2), (1), (3), (2), (1), (3), (2), (1), (3), …. This interleaving is probably not likely to happen, but if it did, it would not be completely unexpected.

In general, non-deterministic execution behavior is caused by one or more of the following:

- the unpredictable rate of progress of threads executing on a single processor (due to context switches between the threads)
- the unpredictable rate of progress of threads executing on different processors (due to differences in processor speeds)
- the use of non-deterministic programming constructs, which make unpredictable selections between two or more possible actions.

Non-deterministic results do not necessarily indicate the presence of an error. Threads are frequently used to model real-world objects and the real-world is non-deterministic.

Non-determinism adds flexibility to a design. A bounded buffer compensates for differences in the rates at which the producers and consumers work. The order of deposits and withdraws is non-deterministic.

Non-determinism and concurrency are related concepts. Concurrent events A and B can be modeled as a non-deterministic choice between two interleavings of events: (A followed by B) or (B followed by A). But the possible number of interleavings explodes as the number of concurrent events increases.

Non-determinism is an inherent property of concurrent programs. The burden of dealing with non-determinism falls on the programmer, who must ensure that threads are correctly synchronized without imposing unnecessary constraints that only reduce the level of concurrency.

Non-deterministic executions create major problems during testing and debugging.

1.7.2 Atomic Actions

A program's state contains a value for each variable defined in the program and other implicit variables, like the program counter.

An atomic action transforms the state of the program, and the state transformation is indivisible: {x==0} x=1; {x==1}.

A state transformation performed during an atomic action is indivisible if other threads can see the program's state as it appears before the action or after the action, but not some intermediate state while the action is occurring.

A context switch may occur while one thread is performing an atomic action as long as we don't allow the other threads to see or interfere with the action while it is in progress.

Individual machine instructions such as *load*, *add*, *subtract* and *store* are typically executed atomically; this is guaranteed by the memory hardware.

In Java, an assignment of 32 bits or less is guaranteed to be implemented atomically, so an assignment statement such as *x = 1* for a variable *x* of type *int* is an atomic action. In general, however, the execution of an assignment statement may not be atomic.

1.7.2.1 Non-atomic arithmetic expressions and assignment statements. An interleaving of the machine instructions from two or more expressions or assignment statements may produce unexpected results.

Example 3. Assume that *y* and *z* are initially 0.

```
Thread1       Thread2
x = y + z;    y = 1;
              z = 2;
```

Assume (incorrectly) that each statement is an atomic action, what are the expected final values?

The machine instructions for *Thread1* and *Thread2*:

```
Thread1          Thread2
(1) load r1, y   (4) assign y, 1
(2) add          r1, z (5) assign z, 2
(3) store r1, x
```

Possible interleavings:

(1), (2), (3), (4), (5) $\Rightarrow$ x is 0
(4), (1), (2), (3), (5) $\Rightarrow$ x is 1
(1), (4), (5), (2), (3) $\Rightarrow$ x is 2 *
(4), (5), (1), (2), (3) $\Rightarrow$ x is 3

Example 4. Assume that the initial value of *x* is 0.

```
Thread1        Thread2
x = x + 1;     x = 2;
```

The machine instructions for *Thread1* and *Thread2* are:

```
Thread1          Thread2
(1) load r1, x   (4) assign x, 2
(2) add r1, 1
(3) store r1, x
```

Possible interleavings:

(1), (2), (3), (4) => x is 2
(4), (1), (2), (3) => x is 3
(1), (2), (4), (3) => x is 1 *

If there are *n* threads (*Thread*$_1$, *Thread*$_2$, …, *Thread*$_n$) such that *Thread*$_i$ executes *mi* atomic actions, then the number of possible interleavings of the atomic actions is:

(m1 + m2 + … + mn)! / (m1! * m2! * … * mn!)

Andrews defined a condition called At-Most-Once under which expression evaluations and assignments will appear to be atomic. A critical reference in an expression is a reference to a variable that is changed by another thread.

At-Most-Once:

- An assignment statement *x* = *e* satisfies the At-Most-Once property if either:

(1) *e* contains at most one critical reference and *x* is neither read nor written by another thread, or

(2) *e* contains no critical references, in which case *x* may be read or written by other threads.

- An expression that is not in an assignment satisfies At-Most-Once if it contains no more than one critical reference.

"There can be at most one shared variable and the shared variable can be referenced at most one time. "

Assignment statements that satisfy At-Most-Once *appear* to execute atomically even though they are not atomic.

Example 5. Assume that *x* and *y* are initially 0.

```
Thread1         Thread2
x = y + 1;      y = 1;
```

- The expression in *Thread1's* assignment statement references *y* (one critical reference) but *x* is not referenced by *Thread2*,
- The expression in *Thread2's* assignment statement has no critical references.

Both statements satisfy At-Most-Once.

1.7.2.2 Non-atomic groups of statements. Another type of undesirable non-determinism in a concurrent program is caused by interleaving groups of statements, even though each statement may be atomic.

Example 5. Variable *first* points to the first *Node* in the list. Assume the list is !empty

```
class Node {
public:
   valueType value;
   Node* next;
}
Node* first;                // first points to the first Node in the list;
void deposit(valueType value) {
   Node* p = new Node;  // (1)
   p->value = value;        // (2)
   p->next = first;         // (3)
   first = p;               // (4) insert the new Node at the front of the list
}
valueType withdraw() {
   valueType value = first->value;   // (5) withdraw the first value in the list
   first = first->next;     // (6) remove the first Node from the list
   return value;            // (7) return the withdrawn value
}
```

The following interleaving of statements is possible:

```
   valueType value = first->value;   // (5) in withdraw
   Node* p = new Node();   // (1) in deposit
   p->value = value             // (2) in deposit
   p->next = first;             // (3) in deposit
   first = p;                   // (4) in deposit
   first = first->next;         // (6) in withdraw
   return value;                // (7) in withdraw
```

At the end of this sequence:

- withdrawn item is still pointed to by *first*
- deposited item has been lost.

To fix this problem, each of methods *deposit* and *withdraw* must be implemented as an atomic action.

1.8 Testing and Debugging Multithreaded Programs

The purpose of testing is to find program failures.

A *failure* is an observed departure of the external result of software operation from software requirements or user expectations [IEEE90]. Failures can be caused by hardware or software faults or by user errors.

A software *fault* (or defect, or bug) is a defective, missing, or extra instruction or a set of related instructions, that is the cause of one or more actual or potential failures [IEEE 88].

Debugging is the process of locating and correcting faults.

The conventional approach to testing and debugging a sequential program:

(1) Select a set of test inputs

(2) Execute the program once with each input and compare the test results with the intended results.

(3) If a test input finds a failure, execute the program *again* with the same input in order to collect debugging information and find the fault that caused the failure.

(4) After the fault has been located and corrected, execute the program *again* with each of the test inputs to verify that the fault has been corrected and that, in doing so, no new faults have been introduced (a.k.a "regression testing").

This process breaks down when it is applied to concurrent programs.

1.8.1 Problems and Issues

Let CP be a concurrent program. Multiple executions of CP with the *same* input may produce *different* results. This non-deterministic execution behavior creates the following problems during the testing and debugging cycle of CP:

Problem 1. When testing CP with input X, a single execution is insufficient to determine the correctness of CP with X. Even if CP with input X has been executed successfully many times, it is possible that a future execution of CP with X will produce an incorrect result.

Problem 2. When debugging a failed execution of CP with input X, there is no guarantee that this execution will be repeated by executing CP with X.

Problem 3. After CP has been modified to correct a fault detected during a failed execution of CP with input X, one or more successful executions of CP with X during regression testing do not imply that the detected fault has been corrected or that no new faults have been introduced.

There are many issues that must be dealt with in order to solve these problems:

Program Replay: Programmers rely on debugging techniques that assume program failures can be reproduced. Repeating an execution of a concurrent program is called "program replay".

Program Tracing: Before an execution can be replayed it must be traced. But what exactly does it mean to replay an execution?

- If a C++ program is executed twice and the inputs and outputs are the same for both executions, are these executions *identical*? Are the context switches always on the same places? Does this matter?
- Now consider a concurrent program. Are the context switches among the threads in a program important? Must we somehow trace the points at which the context switches occur and then repeat these switch points during replay?
  - What should be replayed?
  - Consider the time and space overhead for capturing and storing the trace.
  - In a distributed system, there is an "observability problem": it is difficult to accurately observe the order in which actions on different computers occur during an execution.

Sequence Feasibility: A sequence of actions that is allowed by a program is said to be a *feasible* sequence. Testing involves determining whether or not a given sequence is feasible or infeasible. "Good" sequences are expected to be feasible while "bad" sequences are expected to be infeasible. How are sequences selected?

- Allow sequences to be exercised non-deterministically (*non-deterministic testing*). Can non-deterministic testing be used to show that bad sequences are infeasible?
- Force selected sequences to be exercised (*deterministic testing).* Choosing sequences that are effective for detecting faults is hard to do. A *test coverage criterion* can be used to guide the selection of tests and to determine when to stop testing.

Sequence Validity: Sequences allowed by the specification, i.e. "good" sequences, are called *valid* sequences; other sequences are called *invalid* sequences. A goal of testing is to find valid sequences that are infeasible and invalid sequences that are feasible.

The Probe Effect: Modifying a concurrent program to capture a trace of its execution may interfere with the normal execution of the program.

- The program will behave differently after the trace routines have been added. (But without this interference, it is possible that some failures cannot be observed.)
- Some failures that would have been observed without adding the trace routines will no longer be observed.

One way to address the probe effect is to make sure that every feasible sequence is exercised at least once during testing. Reachability testing can exercise all of the feasible sequences of a program, when the number of sequences is not “too large”.

Three different problems:

- Observability problem: the difficulty of accurately tracing a given execution,
- Probe effect: the ability to perform a given execution at all.
- Replay problem: repeating an execution that has already been observed.

Real-Time: The probe effect is a major issue for real-time concurrent programs. The correctness of a real-time program depends not only on its logical behavior, but also on the time at which its results are produced.

For an example of program tracing and replay, we’ve modified the C++ program in Listing 1.8 so that it can trace and replay its own executions. The new program is shown in Listing 1.9.

1.8.2 Class *TDThread* for Testing and Debugging

Threads in Listing 1.9 are created by inheriting from C++ class *TDThread* instead of class *Thread*. Class *TDThread* provides the same interface as our C++ *Thread* class, plus several additional functions that are used internally during tracing and replay.

```
sharedVariable<int> s(0); // shared variable s
   class communicatingThread: public TDThread {
public:
   communicatingThread(int ID) : myID(ID) {}
   virtual void* run();
private:
   int myID;
};
void* communicatingThread::run() {
   std::cout << "Thread " << myID << " is running" << std::endl;
   for (int i=0; i<2; i++)      // increment s two times (not 10 million times)
      s = s + 1;
   return 0;
}
int main() {
   std::auto_ptr<communicatingThread> thread1(new communicatingThread (1));
   std::auto_ptr<communicatingThread> thread2(new communicatingThread (2));
   thread1->start();  thread2->start();
   thread1->join();   thread2->join();
   std::cout << "s: " << s << std::endl;    // the expected final value of s is 4
   return 0;
}
```
Listing 1.9 Using classes *TDThread* and *sharedVariable<>*.

The main purpose of class *TDThread* is to automatically and unobtrusively generate an integer identifier (ID) and a name for each thread.

- Thread IDs are based on the order in which threads are constructed during execution. The first thread constructed is assigned ID 1, the second thread is assigned ID 2, and so on.
- Thread names are generated automatically, e.g., main_thread1, main_thread2, main_thread1_thread1, main_thread1_thread2. A more descriptive name can be provided by manually supplying a name to *TDThread's* constructor. For example:

```
class namedThread : public TDThread {
   namedThread() : TDThread("namedThread") {} // "namedThread" is the name
   …
}
```

User-supplied thread names must be unique, or they will be rejected.

1.8.3 Tracing and Replaying Executions with Class Template *sharedVariable*<>

Class template *sharedVariable*<> provides functions that allow every shared variable access to be traced and replayed. Class *sharedVariable*<> provides the usual operators (+, -, =, <, etc) for primitive types. It also traces and replays the read and write operations in the implementations of these operators.

- The IDs generated by class *TDThread* for threads *thread1* and *thread2* are 1 and 2
- In trace mode, class *sharedVariable<int>* records the order in which these two threads read and write shared variable *s*.
- Each increment "s = s+1" involves a read of *s* followed by a write of *s*, and each thread increments *s* twice.
- The final value of *s* is expected to be 4 under the incorrect assumption that the increments are atomic.

A possible trace for Listing 1.9 is

```
Read(thread1,s)     // thread1 reads s; s is 0
Read(thread2,s)     // thread2 reads s; s is 0
Write(thread2,s)    // thread2 writes s; s is now 1
Write(thread1,s)    // thread1 writes s; s is now 1
Read(thread1,s)     // thread1 reads s; s is 1
Write(thread1,s)    // thread1 writes s; s is now 2
Read(thread2,s)     // thread2 reads s; s is 2
Write(thread2,s)    // thread2 writes s; s is now 3
```

An execution that produces this trace will display the output 3, not 4.
In replay mode, *sharedVariable<int>* forces the threads to execute their read and write operations on *s* in the order recorded in the trace, always producing the output 3.

1.9 Thread Synchronization

The examples in this chapter showed the failures that can occur when accesses to shared variables are not properly synchronized.

One type of synchronization is called *mutual exclusion*.

Mutual exclusion ensures that a group of atomic actions, called a critical section, cannot be executed by more than one thread at a time. That is, a critical section must be executed as an atomic action.

The increment statements executed by thread1 and thread2 in Listing 1.9 require mutual exclusion to work properly.

Failure to correctly implement critical sections is a fault called a *data race* [Netzer 1992].

Another type of synchronization is called *condition synchronization*.

Condition synchronization ensures that the state of a program satisfies a particular condition before some action occurs.

For the linked list in Example 5 earlier, there is a need for both condition synchronization and mutual exclusion.

- The list must not be in an empty condition when method *withdraw* is allowed to remove an item,
- mutual exclusion is required for ensuring that deposited items are not lost and are not withdrawn more than twice.

2. The Critical Section Problem

A code segment that accesses shared variables (or other shared resources) and that has to be executed as an atomic action is referred to as a critical section.

```
while (true) {
    entry-section
    critical section // contains accesses to shared variables or other resources.
    exit-section
    non-critical section // a thread may terminate its execution in this section.
}
```

The entry- and exit-sections that surround a critical section must satisfy the following correctness requirements:

- *mutual exclusion*: When a thread is executing in its critical section, no other threads can be executing in their critical sections.

- *progress*: If no thread is executing in its critical section and there are threads that wish to enter their critical sections, then only the threads that are executing in their entry- or exit-sections can participate in the decision about which thread will enter its critical section next, and this decision cannot be postponed indefinitely.

- *bounded waiting*: After a thread makes a request to enter its critical section, there is a bound on the number of times that other threads are allowed to enter their critical sections before this thread's request is granted.

One solution:

```
disableInterrupts();   // disable all interrupts
critical section
enableInterrupts();    // enable all interrupts
```

This disables interrupts from the interval timer that is used for time-slicing, and prevents any other thread from running until interrupts are enabled again.

However:

- commands for disabling and enabling interrupts are not always available to user code.
- a thread on one processor may not be able to disable interrupts on another processor.

2.1 Software Solutions to the Two-Thread Critical Section Problem

Threads T0 and T1 are attempting to enter their critical sections.

All assignment statements and expressions involving shared variables in the entry and exit sections are atomic operations.

There are no groups of statements that must be executed atomically.

Thus, the entry- and exit-sections themselves need not be critical sections.

The solutions below assume that the hardware provides mutual exclusion for individual read and write operations on shared variables.

2.1.1 Incorrect Solution 1

Threads T0 and T1 use variables *intendToEnter0* and *intendToEnter1* to indicate their intention to enter their critical section. A thread will not enter its critical section if the other thread has already signaled its intention to enter.

```
boolean intendToEnter0=false, intendToEnter1=false;
T0                                      T1
while (true) {                          while (true) {
   while (intendToEnter1) { ; }(1)         while (intendToEnter0) { ; }   (1)
   intendToEnter0 = true;       (2)        intendToEnter1 = true;         (2)
   critical section             (3)        critical section               (3)
   intendToEnter0 = false;      (4)        intendToEnter1 = false;        (4)
   non-critical section         (5)        non-critical section           (5)
}                                       }
```

This solution does not guarantee mutual exclusion.

The following execution sequence shows a possible interleaving of the statements in T0 and T1 that ends with both T0 and T1 in their critical sections.

```
T0                  T1     Comments
(1)                        T0 exits its while-loop
   context switch →
                    (1)    T1 exits its while-loop
                    (2)    intendToEnter1 is set to true
                    (3)    T1 enters its critical section
   ← context switch
(2)                        intendToEnter0 is set to true
(3)                        T0 enters its critical section
```

2.1.2 Incorrect Solution 2

Global variable *turn* is used to indicate which thread is allowed to enter its critical section, i.e., the threads take turns entering their critical sections. The initial value of *turn* can be 0 or 1.

```
int turn = 1;

T0                                  T1
while (true) {                      while (true) {
   while (turn != 0) { ; }   (1)        while (turn != 1) { ; }   (1)
   critical section          (2)        critical section          (2)
   turn = 1;                 (3)        turn = 0;                 (3)
   non-critical section      (4)        non-critical section      (4)
}                                   }
```

T0 and *T1* are forced to alternate their entries into the critical section, ensuroing mutual exclusion and bounded waiting. However, the progress requirement is violated.

Assume that the initial value of *turn* is 1:

T0	T1	Comments
	(1)	*T1* exits its while-loop
	(2)	*T1* enters and exits its critical section
	(3)	*turn* is set to 0
	(4)	*T1* terminates in its non-critical section
← *context switch*		
(1)		*T0* exits its while-loop \
(2)		*T0* enters and exits its critical section
(3)		*turn* is set to 1
(4)		*T0* executes its non-critical section
(1)		*T0* repeats (1) forever

Thread *T0* cannot exit the loop in (1) since the value of *turn* is 1 and *turn* will never be changed by T1.

2.1.3 Incorrect Solution 3

This solution is a more “polite” version of Solution 1.

When one thread finds that the other thread also intends to enter its critical section, it sets its own *intendToEnter* flag to *false* and waits for the other thread to exit its critical section.

```
T0                                         T1
while (true) {                             while (true) {
   intendToEnter0 = true;          (1)         intendToEnter1 = true;              (1)
   while (intendToEnter1) {        (2)         while (intendToEnter0) {            (2)
     intendToEnter0 = false;       (3)             intendToEnter1 = false;         (3)
     while(intendToEnter1) {;}     (4)             while(intendToEnter0) {;}       (4)
     intendToEnter0 = true;        (5)             intendToEnter1 = true;          (5)
   }                                           }
   critical section                (6)         critical section                    (6)
   intendToEnter0 = false;         (7)         intendToEnter1 = false;             (7)
   non-critical section            (8)         non-critical section                (8)
}                                          }
```

This solution ensures that when both *intendToEnter0* and *intendToEnter1* are true, only one of *T0* and *T1* is allowed to enter its critical section.

Thus, the mutual exclusion requirement is satisfied.

There is a problem with this solution, as illustrated by the following execution sequence:

T0	T1	Comments
(1)		*intendToEnter0* is set to true
(2)		*T0* exits the first while-loop in the entry section
(6)		*T0* enters its critical section; *intendToEnter0* is true
context switch →		
	(1)	*intendToEnter1* is set to true
	(2)-(3)	*intendToEnter1* is set to false
	(4)	*T1* enters the second while-loop in the entry section
←context switch		
(7)		*intendToEnter0* is set to false
(8)		*T0* executes its non-critical section
(1)		*intendToEnter0* is set to true
(2)		*T0* exits the first while-loop in the entry section
(6)		*T0* enters its critical section; *intendToEnter0* is true
context switch →		
	(4)	*T1* is still waiting for *intendToEnter0* to be false
← *context switch*		
(7)		
… repeat infinitely		

In this execution sequence, *T0* enters its critical section infinitely often and *T1* waits forever to enter its critical section.

Thus, this solution does not guarantee bounded waiting.

2.1.4 Peterson's Algorithm

Peterson’s algorithm is a combination of solutions (2) and (3). If both threads intend to enter their critical sections, then *turn* is used to break the tie.

```
boolean intendToEnter0 = false, intendToEnter1 = false;
int turn; // no initial value for turn is needed.

T0                                          T1
while (true) {                              while (true) {
   intendToEnter0 = true;         (1)          intendToEnter1 = true;         (1)
   turn = 1;                      (2)          turn = 0;                      (2)
   while (intendToEnter1 &&       (3)          while (intendToEnter0 &&       (3)
       turn == 1) { ; }                            turn == 0) { ; }
   critical section               (4)          critical section               (4)
   intendToEnter0 = false;        (5)          intendToEnter1 = false;        (5)
   non-critical section           (6)          non-critical section           (6)
}                                            }
```

Peterson’s algorithm is short but tricky:

- If threads T0 and T1 both try to enter their critical sections, which thread wins the race, i.e., gets to enter? (Hint: Suppose T0 executes (2) before T1 executes (2).)

- Suppose the winning thread exits its critical section and immediately tries to reenter; will this thread succeed in entering again or will the other thread enter? (Hint: Whose turn will it be?)

- If only one of T0 or T1 wants to enter their critical section, does it matter whose turn it is? (Hint: Operator && is a short-circuit operator.)

2.1.5 Using the *volatile* modifier

To optimize speed, the compiler may allow each thread in Peterson's algorithm to keep private copies of shared variables *intendToEnter0, intendToEnter1*, and *turn*. If this optimization is performed, updates made to these variables by one thread will be made to that thread's private copies and thus will not be visible to the other thread.

This potential inconsistency causes an obvious problem in Peterson's algorithm.

In Java, the solution to this problem is to declare shared variables (that are not also declared as double or long) as *volatile*:

```
volatile boolean intendToEnter0 = false; volatile boolean intendToEnter1 = false;
volatile int turn;
```

Declaring the variables as *volatile* ensures that consistent memory values will be read by the threads.

C++ also allows variables to be declared as *volatile*. However, C++ and Java have different memory models, so even if C++ variables are *volatile*, some hardware optimizations may cause Peterson's algorithm to fail in C++ (see Section 2.6).

Shared variables do not always have to be *volatile* variables. We will see that the programming language or thread library may handle compiler and hardware optimization issues for us.

2.2 Ticket-Based Solutions to the n-Thread Critical Section Problem

In the n-thread critical section problem, there are *n* threads instead of just two. When a thread wishes to enter a critical section, it requests a ticket. Threads enter their critical sections in ascending order of their ticket numbers.

2.2.1 Ticket Algorithm

```
volatile int number[n];       // number[i], initially 0, is the ticket number for thread Ti
```

Each thread T_i, $0 \le i \le n-1$ executes the following code:

```
volatile long next = 1;      // next ticket number to be issued to a thread
volatile long permit =1;    // ticket number permitted to enter critical section
while (true) {
   number[i] = InterlockedExchangeAdd(&next,1); (1)
   while (number[i] != permit) { ; }     (2)
   critical section                      (3)
   ++permit;                             (4)
   non-critical section                  (5)
}
```

InterlockedExchangeAdd() is part of the Win32 API. A call such as the following:

```
oldValueOfX = InterlockedExchangeAdd(&x, increment);
```

atomically adds the value of *increment* to *x* and returns the old value of *x* (i.e., before adding *increment* to *x*).

```
number[i] = InterlockedExchangeAdd(&next, 1)
```

is equivalent to the following critical section:

```
number[i] = next;     // these statements are executed
next = next + 1;      //     as a critical section
```

Function *InterlockedExchangeAdd()* can be implemented with special hardware instructions. Note that the values of *permit* and *next* grow without bounds.

2.2.2 Bakery Algorithm

The Bakery algorithm does not require any special hardware instructions.

Each thread T_i, $0 \le i \le n-1$, gets a ticket with a pair of values (number[i], i) on it. The value *number*[i] is the ticket number, and *i* is the ID of the thread. For two tickets (a,b) and (c,d) define:

Ticket (a,b) < Ticket (c,d) if a < c or (a == c and b < d).

An *incorrect* version: each thread T_i, $0 \le i \le n-1$, executes:

```
while (true) {
   number[i] = max(number) + 1;             (1)    // non-atomic
   for (int j=0; j<n; j++ )                 (2)
   while (j != i && number[j] != 0 &&       (3)    // array number[] is initially all zeros
       (number[j],j) < (number[i],i) ) { ; } (4)
   critical section                         (5)
   number[i] = 0;                           (6)
   non-critical section                     (7)
}
```

The call to max(*number*) returns the maximum value in array *number*. Since different threads may execute (1) at the same time, thread T_i may obtain the same ticket number as another thread (in which case thread IDs are used to break ties).

This algorithm does not satisfy the mutual exclusion requirement:

T0	T1	Comments
(1)		T_0 evaluates max(*number*) + 1, which is 1, but a context switch occurs *before* assigning 1 to number[0]
context switch →		
	(1)	T_1 sets *number*[1] to max(*number*) + 1, which is 1
	(2)	T_1 starts its for-loop
	(3)	T_1 exits its while and for-loops
	(4)	T_1 enters its critical section
← *context switch*		
(1)		T_0 assigns 1 (*not 2*) to *number*[0].
(2)		T_0 starts its for-loop
(3)		T_0 exits its while- and for-loops since *number*[0] and *number*[1] are 1, and (*number*[0],1) < (*number*[1],2)
(4)		T_0 enters its critical section while T_1 is in its critical section

Thus, once thread T_i, i>0, starts executing statement (1), *number*[i] should not be accessed by other threads during their execution of statement (3) until T_i finishes executing statement (1).

The complete bakery algorithm is given below. It uses the following global array:

```
volatile bool choosing[n];    // initially all elements of choosing are false.
```

If *choosing[i]* is true, then thread T_i is choosing its ticket number at statement (2).

```
while (true) {
   choosing[i] = true;                                                          (1)
   number[i] = max(number)+1;                                                   (2)
   choosing[i] = false;                                                         (3)
   for (int j=0; j<n; j++) {                                                    (4)
        while (choosing[j]) { ; }                                               (5)
        while ( j != i && number[j] !=0 && (number[j],j) < (number[i],i) ) { ; }   (6)
   }
   critical section                                                             (7)
   number[i] = 0;                                                               (8)
   non-critical section                                                         (9)
}
```

The Bakery algorithm satisfies the mutual exclusion, progress, and bounded waiting requirements. However, the values in *number* grow without bound. (How long will it take these numbers to overflow?)

Lamport showed that the Bakery algorithm can be made to work even when read and write operations are not atomic, i.e., when the read and write operations on a variable may overlap.

2.3 Hardware Solutions to the n-Thread Critical Section Problem

The win32 *InterlockedExchange* function atomically exchanges a pair of 32-bit values and behaves like the following atomic function:

```
long InterLockedExchange(long* target, long newValue) { // executed atomically
  long temp = *target; *target = newValue; return temp;
}
```

2.3.1 A Partial Solution

This solution uses *InterlockedExchange* to guarantee mutual exclusion and progress, but not bounded waiting.

```
volatile long lock = 0;

Each thread executes:
   while (true) {
      while (InterlockedExchange(const_cast<long*>(&lock), 1) == 1) { ; }   (1)
      critical section                                                       (2)
      lock = 0;                                                              (3)
      non-critical section                                                   (4)
}
```

- If the value of *lock* is 0 when *InterlockedExchange* is called, *lock* is set to 1 and *InterlockedExchange* returns 0 allowing the calling thread to enter its critical section.

- While this thread is in its critical section, calls to *InterlockedExchange* will return the value 1, keeping the calling threads delayed in their while-loops.

If critical sections are small and contention among threads for the critical sections is low, then unbounded waiting is not a big problem.

2.3.2 A Complete Solution

This solution to the n-process critical section satisfies all three correctness requirements.

```
volatile bool waiting[n];  // waiting[i] is true when Ti is waiting to enter
```

Each thread T_i executes the following code:

```
volatile bool waiting[n];  // initialized to false
volatile long lock, key;// initialized to 0
while (true) {
  waiting[i] = true;                                              (1)
  key = 1;                                                        (2)
  while (waiting[i] && key) {                                     (3)
    key = InterlockedExchange(const_cast<long*>(&lock), 1);       (4)
  }                                                               (5)
  waiting[i] = false;                                             (6)
  critical section                                                (7)
  j = (i+1) % n;                                                  (8)
  while ((j != i) && !waiting[j]) { j = (j+1) % n; }              (9)
  if (j == i)                                                     (10)
    lock = 0;                                                     (11)
  else                                                            (12)
    waiting[j] = false;                                           (13)
  non-critical section                                            (14)
}
```

Thread T_i stays in the while-loop in (3) until either *InterlockedExchange* returns 0 or *waiting*[i] is set to 0 by another thread when that thread exits its critical section.

When thread T_i exits its critical section it uses the while-loop in statement (9) to search for a waiting thread.

- If the while-loop terminates with *j* == *i*, then no waiting threads exist;
- otherwise, thread T_j is a waiting thread and *waiting*[j] is set to 0 to let T_j exit the while-loop at statement (3) and enter its critical section.

2.3.3 A Note on Busy-Waiting

Busy-waiting: a waiting thread executes a loop that maintains its hold of the CPU.

Busy-waiting wastes CPU cycles. To reduce the amount of busy-waiting, some type of *sleep* instruction can be used.

```
 while (waiting[i] && key) {
    Sleep(100); // release the CPU
    key = InterlockedExchange(const_cast<long*>(&lock), 1);
 }
```

Win32: Sleep(time); *time* in millisec.

Java: try {Thread.sleep(time);} catch(InterruptedException e) {}, *time* in millisec.

Pthreads: sleep(time), *time* in seconds.

Executing a sleep statement results in a context switch that allows another thread to execute.

If contention among threads for the critical section is low and critical sections are small, then there may be little chance that a thread will execute the while-loop for more than a few iterations.

In such cases, it may be more efficient to use busy-waiting and avoid the time-consuming context switch caused by executing a sleep statement.

A potential performance problem exits on multiprocessor systems with private caches and cache-coherence protocols that allow shared, writeable data to exist in multiple caches.

Suppose two processors are busy-waiting on the value of *lock* in the complete solution above.

- When a waiting processor modifies *lock* in statement (4) it causes the modified *lock* to be invalidated in the other processor's cache.
- As a result, the value of *lock* can bounce repeatedly from one cache to the other

The solution is to busy-wait on a read of *lock*:

```
while (waiting[i] && key) {                                               (3)
   while (lock) { ; } // wait for lock to be released                     (4)
   key = InterlockedExchange(const_cast<long*>(&lock), 1); // try to grab lock  (5)
}
```

When a processor wants to acquire the lock, it spins locally in its cache without modifying and invalidating the *lock* variable.

When the lock is eventually released, function *InterlockedExchange* is used to *attempt* to change the value of *lock* from 0 to 1.

If another thread "steals" the lock between statements (4) and (5), then looping will continue at statement (3).

In general, the performance of busy-waiting algorithms depends on the number of processors, the size of the critical section, and the architecture of the system.

2.4 Deadlock, Livelock and Starvation

One general correctness requirement of concurrent programs is the absence of deadlock, livelock and starvation.

2.4.1 Deadlock

A deadlock requires one or more threads to be blocked *forever*.

Sleep blocks a thread *temporarily*, but the thread is eventually allowed to run again.

A thread that executes a *receive* statement to receive a message from another thread will block until the message arrives, but it is possible that a message will never arrive.

Let CP be a concurrent program containing two or more threads. Assume there is an execution of CP that exercises an execution sequence S, and at the end of S, there exists a thread T that satisfies these conditions:

- T is blocked due to the execution of a synchronization statement (e.g., waiting to receive a message)
- T will remain blocked forever, regardless of what the other threads will do

CP is said to have a deadlock. A global deadlock refers to a deadlock in which all non-terminated threads are deadlocked.

Example:

```
Port p,q;
         T1                        T2
msg = p.receive();        msg = q.receive();    // receive blocks until a message arrives
q.send(msg);              p.send(msg);
```

2.4.2 Livelock

We assume that some statements in CP are labeled as "progress statements”, indicating that threads are expected to eventually execute these statements (e.g., last statement of a thread, first statement of a critical section, a statement immediately following a loop.)

Assume there is an execution of CP that exercises an execution sequence S, and at the end of S there exists a thread T that satisfies the following conditions, regardless of what the other threads will do:

- T will not terminate or deadlock
- T will never make progress

CP is said to have a livelock. Livelock is the busy-waiting analog of deadlock. A livelocked thread is running, not blocked, but it will never make any progress.

Incorrect solution 2 in Section 2.1.2 has a livelock:

- T0 executes (1), (2), (3), and (4). Now *turn* is 1.
- T1 executes (1), (2), and (3) and terminates in its non-critical section. Now *turn* is 0.
- T0 executes (1), (2), (3), and (4), making *turn* 1, and executes its while-loop at (1)

At the end this sequence, *T0* is stuck in a busy-waiting loop at (1) waiting for *turn* to become 0. T0 will never enter its critical section, i.e., make any progress. Thus, *T0* is livelocked.

2.4.3 Starvation

Assume that CP contains an infinite execution sequence S satisfying the following three properties:

(a) S ends with an infinite repetition of a fair cycle of statements. (A cycle of statements in CP is said to be fair if each non-terminated thread in CP is either always blocked in the cycle or is executed at least once. Non-fair cycles are not considered here, since such cycles cannot repeat forever when fair scheduling is used.)

(b) There exists a non-terminated thread T that does not make progress in the cycle.

(c) Thread T is neither deadlocked nor livelocked in the cycle. In other words, when CP reaches the cycle of statements that ends S, CP may instead execute a different sequence S' such that T makes progress or terminates in S'.

CP is said to have a *starvation*.

When CP reaches the cycle of statements at the end of S, whether thread T will starve depends on how the threads in CP are scheduled. Note that fair scheduling of the threads in CP does not guarantee a starvation-free execution of CP.

Incorrect solution 3 in section 2.1.3 has a possible starvation.

(a) *T0* executes (1), (2), and (6). Now *T0* is in its critical section, *intendToEnter*[0] is *true*.

(b) *T1* executes (1)-(4). Now *intendToEnter* [1] is *false* and *T1* is waiting for *intendToEnter* [0] to be *false*.

(c) *T0* executes (7), (8), (1), (2) and (6). Now *T0* is in its critical section and *intendToEnter* [0] is *true*.

(d) *T1* resumes execution at (4) and is still waiting for *intendToEnter* [0] to be *false*.

(e) *T0* executes (7), (8), (1), (2) and (6).

(f) *T1* resumes execution at (4).

… Infinite repetition of steps (e) and (f) ...

In this sequence, *T1* never makes progress in the cycle of statements involving (e) and (f).

However, when execution reaches step (e), the following sequence may be executed instead:

(e) *T0* executes (7).

(f) *T1* resumes execution at (4) and executes (5), (2), (6), and (7).

Thus, *T1* is not deadlocked or livelocked, but it is starved in the cycle involving (e) and (f).

If contention for a critical section is low, as it is in many cases, then starvation is unlikely, and solutions to the critical section problem that theoretically allow starvation may actually be acceptable.

This is the case for the partial hardware solution in section 2.3.1.

2.5 Tracing and Replay for Shared Variables

Assume:

- Read and write operations are atomic, i.e. each thread issues its read and write operations according to the order in its code and its operations are serviced by the memory system one-at-a-time.
- The total order that results is consistent with the (partial) order specified by each thread's code. (As we will see in Section 2.5.6, this assumption may not always hold.)
- The interleaving of read and write operations is the only source of non-determinism. (e.g., no uninitialized variables or memory allocation problems.)

An execution of a program with the same input and the same sequence of read and write operations will produce the same result.

2.5.1 ReadWrite-sequences

In general:

- A thread executes synchronization events on synchronization objects.
- Each shared object in a concurrent program is associated with a sequence of synchronization events, called a SYN-sequence, which consists of the synchronization events that are executed on that object.
- A SYN-sequence of a concurrent program is a collection of the SYN-sequences for its synchronization objects

For programs like Peterson's algorithm, the synchronization objects are shared variables, and the synchronization events are read and write operations on the shared variables.

.

A SYN-sequence for a shared variable *v* is a sequence of read and write operations, called a ReadWrite-sequence.

A version number is associated with each read and write operation performed by a thread. Each write operation on a shared variable creates a new version of the variable and thus increases the variable's version number by 1.

An execution can be replayed by ensuring each thread reads and writes the same versions of variables that were recorded in the execution trace.

The format for a read event is: Read(thread ID, current version number of variable).

The format for a write event is: Write(thread ID, old version number of variable, total readers),

where

- *old version number* is the number before the increment by this write
- *total readers* is the number of readers that read the old version of the variable.

```
int s = 0; // shared variable
Thread1     Thread2     Thread3
temp = s;   temp = s;   s = s + 1;
```

A possible ReadWrite-sequence of *s* is

```
Read(3,0)      // Thread3 reads version 0 of s
Read(1,0)      // Thread1 reads version 0 of s
Write(3,0,2)   // 2 readers read version 0 of s
Read(2,1)      // Thread2 reads version 1 of s
```

This ReadWrite-sequence specifies that during replay Thread3 should write *s* after Thread1 reads *s* but before Thread2 reads *s*.

Note:

- The value read or written by a memory operation is not recorded in a ReadWrite-sequence. If the order of read and write operations is replayed as specified by a ReadWrite-sequence, then the values read and written will also be replayed.
- We do not actually need to label the events in a ReadWrite-sequence as “Read” or “Write” events. The type (read or write) of the next operation to be performed by a thread is fixed by its source code, the program input, and the operations that precede it. We specify “Read” or “Write” to make the sequence more readable.

2.5.2 An Alternative Definition of ReadWrite-sequences

For each Read or Write operation, record the ID of the thread that executes the operation.

1, 1, 1, 1, 1, 2, 2, 2 // Thread1 executed 5 ops and then Thread2 executed 3 ops.

Compress this sequence into

(1, 5), (2, 3) // format is (a, b): *a* is the thread ID and *b* is the number of operations

This indicates that Thread1 executed the first five operations (1, 5) then Thread2 executed the next three (2, 3).

Compressing sequences may reduce the amount of space that is required for storing traces.

2.5.3 Tracing and Replaying ReadWrite-sequences

During program tracing, information is recorded about each read and write operation, and a ReadWrite-sequence is recorded for each shared variable.

During program replay, the read and write operations on a shared variable are forced to occur in the order specified by the ReadWrite-sequence recorded during tracing.

Read and write operations are instrumented by adding routines to control the start and end of the operations:

```
Read(x) {              Write(x) {
   startRead(x);          startWrite(x);
   read value of x;       write new value of x;
   endRead(x);            endWrite(x);
}                      }
```

Functions *startRead()*, *endRead()*, *startWrite(),* and *endWrite()* contain the trace and replay code. Listing 2.1 shows a sketch of these functions.

In trace mode, these functions record the thread IDs of the reading and writing threads and the version numbers of the variables . They also keep track of the total number of threads that read each version of a variable.

In replay mode, *startRead(x)* delays a thread's read operation until the version number of a variable matches the version number recorded during tracing. If two or more readers read a particular version during tracing, then they must read the same version during replay, but these reads can be in any order. Write operations are delayed in *startWrite(x)* until the *version* number of *x* matches the *version* number recorded during tracing, and until *totalReaders* reaches the same value that it reached during tracing.

```
startRead(x) {
   if (mode == trace) {
       ++x.activeReaders; // one more reader is reading x
       ReadWriteSequence.record(ID,x.version);  // record read event for x
   }
   else {   // replay mode
      // get next event to be performed by thread ID
      readEvent r = ReadWriteSequence.nextEvent(ID);
      myVersion = r.getVersion(); // find version of x read during tracing
      while (myVersion != x.version) delay; // wait for correct version of x to read
   }
}
endRead(x) {
   ++x.totalReaders;    // one more reader has read this version of x
   --x.activeReaders;   // one less reader is reading x (ignored in replay mode)
}
startWrite(x) {
   if (mode = trace) {
   // wait for all active readers to finish reading x
   while (x.activeReaders>0) delay;
      // record write event for x
      ReadWriteSequence.record(ID,x.version,x.totalReaders);
   }
   else { // replay mode
      // find version modified during tracing
      writeEvent w = ReadWriteSequence.nextEvent(ID);
      myVersion = w.getVersion();
      // wait for correct version of x to write
      while (myVersion != x.version) delay;
      // find count of readers for previous version
      myTotalReaders = w.getTotalReaders();
      // wait until all readers have read this version of x
      while (x.totalReaders < myTotalReaders) delay;
   }
}
endWrite(x) {
   x.totalReaders = 0;
   ++x.version;              // increment version number for x
}
```

Listing 2.1 Tracing and Replaying a ReadWrite-sequence.

2.5.4 Class Template sharedVariable<>

Class template *sharedVariable* is used to create shared variables that are of primitive types such as *int* and *double*.

Class *sharedVariable* provides operators for adding, subtracting, assigning, comparing, etc, and implements functions startRead, endRead, startWrite, and endWrite, for tracing and replaying the read and write operations in the implementations of these operators.

Listing 2.3 shows the implementation of operator+=( ).

```
template <class T>
class sharedVariable {
public:
   const sharedVariable<T>& operator+=(const sharedVariable<T>& sv);
   …
private:
   T value; // actual value of the shared variable
   …
};

template<class T>
sharedVariable<T>::operator+=(const sharedVariable<T>& sv) const {
// implementation of operation A += B, which is shorthand for A = A + B
   T svTemp, thisTemp;
   startRead(*this);
   thisTemp = value;            // read A
   endRead(*this);
   startRead(sv);
   svTemp = sv.value;           // read B
   endRead(sv);
   startWrite(*this);
   value = thisTemp + svTemp;   // write A
   endWrite(*this);
   return *this;
}
```

Listing 2.3 Implementation of *sharedVariable<T>::operator+=( ).*

2.5.5 Putting it all Together

Listing 2.4 shows a C++ implementation of Peterson's algorithm using *sharedVariables*.

Each thread enters its critical section twice.

```
sharedVariable<int> intendToEnter1(0), intendToEnter2(0), turn(0);
const int maxEntries = 2;
class Thread1: public TDThread {        // TDThread generates thread IDs
private:
   virtual void* run() {
      for (int i=0; i<maxEntries; i++) {
         intendToEnter1 = 1;
         turn = 2;
         while (intendToEnter2 && (turn == 2))  ;
         // critical section
         intendToEnter1 = 0;
      }
      return 0;
   }
};
class Thread2 : public TDThread {
private:
   virtual void* run() {
      for (int i=0; i<maxEntries; i++) {
          intendToEnter2 = 1;
          turn = 1;
          while (intendToEnter1 && (turn == 1))  ;
          // critical section
          intendToEnter2 = 0;
      }
      return 0;
   }
};
int main() {
   std::auto_ptr<Thread1> T1(new Thread1); std::auto_ptr<Thread2> T2(new Thread2);
   T1->start();  T2->start();  T1->join();   T2->join();
   return 0;
}
```

Listing 2.4 Peterson's algorithm using class template *sharedVariable*<>.

Figure 2-5 shows ReadWrite-sequences produced for shared variables *intendtoEnter1*, *intendToEnter2*, and *turn*. A totally-ordered sequence is given in Fig. 2-5 in the text.

ReadWrite-sequence for *intendToEnter1*:

```
1 0 0    // T1: intendtoenter1 = 1
2 1      // T2: while(intendtoenter1 ... )  –  T2 is busy-waiting
1 1 1    // T1: intendtoenter1 = 0
2 2      // T2: while(intendtoenter1 ... )  – T2 will enter its critical section
1 2 1    // T1: intendtoenter1 = 1
2 3      // T2: while(intendtoenter1 ... )  – T2 is busy-waiting
1 3 1    // T1: intendtoenter1 = 0
2 4      // T2: while(intendtoenter1 ... )  –  T2 will enter its critical section
```

ReadWrite-sequence for *intendToEnter2*:

```
2 0 0    // T2: intendtoenter2 = 1
1 1      // T1: while(intendtoenter2 ... ) – T1 will enter its critical section
2 1 1    // T2: intendtoenter2 = 0
2 2 0    // T2: intendtoenter2 = 1
1 3      // T1: while(intendtoenter2 ... ) – T1 is busy-waiting
1 3      // T1: while(intendtoenter2 ... ) – T1 will enter its critical section
2 3 2    // T2: intendtoenter2 = 0
```

ReadWrite-sequence for *turn*:

```
1 0 0    // T1: turn = 2
2 1 0    // T2: turn = 1
1 2      // T1: while(... && turn == 2) – T1 will enter its critical section
2 2      // T2: while(... && turn == 1) – T2 is busy-waiting
2 2      // T2: while(... && turn == 1) – T2 will enter its critical section
1 2 3    // T1: turn = 2
1 3      // T1: while(.. && turn == 2) – T1 is busy-waiting
2 3 1    // T2: turn = 1
1 4      // T1: while(... && turn == 2) – T1 will enter its critical section
2 4      // T2: while(... && turn == 1) – T2 is busy-waiting
2 4      // T2: while(... && turn == 1) – T2 will enter its critical section
```

Figure 2.5 ReadWrite-sequences for sharedVariables *intendtoEnter1*, *intendToEnter2*, and *turn*.

2.5.6 A Note on Shared Memory Consistency

We have been assuming that read and write operations in different threads are interleaved, but operations in the same thread occur in the order specified by that thread's code.

Lamport: A multiprocessor system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Sequential consistency does not require the actual execution order of operations to be consistent with any *particular* sequential order. Operations of the same processor may be performed in an order that differs from program order as long as both orders give the same result.

Similarly, the events exercised during replay may be performed in an order that differs from the original execution as long as both executions give the same result (e.g., two concurrent write events on different shared variables can be performed in either order).

Sequential consistency may not hold for several reasons:

- Due to performance optimizations performed by the compiler or the memory hardware, the actual order of read and write operations by a thread may be different from the order specified in the program and the result of the execution may violate the semantics of sequential consistency.

- If multiple copies of the same shared variable exist, as is the case with cache-based multiprocessor systems and certain compiler optimizations, and if all the copies are not updated at the same time, then different threads can individually observe different ReadWrite-sequences during the same execution.

In such cases:

- The effects of compiler and hardware optimizations create a larger number of possible interleavings than is implied by the program's source code.

- The actual order of read and write operations might not be known at the program level; rather, the order may be known only at the hardware level.

- Verifying program correctness becomes a "monumental task"

Example:

```
int x = 1;              // x and y are shared variables
boolean y = false;
Thread1   Thread2
x = 0;    if (y)
y = true;   /* what is the value of x? */
```

Thread2 might find that *y* is *true*, but x is not 0:

Problem 1: If each thread is allowed to keep a private copy of shared variable *y* in a register in order to speed up access to *y*, then the update to *y* by *Thread1* will not be visible to *Thread2*. Thus, *Thread2* will always find that *y* is *false*.

Problem 2: Hardware optimizations such as allowing write operations to overlap can result in reorderings of the write operations in *Thread1*. Thus, *y* may be assigned *true* before *x* is assigned 0.

Problem 3: A compiler optimization may reorder the assignment statements in *Thread1* so that the assignment to *y* is performed before the assignment to *x*.

Guaranteeing sequential consistency would rule out these and some other optimizations, which would also mean that these optimizations could not be used to speed up individual sequential processors.

Instead of slowing down all the processors, programmers are asked to identify the places in the program at which the optimizations should be turned off, e.g., declaring a variable as *volatile*.

Declaring a variable as *volatile* indicates that the value of the variable may be read or written from outside the thread or even outside the program in which the variable appears.

In the example program above, the value of variable *y* in *Thread2* can be changed by *Thread1*. If *y* is declared as volatile, the compiler will ensure that updates to *y* by *Thread1* are immediately visible to *Thread2*.

A variable's memory location may be mapped to a data port for an input device:

- This allows the program to perform input simply by reading the value of the variable.
- Since the variable is written by the input device, not by the program, the compiler may assume that the value of the variable never changes and then perform optimizations that are based on this incorrect assumption.
- Declaring the variable as volatile turns these optimizations off for this variable.

In our example program above, we can deal with problem (1) above by declaring *y* to be volatile:

```
volatile boolean y = false;
int x = 1;
```

This prohibits the compiler from allocating *y* to a register.

However, in versions of Java before J2SE 5.0, problems (2) and (3) remain. In these earlier versions, reads and writes of volatile variables cannot be reordered with reads and writes of other volatile variables, but they can be reordered with reads and writes of non-volatile variables.

This type of reordering is prohibited in J2SE 5.0.

To address problems (2) and (3), and to ensure that the reordering rules for volatile variables are obeyed, the compiler emits special “memory barrier” instructions, which control the interactions among the caches, memory, and CPUs.

For example, on a Pentium processor, the atomic XADD (Exchange and Add) instruction is a memory barrier.

Prior to executing a memory barrier instruction, the processor ensures that all previous read and write operations have been completed.

The semantics of volatile variables and other issues related to shared memory consistency in Java are spelled out in the Java Memory Model, which was updated in J2SE 5.0.

Since C++ does not have threads built-in to the language, C++ does not have a memory model like Java’s. The reorderings that are permitted in a multithreaded C++ program depend on the compiler, the thread library being used, and the platform on which the program is run.

God news: The high-level synchronization constructs that we will study in later chapters are implemented with memory barrier instructions that guarantee sequential consistency as a side effect of using the constructs to create critical sections.

3. Semaphores and Locks

Semaphores are used to provide mutual exclusion and condition synchronization. Locks provide mutual exclusion and have special properties that make them useful in object-oriented programs.

3.1 Counting Semaphores

A counting semaphore is a synchronization object that is initialized with an integer value and then accessed through two operations, which are named *P* and *V* (or *down* and *up*, *decrement* and *increment*, *wait* and *signal*).

```
class countingSemaphore {
   public countingSemaphore(int initialPermits) {permits = initialPermits;}
   public void P() {...};
   public void V() {...};
   private int permits;
}
```

When defining the behavior of methods *P*() and *V*(), it is helpful to interpret a counting semaphore as having a pool of permits.

- A thread calls method *P*() to request a permit. If the pool is empty, the thread waits until a permit becomes available.
- A thread calls method *V*() to return a permit to the pool.

A counting semaphore *s* is declared and initialized using

```
countingSemaphore s(1);
```

The initial value, in this case 1, represents the initial number of permits in the pool.

A sketch of a possible implementation of methods P() and V().

```
public void P( ) {
if (permits > 0)    // permits holds the current number of permits in the pool:
      --permits;   // take a permit from the pool
   else               // the pool is empty so wait for a permit
      wait until permits becomes positive and then decrement permits by one.
}


public void V( ) {
   ++permits;     // return a permit to the pool
}
```

There may be many threads waiting in *P()* for a permit. The waiting thread that gets a permit as the result of a *V*() operation is not necessarily the thread that has been waiting the longest.

The invariant for semaphore *s*:

(the initial number of permits) + (the number of completed *s.V*() operations)

≥ (the number of *completed s.P*() operations).

The number of completed *P()* operations may be less than the number of started *P()* operations.

There are no methods for accessing the value of a semaphore. We rely on a semaphore's invariant to define its behavior.

3.2 Using Semaphores

There are some common idioms or mini-patterns in the way semaphores are used to solve problems. Being able to recognize and apply these patterns is a first step towards understanding and writing semaphore-based programs.

3.2.1 Resource Allocation

Problem: Three threads are contending for two resources. If neither resource is available, a thread must wait until one of the resources is released.

Solution:

```
               countingSemaphore s(2); // two resources are available initially
Thread 1                    Thread 2                    Thread 3
s.P();                      s.P();                      s.P();
/* use the resource */      /* use the resource */      /* use the resource */
s.V();                      s.V();                      s.V();
```

Listing 3.1 Resource allocation using semaphores.

The invariant for semaphore *s* and the placement of the *P()* and *V()* operations guarantees that there can be no more than two consecutive completed *s.P()* operations without an intervening *s.V()* operation.

The pool of permits represented by semaphore *s* maps directly to the managed pool of resources. Methods *P*() and *V*() do all of the necessary bookkeeping internally - counting resources, checking the number of available resources, and blocking threads when no resources are available.

3.2.2 More Semaphore Patterns

Listing 3.2 shows an alternate solution to the resource allocation problem.

```
 // variables shared by the threads
 int count = 2;                      // number of available resources
 int waiting = 0;                    // number of waiting threads
 // provides mutual exclusion for count and waiting
 countingSemaphore mutex = new countingSemaphore(1);
 // used as a queue of blocked threads
 countingSemaphore resourceAvailable = new countingSemaphore(0);
 Thread i { // each thread executes the following code:
 mutex.P();                          // enter the critical section
 if (count>0) {                      // is a resource available?
   count--;                          // one less resource available
   mutex.V();                        // exit the critical section
 }
 else {
   waiting++;                        // one more waiting thread
   mutex.V();                        // exit the critical section
   resourceAvailable.P();            // wait for a resource
 }
 /* use the resource */
 mutex.P();                          // enter the critical section
 if (waiting>0) {                    // are there waiting threads?
   --waiting;                        // one less waiting thread
   resourceAvailable.V();            // notify a waiting thread
 }
 else count++;                       // return a resource to the pool
 mutex.V();
}
```

Listing 3.2 An alternate solution to the resource allocation problem.

- Variable *count* tracks the number of available resources.
- Shared variable *waiting* tracks the number of waiting threads.
- Notice that *count* is not incremented when a waiting thread is notified, just as *count* is not decremented after a waiting thread receives a resource.

This solution contains several patterns.

Mutex: A semaphore, typically named *mutex* (for "*mut*ual *ex*clusion") is initialized to one. A critical section begins with a call to *mutex.P*() and ends with a call to *mutex.V*():

```
mutex.P()
/* critical section */
mutex.V()
```

The semaphore invariant ensures that the completion of *P()* and *V()* operations alternates, which allows one thread at a time to be inside the critical section.

Enter-and-Test: Often a thread will enter a critical section and then test a condition that involves shared variables.

```
mutex.P();
if (count>0) {          // test for available resources; count is a shared variable
  …;                    // count>0 means that resource is available
}
else {
…;
resourceAvailable.P(); // wait for a resource
}
```

One of the alternatives of the if-statement will contain a *P*() operation so that the thread can block itself until it is notified that the condition is satisfied.

Exit-before-Wait: A thread executing inside a critical section will exit the critical section before blocking itself on a *P*() operation.

```
mutex.V();                // exit critical section
resourceAvailable.P();    // wait for a resource
```

If a thread did not call *mutex.V*() to leave the critical section before it called *resourceAvailable.P*(), then no other threads would be able to enter the critical section and no calls to *resourceAvailable.V*() would ever occur.

Condition Queue: A semaphore can be used as a queue of threads that are waiting for a condition to become true. If the initial value of a semaphore *s* is 0, and the number of started *s.P()* operations is never less than the number of completed *s.V()* operations, then the semaphore invariant ensures that every *s.P*() operation is guaranteed to block the calling thread.

```
  if (waiting>0)          // if one or more threads are waiting
     resourceAvailable.V(); // then notify a blocked thread
```

A call to *resourceAvailable.P*() will always block the calling thread and a call to *resourceAvailable.V*() will always unblock a waiting thread.

3.3 Binary Semaphores and Locks

A binary semaphore must be initialized with the value 1 or the value 0 and the completion of *P()* and *V()* operations must alternate. (Note that *P()* and *V()* operations can be started in any order, but their completions must alternate.)

- If semaphore's initial value is 1, the first completed operation must be *P()*.
- If semaphore's initial value is 0, the first completed operation must be V().

Thus, the *P()* and *V()* operations of a binary semaphore may block the calling threads.

```
binarySemaphore mutex = new binarySemaphore(1);
Thread 1              Thread 2
mutex.P();            mutex.P();
/* critical section */ /* critical section */
mutex.V();            mutex.V();
```

A “mutex lock” (or “lock”), can also be used to solve the critical section problem. (Locks can provide mutual exclusion but not condition synchronization.)

```
mutexLock mutex;
Thread 1                 Thread 2
mutex.lock();            mutex.lock();
/* critical section */   /* critical section */
mutex.unlock();          mutex.unlock();
```

Unlike a semaphore, a lock has an owner, and ownership plays an important role in the behavior of a lock:

- A thread requests ownership of lock *L* by calling *L.lock()*.
- A thread that calls *L.lock()* becomes the owner if no other thread owns the lock; otherwise the thread is blocked.
- A thread releases its ownership of *L* by calling *L.unlock()*. If the thread does not own *L*, the call to *L.unlock()* generates an error.
- A thread that already owns lock *L* and calls *L.lock()* again is not blocked. In fact, it is common for a thread to request and receive ownership of a lock that it already owns. But the owning thread must call *L.unlock()* the same number of times that it called *L.lock()* before another thread can become *L*’s owner.
- A lock that allows its owning thread to lock it again is called a *recursive* lock.

Locks are commonly used in the methods of classes.

```
class lockableObject {
    public void F() {
        mutex.lock();
        ...;
        mutex.unlock();
    }
    public void G() {
        mutex.lock();
        ...; F(); ...;                  // method G() calls method F()
        mutex.unlock();
    }
    private mutexLock mutex;
}
```

Methods *F()* and *G()* are critical sections.

When a thread calls method *G()*, the *mutex* is locked. When method *G()* calls method *F()*, *mutex.lock()* is executed in *F()*, but the calling thread is not blocked since it already owns *mutex*.

Differences between locks and binary semaphores:

- For a binary semaphore, if two calls are made to *P*() without any intervening call to *V*(), the second call will block. But a thread that owns a lock and requests ownership again is not blocked. (Beware of the fact that locks are not always recursive, so check the documentation before using a lock.)

- The owner for successive calls to *lock()* and *unlock()* must be the same thread. But successive calls to *P*() and *V*() can be made by different threads.

3.4 Implementing Semaphores

Semaphores can be implemented at the user level, the operating system level, or with hardware support.

3.4.1 Implementing P() and V()

Implementation 1 in Listing 3.3 uses busy waiting to delay threads. It's possible that an awakened thread will always find that the value of *permits* is zero and will thus never be allowed to complete its *P()* operation.

Semaphores with this type of implementation are called *weak semaphores*.

Implementation 2 blocks waiting threads in a (operating system) queue until they are notified. When a thread blocked in *P()* is awakened by a *V()* operation, the thread is allowed to complete its *P()* operation. This is a s*trong semaphore*.

```
// Implementation 1 uses semi-busy-waiting.
P():    while (permits == 0) {
           Sleep(..);      // voluntarily relinquish CPU
        }
        permits = permits - 1;
V():    permits = permits + 1;

// Implementation 2 uses a queue of blocked threads; permits may be negative.
P():    permits = permits - 1;
        if (permits < 0) wait on a queue of blocked threads until notified;
V():    permits = permits + 1;
        if (permits <= 0) notify one waiting thread;
```

Listing 3.3 Implementations of *P()* and *V()* for counting semaphores.

Implementation 3 in Listing 3.4 is a (strong) binary semaphore.

According to this implementation, the invariant for a binary semaphore *s* is

```
((the initial value of s (which is 0 or 1)) +
    (the number of completed s.V() operations)
        - (the number of completed s.P() operations)) = 0 or 1.
```

```
// Implementation 3 uses two queues of blocked threads. V() may block caller.
P():    if (permits == 0)
            wait in a queue of blocked threads until notified;
        permits = 0;
        if (queue of threads blocked in V() is not empty)
            notify one blocked thread in V();

V():    if (permits == 1)
            wait in a queue of blocked threads until notified;
        permits = 1;
        if (queue of threads blocked in P() is not empty)
            notify one blocked thread in P();
```

Listing 3.4 Implementation of *P()* and *V()* for binary semaphores.

Since methods *P*() and *V*() access shared variables (e.g., permits), they must be critical sections. Here's an implementation using the techniques in Chapter 2:

```
P(): entry-section;
     while (permits == 0) {
        exit-section;
        ;  // null-statement
        entry-section;
     };
     permits = permits - 1;
     exit-section;

V(): entry-section;
     permits = permits + 1;
     exit-section;
```

Semaphores can also be implemented at the operating system level.

Listing 3.5 shows implementations of *P()* and *V()* as operations in the kernel of an operating system. Critical sections are created by disabling and enabling interrupts.

```
P(s): disable interrupts;
      permits = permits - 1;
      if (permits < 0) {
         add the calling thread to the queue for s and change its state to blocked;
         schedule another thread;
      }
      enable interrupts;


V(s): disable interrupts;
      permits = permits + 1;
      if (permits <= 0)  {
         select a thread from the queue for s; change the thread's state to ready;
      }
      enable interrupts;
```

Listing 3.5 Implementation of *P()* and *V()* in the kernel of an operating system.

For a shared-memory multiprocessor machine, disabling interrupts may not work, so atomic hardware instructions like those described in Section 2.3 can be used.

3.4.2 The VP() Operation

Instead of writing

```
s.V();
t.P();
```

we combine the separate *V()* and *P()* operations into a single atomic *VP*() operation:

```
t.VP(s);
```

An execution of *t.VP*(*s*) is equivalent to *s.V*(); *t.P*(), except that during the execution of *t.VP*(*s*), no intervening *P*(), *V*(), or *VP*() operations are allowed to be started on *s* and *t*.

Operation *t.VP*(*s*) can only be used in cases where the *V()* operation on *s* cannot block the calling thread. This means that *s* is a counting semaphore, or *s* is or a binary semaphore that is guaranteed to be in a state in which a *V()* operation will not block.

A context switch between successive *V*() and *P*() operations is often a source of subtle programming errors. A *V*() operation followed by a *P*() operation appears in the *Exit-before-Wait* pattern:

```
mutex.V();                // exit critical section
someCondition.P();        // wait until some condition is true
```

- A thread T that executes *mutex.V*() releases mutual exclusion.

- Other threads may enter their critical sections and perform their *mutex.V()* and *someCondition.P()* operations before thread T has a chance to execute *someCondition.P*().

- This can create a problem, which is often in the form of a deadlock. The *VP*() operation removes these types of errors.

3.5 Semaphore-Based Solutions to Concurrent Programming Problems

These problems demonstrate how semaphores can be used to solve a commonly occurring synchronization problem.

3.5.1 Event Ordering

Assume that code segment *C1* in *Thread1* has to be executed after code segment *C2* in *Thread2*. Let *s* be a counting or binary semaphore initialized to 0.

```
Thread1   Thread2
s.P();    C2;
C1;       s.V();
```

The *s.P()* operation will block *Thread1* until *Thread2* does its *s.V()* operation. This guarantees that code segment *C1* is executed after segment *C2*.

3.5.2 Bounded Buffer

A bounded buffer has *n* slots. Each slot is used to store one item. A producer is not permitted to deposit an item when all the slots are full. A consumer is not permitted to withdraw an item when all the slots are empty.

The solution to the bounded buffer problem in Listing 3.6 uses semaphores *fullSlots* and *emptySlots* as resource counters, counting the full and empty slots, respectively, in the buffer.

The condition (number of full slots + number of empty slots == n) is true before and after each *deposit* and *withdraw* operation.

```
int buffer[ ] = new int[n];
countingSemaphore emptySlots(n);             // the number of empty slots
countingSemaphore fullSlots(0);              // the number of full slots

Producer {
   int in = 0;
   int item;
   ...
   /* produce item */
   emptySlots.P();       // wait if there are no empty slots
   buffer[in] = item;    // in is the index for a deposit
   in = (in + 1) % n;    // 0 ≤ in < n; and in = (out+#items in buffer)%n
   fullSlots.V();        // signal that a slot was filled
   ...
}

Consumer {
   int out = 0;
   int item;
   fullSlots.P();        // wait if there are no full slots
   item = buffer[out];   // out is the index for a withdraw
   out = (out + 1) % n;  // 0 ≤ out < n
   emptySlots.V();       // signal that a slot was emptied
   /* consume item */
}
```

Listing 3.6 Bounded buffer using semaphores.

Since *in* and *out* cannot have the same value when the buffer is accessed, no critical section is required for accessing the slots in the *buffer*.

3.5.3 Dining Philosophers

There are *n* philosophers who spend their time eating and thinking [Dijkstra 1971]. They sit at a table with *n* seats. A bowl of rice sits in the center of the table. There is one chopstick between each pair of philosophers.

- When a philosopher is hungry, she picks up the two chopsticks that are next to her, one at a time.
- When she gets her chopsticks, she holds them until she is finished eating.
- Then she puts down her chopsticks one at a time and goes back to thinking.

Solutions to the dining philosophers problem are required to be free from deadlock and starvation. An additional property, called "maximal parallelism", may also be required. This property is satisfied by a solution that allows a philosopher to eat as long as her neighbors are not eating.

3.5.3.1 Solution 1. In the solution in Listing 3.7, each chopstick is represented as a binary semaphore, which serves as a simple resource counter that is initialized to 1. Picking up a chopstick is implemented as a *P()* operation, and releasing a chopstick is implemented as a *V()* operation. A global deadlock occurs when each philosopher holds her left chopstick and waits forever for her right chopstick.

```
philosopher(int i  /* 0..n-1 */) {
   while (true) {
      /*  think */
      chopsticks[i].P();              // pickup left chopstick
      chopsticks[(i+1) % n].P();   // pickup right chopstick
      /*  eat */
      chopsticks[i].V();              // put down left chopstick
      chopsticks[(i+1) % n].V();   // put down right chopstick
   }                              }
```

Listing 3.7 Dining philosophers using semaphores – Solution 1.

3.5.3.2 Solution 2. This solution is the same as Solution 1 except that only (*n*-1) philosophers are allowed to sit at a table that has *n* seats. A semaphore *seats* with initial value (*n*-1) is used as a resource counter to count the number of available seats. Each philosopher executes *seats.P()* before she picks up her left chopstick and *seats.V()* after she puts down her right chopstick.

- maximal parallelism is not satisfied (two neighboring philosophers hold a single chopstick but are unwilling to let each other eat).
- deadlock-free.
- if semaphores with First-Come-First-Serve (FCFS) *P()* and *V()* operations are used this solution is also starvation free.

3.5.3.3 Solution 3. This solution is the same as Solution 1 except that one philosopher is designated as the "odd" philosopher. The odd philosopher picks up her right chopstick first (instead of her left chopstick).

- maximal parallelism is not satisfied
- deadlock-free.
- if semaphores with First-Come-First-Serve (FCFS) *P()* and *V()* operations are used this solution is also starvation free.

3.5.3.4. Solution 4. In the solution in Listing 3.8, a philosopher picks up two chopsticks only if both of them are available. Each philosopher has three possible states: thinking, hungry and eating.

- A hungry philosopher can eat if her two neighbors are not eating.
- After eating, a philosopher unblocks a hungry neighbor who is able to eat.
- This solution is deadlock-free and it satisfies maximal parallelism, but it is not starvation-free. A hungry philosopher will starve if whenever one of her neighbors puts her chopstick down the neighbor on her other side is eating.

```
final int thinking = 0; final int hungry = 1; final int eating = 2;
int state[] = new int[n];
for (int j = 0; j < n; j++) state[j] = thinking;
// mutex provides mutual exclusion for accessing state[].
binarySemaphore mutex = new binarySemaphore(1);
// philosopher i blocks herself on self[i] when she is hungry but unable to eat
binarySemaphore self[] = new binarySemaphore[n];
for (int j = 0; j < n; j++) self[j] = new binarySemaphore(0);

philosopher(int i /* 0..n-1 */) {
   while (true) {
      /* think */
      mutex.P()
      state[i] = hungry;
      test(i);            // performs self[i].V() if philosopher i can eat
      mutex.V();
      self[i].P();        // self[i].P() will not block if self[i].V() was
      /* eat */           //    performed during call to test(i)
      mutex.P();
      state[i] = thinking;
      test((i - 1) % n); // unblock left neighbor if she is hungry and can eat
      test((i +1) % n); // unblock right neighbor she is hungry and can eat
      mutex.V();
   }
}
void test(int k /* 0..n-1 */) {
// philosopher i calls test(i) to check whether she can eat.
// philosopher i calls test((i - 1) % n) when she is finished eating to unblock a
// hungry left neighbor.
// philosopher i calls test((i + 1) % n) when she is finished eating to unblock a
// hungry right neighbor.
   if ((state[k] == hungry) && (state[(k+n-1) % n)] != eating) &&
        (state[(k+1) % n] != eating)) {
     state[k] = eating;
     self[k].V();   // unblock philosopher i's neighbor, or guarantee that
   }                // philosopher i will not block on self[i].P().
}
```

Listing 3.8 Dining philosophers using semaphores – Solution 4.

Array *self[]* is an array of semaphores used to block philosophers who are unable to eat.

Philosopher i blocks itself on semaphore *self[i]* when she is hungry but unable to eat.

Philosopher *i*'s first call to *test()* is to check whether her neighbors are eating:

- if neither neighbor is eating, Philosopher *i* executes *self[i].V()* in function *test()*. Since only Philosopher i executes a *P()* operation on *self[i]*, and since Philosopher *i* is obviously not blocked on semaphore *self[i]* at the time she executes *self[i].V()*, no thread is unblocked by this *V()* operation. Furthermore, since *self[i]* is initialized to 0, this *V()* operation does not block and Philosopher *i* is allowed to continue. The purpose of this *V()* operation is not clear until we notice that Philosopher *i* immediately thereafter executes *self[i].P()*, which, thanks to the just performed *V()* operation, is guaranteed not to block.
- if one or both of Philosopher *i*'s neighbors are eating when *test(i)* is called, then *self[i].V()* is not executed, causing Philosopher *i* to be blocked when she executes *self[i].P()*.

The two other calls to *test()* that Philosopher *i* makes are to unblock hungry neighbors that are able to eat when Philosopher *i* finishes eating.

3.5.4 Readers and Writers

Data is shared by multiple threads [Courtois et al. 1971]. When a thread reads (writes) the shared data, it is considered to be a reader (writer). Readers may access the shared data concurrently, but a writer always has exclusive access.

Table 3.1 shows six different strategies for controlling how readers and writers access the shared data. These strategies fall into one of three categories based on whether readers or writers get priority when they both wish to access the data:

1. R=W: readers and writers have equal priority; served together in FCFS order.
2. R>W: readers generally have a higher priority than writers.
3. R<W: readers generally have a lower priority than writers.

Access Strategy	**Description**
R=W.1	One reader or one writer with equal priority
R=W.2	Many readers or one writer with equal priority
R>W.1	Many readers or one writer with readers having a higher priority
R>W.2	Same as R>W.1 except that when a reader arrives, if no other reader is reading or waiting, it waits until all writers that arrived earlier have finished
R<W.1	Many readers or one writer with writers having a higher priority
R<W.2	Same as R<W.1 except that when a writer arrives, if no other writer is writing or waiting, it waits until all readers that arrived earlier have finished

Table 3.1 Strategies for the readers and writers problem.

- In strategies R>W.1 and R>W.2, writers will starve if before a group of readers finishes reading there is always another reader requesting to read.
- In strategies R<W.1 and R<W.2, readers will starve if before a writer finishes writing there is always another writer requesting to write.
- In strategies R=W.1 and R=W.2, no readers or writers will starve.

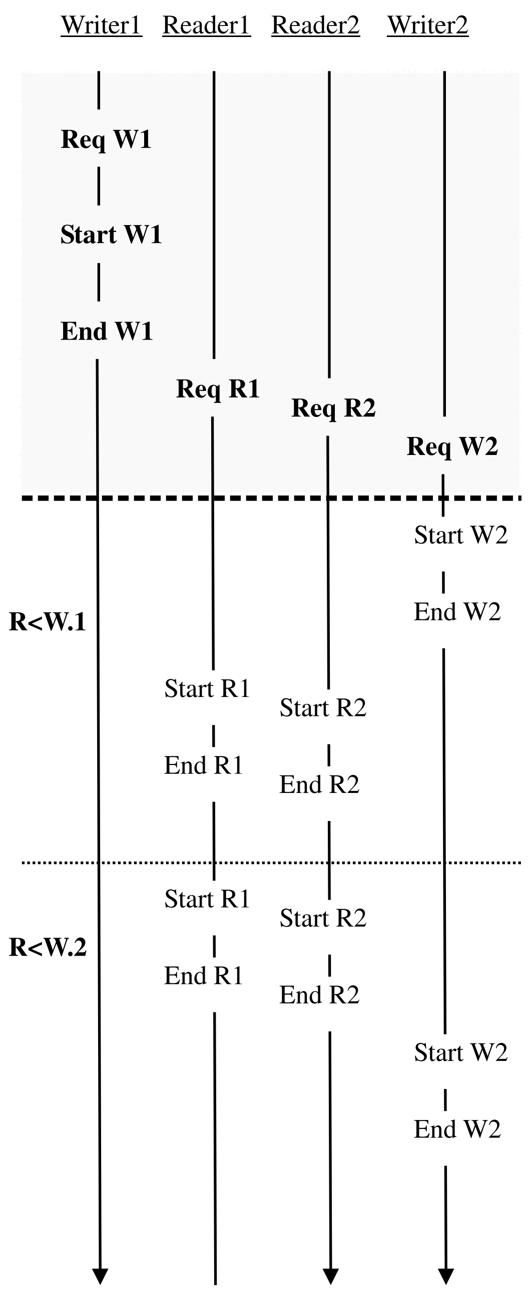
Fig. 3.9 compares strategies R<W.1 and R<W.2. In the shaded part at the top, Reader 1, Reader 2, and Writer 2 issue a request after Writer 1 finishes writing. Below the dashed line, the scenario is completed using the two different strategies.

- For strategy R<W.1, Writer 2 writes before the readers read.
- For strategy R<W.2, both readers are allowed to read before Writer 2 writes. Strategy R<W.2 requires Writer 2 to wait until the readers that arrived earlier have finished since Writer 2 requested to write when no other writer was writing or waiting.

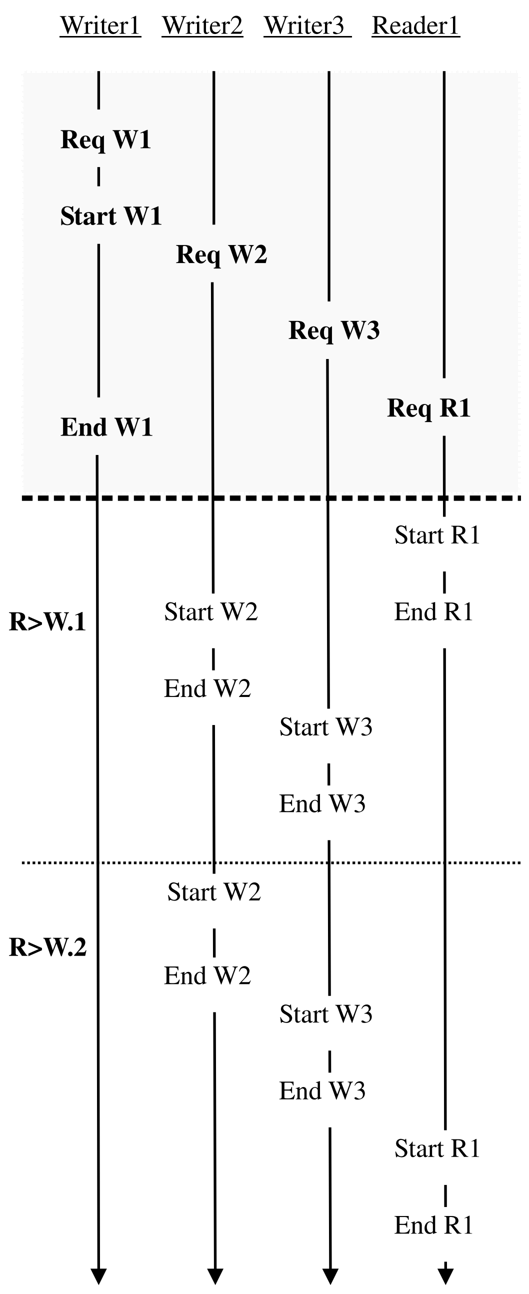
We are assuming that multiple request events can occur before a decision is made as to whether to allow a reader or writer to start. This assumption is important in order to distinguish between the various strategies.

Fig. 3.10 compares strategies R>W.1 and R>W.2. In the shaded part at the top, Reader 1, Writer 2, and Writer 3 issue a request while Writer 1 is writing.

- For strategy R>W.1, Reader 1 reads before the writers writes.
- For strategy R>W.2, both writers are allowed to write before Reader 1 reads. This is because Reader 1 requested to read when no other reader was reading or waiting, so Reader 1 must wait until the writers who arrived earlier have finished.


Writer1
Reader1
Reader2
Writer2
Req W1
Start W1
End W1
Req R1
Req R2
Req W2
Start W2
End W2
R<W.1
Start R1
End R1
Start R2
End R2
Start R1
End R1
Start R2
End R2
R<W.2
Start W2
End W2

3.9


Writer1
Writer2
Writer3
Reader1
Req W1
Start W1
Req W2
Req W3
End W1
Req R1
Start R1
End R1
R>W.1
Start W2
End W2
Start W3
End W3
Start W2
End W2
R>W.2
Start W3
End W3
Start R1
End R1

3.10

3.5.4.1 R>W.1. This strategy gives readers a higher priority than writers and may cause waiting writers to starve.

In Listing 3.11, semaphore *mutex* provides mutual exclusion for the *Read()* and *Write()* operations, while semaphores *readers_que* and *writers_que* implement the Condition Queue pattern from section 3.2.2.

In operation *Write()*, a writer can write if no readers are reading and no writers are writing. Otherwise, the writer releases mutual exclusion and blocks itself by executing writers_que.P(). The *VP()* operation ensures that delayed writers enter the *writer_que* in the same order that they entered operation *Write()*.

In operation *Read()*, readers can read if no writers are writing; otherwise, readers block themselves by executing readers_que.P(). At the end of the read operation, a reader checks to see if any other readers are still reading. If not, it signals a writer blocked on *writers_que*. A continuous stream of readers will cause waiting writers to starve.

At the end of a write operation, waiting readers have priority. If readers are waiting in the *readers_que*, one of them is signaled. This first reader checks to see if any more readers are waiting. If so, it signals the second reader, which signals the third, and so on. This cascaded wakeup continues until no more readers are waiting. If no readers are waiting when a writer finishes, a waiting writer is signaled.

```
int activeReaders = 0;              // number of active readers
int activeWriters = 0;              // number of active writers
int waitingWriters = 0;             // number of waiting writers
int waitingReaders = 0;             // number of waiting readers
binarySemaphore mutex = new binarySemaphore(1); // exclusion
binarySemaphore readers_que = new binarySemaphore(0);// waiting readers
binarySemaphore writers_que = new binarySemaphore(0);// waiting writers
sharedData x ... ;                  // x is shared data

   Read() {                            Write() {
     mutex.P();                           mutex.P();
     if (activeWriters > 0) {             if (activeReaders > 0 ||
                                            activeWriters > 0 ) {
        waitingReaders++;                    waitingWriters++;
        readers_que.VP(mutex);               writers_que.VP(mutex);
     }                                    }
     activeReaders++;                     activeWriters++;
     if (waitingReaders > 0) {            mutex.V();
        waitingReaders--;
        readers_que.V();                  /* write x */
     }
     else                                 mutex.P();
        mutex.V();                        activeWriters--;
                                          if (waitingReaders > 0) {
     /* read x */                            waitingReaders--;
                                             readers_que.V();
     mutex.P();                           }
     activeReaders--;                     else if (waitingWriters > 0) {
     if (activeReaders == 0 &&               waitingWriters--;
        waitingWriters > 0) {                writers_que.V();
           waitingWriters--;              }
           writers_que.V();               else
     }                                       mutex.V();
     else                              }
        mutex.V();
   }
```

Listing 3.11 Strategy R>W.1.

This implementation illustrates another important semaphore pattern called "passing-the-baton".

Notice that delayed readers and writers exit their critical sections before they block themselves on *readers_que.P()* or *writers_que.P()*. This is part of the Exit-and-Wait pattern mentioned in Section 3.2. However, after these delayed threads are signaled, they never execute *mutex.P()* to reenter the critical section.

That is, we expect to see readers executing

```
readers_que.VP(mutex);      // exit critical section (mutex.V()) and wait
                            //    (readers_que.P())
mutex.P();                  // reenter the critical section before continuing
```

but the *mutex.P()* operation is missing.

To understand why this operation is not needed, it helps to think of mutual exclusion as a baton that is passed from thread to thread.

- When a waiting thread is signaled, it receives the baton from the signaling thread. Possession of the baton gives the thread permission to execute in its critical section.
- When that thread signals another waiting thread, the baton is passed again.
- When there are no more waiting threads to signal, a *mutex.V()* operation is performed to release mutual exclusion. This will allow some thread to complete a future *mutex.P()* operation and enter its critical section for the first time.

This technique is implemented using an if-statement, such as the one used in the cascaded wakeup of readers:

```
if (waitingReaders > 0) {  // if another reader is waiting
   waitingReaders--;
   readers_que.V();   // pass the baton to a reader (i.e., do not release mutual
}                     // exclusion)
else
   mutex.V();         // else release mutual exclusion
```

- When a waiting reader is awakened, it receives mutual exclusion for accessing shared variables *activeReaders* and *waitingReaders.*
- If another reader is waiting, the baton is passed to that reader.
- If no more readers are waiting, mutual exclusion is released by executing *mutex.V()*.

3.5.4.2 R>W.2. This strategy allows concurrent reading and generally gives readers a higher priority than writers. Writers have priority in the following situation: when a reader requests to read, if it is a "lead reader" (i.e., no other reader is reading or waiting), it waits until all writers that arrived earlier have finished writing. This strategy may cause waiting writers to starve.

In Listing 3.12, when a reader R executes *Read()*:

- if one or more other readers are reading, then R starts reading immediately
- if one or more other readers are waiting for writers to finish, then R is blocked on *mutex*
- if no reader is reading or waiting then R is a lead reader, so R executes *writers_r_que.P()*.
    - If a writer is writing, then R will be blocked on *writers_r_que* behind any waiting writers that arrived before R. (Writers that arrived before R executed *writers_r_que.P()* before R did.)
    - Otherwise, R can start reading, and writers will be blocked when they execute *writers_r_que.P()*.

When a *Write()* operation ends, semaphore *writers_r_que* is signaled. Waiting writers that arrived before a lead reader will be ahead of the reader in the queue for *writers_r_que*; thus, the lead reader will have to wait for these writers to finish writing.

```
int activeReaders = 0;          // number of active readers
mutexLock mutex = new mutexLock();   // mutual exclusion for activeReaders
// condition queue for waiting writers and the first waiting reader
binarySemaphore writers_r_que = new binarySemaphore(1);
sharedData x ... ;              // x is shared data

Read() {
   mutex.lock();// block readers if lead reader waiting in writers_r_que
   ++activeReaders;
   if (activeReaders == 1)
      writers_r_que.P();  // block lead reader if a writer is writing
   mutex.unlock();

   /* read x */

   mutex.lock();
   --activeReaders;
   if (activeReaders == 0)
      writers_r_que.V(); // allow waiting writers, if any, to write
   mutex.unlock();
}

Write() {
   writers_r_que.P(); // block until no readers are reading or waiting and
   /* write x */     //   no writers are writing
   writers_r_que.V();// signal lead reader or a writer at the front of the queue
}
```

Listing 3.12 Strategy R>W.2.

This solution is interesting because it violates the *Exit-and-Wait* pattern described in Section 3.2.

As a rule, a thread will exit a critical section before blocking itself on a *P*() operation. However, lead readers violate this rule when they execute *writers_r_que.P*() without first executing *mutex.V()* to exit the critical section.

This is a key part of the solution.

- Only a lead reader (i.e. when *activeReaders* == 1) can enter the queue for *writers_r_que* since any other readers are blocked by *mutex.P*() at the beginning of the read operation.
- When the lead reader is released by *writers_r_que.V*(), the lead reader allows the other waiting readers to enter the critical section by signaling *mutex.V*().
- These other readers will not execute *writers_r_que.P*() since (*activeReaders* == 1) is only true for the lead reader. (Note that the name of the semaphore "*writers_r_que*" indicates that multiple writers but only one reader may be blocked on the semaphore.)

3.5.4.3 R<W.2. This strategy allows concurrent reading and generally gives writers a higher priority than readers. Readers have priority in the following situation: when a writer requests to write, if it is a lead writer (i.e., no other writer is writing or waiting), it waits until all readers that arrived earlier have finished reading.

This strategy and strategy R>W.2 are symmetrical. The solution in Listing 3.13 for R<W.2 differs from the solution in Listing 3.12 for R>W.2 as follows:

- semaphore *readers_w_que* is used to allow a lead writer to block readers. Only one writer (a lead writer) can be blocked in this queue.
- semaphore *writers_r_que* is renamed *writers_que,* since readers are never blocked in this queue.
- variable *waitingOrWritingWriters* is used to count the number of waiting or writing writers. At the end of a write operation, readers are permitted to read only if there are no waiting or writing writers.

When a writer W executes *Write()*:

- if one or more other writers are waiting for readers to finish, then W is blocked on *mutex_w*.
- if no writer is writing or waiting then W is a lead writer, so W executes *readers_w_que.P()*.
    - (If a writer is writing, then *waitingOrWritingWriters* will be greater than one after it is incremented by W, so W will not execute *readers_w_que.P()*.)
    - If a reader is reading, then W will be blocked on *readers*_w_*que* behind any waiting readers; otherwise, W can start writing, and readers will be blocked when they execute *readers_w_que.P()*.
- W can be followed by a stream of writers. When the last of these writers finishes, it executes *readers_w_que.V()* to signal waiting readers. If a finishing writer always finds that another writer is waiting, readers will starve.

The use of *readers_w_que* ensures that readers that arrive before a lead writer have a higher priority. This is because readers and lead writers both call *readers_w_que.P()*, and they are served in FCFS order.

- When a reader blocked in *readers_w_que* is given permission to read, it executes *readers_w_que.V()* to signal the next waiting reader, who executes *readers_w_que.V()* to signal the next waiting reader, and so on.
- This cascaded wakeup continues until there are no more waiting readers or until the *readers_w_que.V()* operation wakes up a lead writer.
- The lead writer executes *writers_que.P()* to block itself until the readers have finished.
- The last reader executes *writers_que.V()* to signal the lead writer that readers have finished.

```
int activeReaders = 0;  // number of active readers
int waitingOrWritingWriters = 0;// number of writers waiting or writing
mutexLock mutex_r = new mutexLock();    // exclusion for activeReaders
// exclusion for waitingOrWritingWriters
mutexLock mutex_w = new mutexLock();
binarySemaphore writers_que = new binarySemaphore(1);// waiting writers
// condition queue for waiting readers and the first waiting writer
binarySemaphore readers_w_que = new binarySemaphore(1);
sharedData x ... ;                       // x is shared data
Read() {
   readers_w_que.P(); // serve waiting readers and lead writer FCFS
   mutex_r.lock();
   ++activeReaders;
   if (activeReaders == 1)
   writers_que.P();      // block writers reads are occurring
   mutex_r.unlock();
   readers_w_que.V();// signal the next waiting reader or a lead writer

   /* read x */

   mutex_r.lock();
   --activeReaders;
   if (activeReaders == 0)
   writers_que.V();      // allow writing
   mutex_r.unlock();
}
```

```
Write () {
    // if a lead writer is waiting in readers_w_que, this blocks other writers
    mutex_w.lock();
    ++waitingOrWritingWriters;
    if (waitingOrWritingWriters == 1)  // true if this is a lead writer
    readers_w_que.P();  // block lead writer if there are waiting readers
    mutex_w.unlock();
    writers_que.P();        // block if a writer is writing or a reader is reading

    /* write x */

    writers_que.V();  // signal writing is over; wakes up waiting writer (if any)
    mutex_w.lock();
    -- waitingOrWritingWriters;
    if (waitingOrWritingWriters == 0)      // no writers are waiting or writing
    readers_w_que.V();              // so allow reading
    mutex_w.unlock();
}
```

Listing 3.13 Strategy R<W.2.

3.5.5 Simulating Counting Semaphores

Suppose that a system provides binary semaphores but not counting semaphores. Listing 3.14 shows how to use binary semaphores to implement the *P*() and *V*() operations in a *countingSemaphore* class.

Here is a possible execution scenario for four threads T1, T2, T3, and T4 that are using a *countingSemaphore s* initialized to 0:

1. T1 executes *s.P()*: T1 decrements *permits* to -1 and has a context switch immediately after completing the *mutex.V*() operation but before it executes *delayQ.P*().
2. T2 executes *s.P()*: T2 decrements *permits* to -2 and has a context switch immediately after completing the *mutex.V*() operation but before it executes *delayQ.P*().
3. T3 executes s.V(): T3 increments *permits* to -1 and executes *delayQ.V*().
4. T4 executes s.V(): T4 increments *permits* to 0 and executes *delayQ.V*(). Since T3 just previously completed an *s.V()* operation in which the value of *delayQ* was incremented to 1, T4 is blocked at *delayQ.V*().
5. T1 resumes execution, completes *delayQ.P*(), and then completes its *s.P()* operation.
6. T4 resumes execution and completes its *delayQ.V()* operation.
7. T2 resumes execution, completes *delayQ.P*(), and then completes its *s.P()* operation.

In step (4), T4 executes *s.V()* and is blocked by the *delayQ.V()* operation.

3.6 Semaphores and Locks in Java: see Chapter3NotesJava

3.7 Semaphores and Locks in Win32: see Chapter3NotesWin32

3.8 Semaphores and Locks in Pthreads: see Chapter3NotesPthreads

3.9 Another Note on Shared Memory Consistency

Recall from Section 2.5.6 the issues surrounding shared memory consistency.

Compiler and hardware optimizations may reorder read and write operations on shared variables making it difficult to reason about the behavior of multithreaded programs.

Critical sections created using Java's built-in synchronization operations or the operations in the Win32 or Pthreads library provide mutual exclusion and also protect against unwanted re-orderings.

For example, the shared variable values that a thread can see when it unlocks a mutex can also be seen by any thread that later locks the *same* mutex.

Thus, an execution in which shared variables are correctly protected by locks or semaphores is guaranteed to be sequentially consistent

⇒ Rule: always access shared variables inside critical sections.

We will see that this rule also simplifies testing and debugging.

4. Monitors

Problems with semaphores:

- shared variables and the semaphores that protect them are global variables
- Operations on shared variables and semaphores distributed throughout program
- difficult to determine how a semaphore is being used (mutual exclusion or condition synchronization) without examining all of the code.

The monitor concept was developed by Tony Hoare and Per Brinch Hansen in the early ‘70’s to overcome these problems. (Same time period in which the concept of information hiding [Parnas 1972] and the class construct [Dahl et al. 1970] originated.)

Monitors support data encapsulation and information hiding and are easily adapted to an object-oriented environment.

4.1 Definition of Monitors

A monitor encapsulates shared data, all the operations on the data, and any synchronization required for accessing the data.

Object-oriented definition: a monitor is a synchronization object that is an instance of a special *monitor* class.

- A monitor class defines private variables and public and private access methods.
- The variables of a monitor represent shared data.
- Threads communicate by calling monitor methods that access shared variables.

4.1.1 Mutual Exclusion

At most one thread is allowed to execute inside a monitor at any time.

- Mutual exclusion is automatically provided by the monitor's implementation.
- If a thread calls a monitor method, but another thread is already executing inside the monitor, the calling thread must wait outside the monitor.
- A monitor has an entry queue to hold the calling threads that are waiting to enter the monitor.

4.1.2 Condition Variables and SC Signaling

Condition synchronization is achieved using condition variables and operations *wait()* and *signal()*.

A condition variable *cv* is declared as

```
conditionVariable cv;
```

- Operation *cv.wait()* is used to block a thread (analogous to a *P* operation).
- Operation *cv.signal()* unblocks a thread (analogous to a *V* operation).

A monitor has one entry queue plus one queue associated with each condition variable. For example, Listing 4.1 shows the structure of monitor class *boundedBuffer*. Class *boundedBuffer* inherits from class *monitor*. It has five data members, condition variables named *notFull* and *notEmpty*, and monitor methods *deposit()* and *withdraw()*.

Fig. 4.2 is a graphical view of class *boundedBuffer*, which shows its entry queue and the queues associated with condition variables *notFull* and *notEmpty*.

```
class boundedBuffer extends monitor {
   public void deposit(…) { … }
   public int withdraw (…) { … }
   public boundedBuffer( ) { … }

   private int fullSlots = 0;  // # of full slots in the buffer
   private int capacity = 0;  // capacity of the buffer
   private int [] buffer = null;  // circular buffer of ints
   // in is index for next deposit, out is index for next withdrawal
   private int in = 0, out = 0;
   // producer waits on notFull when the buffer is full
   private conditionVariable notFull;
   // consumer waits on notEmpty when the buffer is empty
   private conditionVariable notEmpty;
}
```
Listing 4.1 Monitor class *boundedBuffer*.

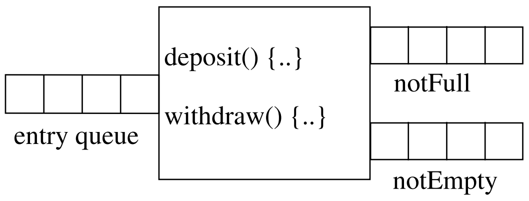


Figure 4.2 Graphical view of monitor class *boundedBuffer*.

A thread that is executing inside a monitor method blocks itself on condition variable *cv* by executing cv.wait():

- releases mutual exclusion (to allow another thread to enter the monitor)
- blocks the thread on the rear of the queue for *cv*.

A thread blocked on condition variable *cv* is awakened by cv.signal();

- If there are no threads blocked on *cv*, *signal()* has no effect; otherwise, *signal()* awakens the thread at the front of the queue for *cv*.
- For now, we will assume that the "signal-and-continue" (SC) discipline is used. After a thread executes an SC signal to awaken a waiting thread, the signaling thread continues executing in the monitor and the awakened thread is moved to the entry queue; *the awakened thread does not reenter the monitor immediately*.

A: denotes the set of threads that have been awakened by *signal()* operations and are waiting to reenter the monitor,

S: denotes the set of signaling threads,

C: denotes the set of threads that have called a monitor method but have not yet entered the monitor. (The threads in sets A and C wait in the entry queue.)

=> The relative priority associated with these three sets of threads is S > C = A.

cv.signalAll() wakes up all the threads that are blocked on condition variable *cv*.

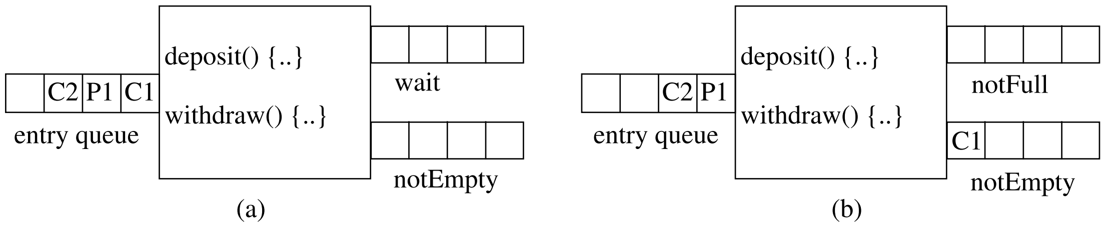
cv.empty() returns *true* if the queue for *cv* is empty, and *false* otherwise.

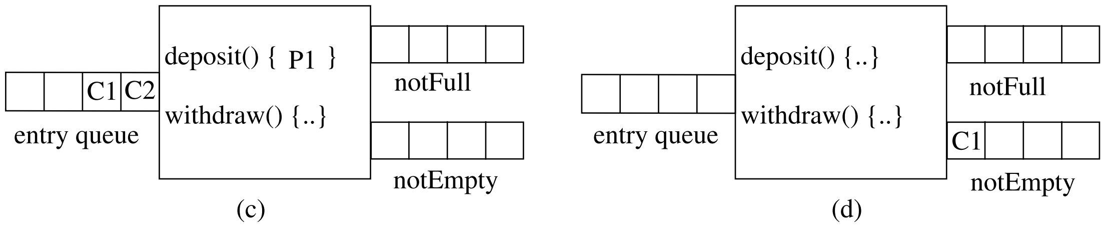
cv.length() returns the current length of the queue for *cv*.

Listing 4.3 shows a complete *boundedBuffer* monitor.

```
class boundedBuffer extends monitor {
    private  int fullSlots = 0;  // number of full slots in the buffer
    private int capacity = 0;   // capacity of the buffer
    private int[] buffer = null;    // circular buffer of ints
    private int in = 0, out = 0;
    private  conditionVariable notFull = new conditionVariable();
    private  conditionVariable notEmpty = new conditionVariable();
    public boundedBuffer(int bufferCapacity ) {
       capacity = bufferCapacity;buffer = new int[bufferCapacity];}
       public void deposit(int value) {
       while (fullSlots == capacity)
          notFull.wait();
       buffer[in] = value; in = (in + 1) % capacity;     ++fullSlots;
       notEmpty.signal();     //alternatively:if (fullSlots == 1) notEmpty.signal();
    }
    public int withdraw() {
       int value;
       while (fullSlots == 0)
          notEmpty.wait();
       value = buffer[out]; out = (out + 1) % capacity; --fullSlots;
       notFull.signal();      //alternatively:if (fullSlots == capacity–1) notFull.signal();
       return value;
    }
}
```

Listing 4.3 Monitor class *boundedBuffer*.





Assume that the buffer is empty and that the thread at the front of the entry queue is Consumer$_1$ (C$_1$). The queues for condition variables *notFull* and *notEmpty* are also assumed to be empty (Fig. 4.4a).

When Consumer$_1$ enters method *withdraw()*, it executes the statement

```
while (fullSlots == 0)
    notEmpty.wait();
```

Since the buffer is empty, Consumer$_1$ blocks itself by executing a *wait()* operation on condition variable *notEmpty* (Fig. 4.4b)

Producer$_1$ (P$_1$) then enters the monitor. Since the buffer is not full, Producer$_1$ deposits an item and executes *notEmpty.signal()*.

This signal operation awakens Consumer$_1$ and moves Consumer$_1$ to the rear of the entry queue behind Consumer$_2$ (C$_2$) (Fig. 4.4c).

After its *signal()* operation, Producer$_1$ can continue executing in the monitor, but since there are no more statements to execute, Producer$_1$ exits the monitor.

Consumer$_2$ now barges ahead of Consumer$_1$ and consumes an item. Consumer$_2$ executes *notFull.signal()*, but there are no Producers waiting so the signal has no effect.

When Consumer$_2$ exits the monitor, Consumer$_1$ is allowed to reenter, but the loop condition (*fullSlots* == *0*) is true again:

```
while (fullSlots == 0)
    notEmpty.wait();
```

Thus, Consumer$_1$ is blocked once more on condition variable *notEmpty* (Fig4.4d). Even though Consumer$_1$ entered the monitor first, it is Consumer$_2$ that consumes the first item.

This example illustrates why the *wait()* operations in an SC monitor are usually found inside while-loops: A thread waiting on a condition variable cannot assume that the condition it is waiting for will be true when it reenters the monitor.

4.2 Monitor-Based Solutions to Concurrent Programming Problems

These solutions assume that condition variable queues are First-Come-First-Serve.

4.2.1 Simulating Counting Semaphores

4.2.1.1 Solution 1. Listing 4.5 shows an SC monitor with methods *P()* and *V()* that simulates a counting semaphore. In this implementation, a waiting thread may get stuck forever in the while-loop in method *P()*:

- assume that the value of *permits* is 0 when thread T1 calls *P()*. Since the loop condition (*permits == 0*) is true, T1 will block itself by executing a *wait* operation.
- assume some other thread executes *V()* and signals T1. Thread T1 will join the entry queue behind threads that have called *P*() and are waiting to enter for the first time.
- These other threads can enter the monitor and decrement *permits* before T1 has a chance to reenter the monitor and examine its loop condition. If the value of *permits* is 0 when T1 eventually evaluates its loop condition, T1 will block itself again by issuing another *wait* operation.

```
class countingSemaphore1 extends monitor {
   private int permits; // The value of permits is never negative.
   private conditionVariable permitAvailable = new conditionVariable();
   public countingSemaphore1(int initialPermits) { permits = initialPermits;}
   public void P()  {
      while (permits == 0)
         permitAvailable.wait();
      --permits;
   }
   public void V() {
      ++permits;
      permitAvailable.signal();
   }
}
```

Listing 4.5 Class *countingSemaphore1*.

4.2.1.2 Solution 2. The SC monitor in Listing 4.6 does not suffer from a starvation problem. Threads that call *P*() cannot barge ahead of signaled threads and "steal" their permits.

Consider the scenario that we described in Solution 1:

- If thread T1 calls *P()* when the value of *permits* is 0, T1 will decrement *permits* to –1 and block itself by executing a *wait* operation.
- When some other thread executes *V()*, it will increment *permits* to 0 and signal T1. Threads ahead of T1 in the entry queue can enter the monitor and decrement *permits* before T1 is allowed to reenter the monitor.
- However, these threads will block themselves on the *wait* operation in P(), since *permits* will have a negative value.
- Thread T1 will eventually be allowed to reenter the monitor. Since there are no statements after the *wait* operation, T1 will complete its *P()* operation.

⇒ In this solution, a waiting thread that is signaled is guaranteed to get a permit.

```
class countingSemaphore2 extends monitor {
   private int permits;  // The value of permits may be negative.
   private conditionVariable permitAvailable = new conditionVariable();
   public countingSemaphore2(int initialPermits) { permits = initialPermits;}
   public void P()  {
      --permits;
      if (permits < 0)
         permitAvailable.wait();
   }
   public void V() {
      ++permits;
      permitAvailable.signal();
   }
}
```

Listing 4.6 Class *countingSemaphore2*.

4.2.2 Simulating Binary Semaphores

In the SC monitor in Listing 4.7, Threads in *P()* wait on condition variable *allowP* while threads in *V()* wait on condition variable *allowV*. Waiting threads may get stuck forever in the while-loops in methods *P()* and *V()*.

```
class binarySemaphore extends monitor {
   private int permits;
   private conditionVariable allowP = new conditionVariable();
   private conditionVariable allowV = new conditionVariable();
   public binarySemaphore(int initialPermits) { permits = initialPermits;}
   public void P() {
      while (permits == 0)
         allowP.wait();
      permits = 0;
      allowV.signal();
   }
   public void V() {
      while (permits == 1)
         allowV.wait();
      permits = 1;
      allowP.signal();
   }
}
```

Listing 4.7 Class *binarySemaphore*.

4.2.3 Dining Philosophers

4.2.3.1 Solution 1. In the SC monitor in Listing 4.8, a philosopher picks up two chopsticks only if both of them are available. Each philosopher has three possible states: thinking, hungry and eating:

- A hungry philosopher can eat if her two neighbors are not eating.
- A philosopher blocks herself on a condition variable if she is hungry but unable to eat.
- After eating, a philosopher will unblock a hungry neighbor who is able to eat.

This solution is deadlock-free, but not starvation-free, since a philosopher can starve if one of its neighbors is always eating

However, the chance of a philosopher starving may be so highly unlikely that perhaps it can be safely ignored?

4.2.3.2 Solution 2. In Listing 4.9, each philosopher has an additional state called "starving":

- A hungry philosopher is not allowed to eat if she has a starving neighbor, even if both chopsticks are available.
- Two neighboring philosophers are not allowed to be starving at the same time.

⇒ a hungry philosopher enters the "starving" state if she cannot eat and her two neighbors are not starving.

This solution avoids starvation. If there are five philosophers, then no more than four philosophers can eat before a given hungry philosopher is allowed to eat. However, some philosophers may not be allowed to eat even when both chopsticks are available.

Compared to Solution 1, this solution limits the maximum time that a philosopher can be hungry, but it can also increase the average time that philosophers are hungry.

```
class diningPhilosopher1 extends monitor {
   final int n = …;        // number of philosophers
   final int thinking = 0; final int hungry = 1; final int eating = 2;
   int state[] = new int[n];    // state[i] indicates the state of philosopher i
   // philosopher i blocks herself on self[i] when she is hungry but unable to eat
   conditionVariable[] self  = new conditionVariable[n];
   diningPhilosopher1() {
   for (int i = 0; i < n; i++) state[i] = thinking;
   for (int j = 0; j < n; j++) self[j] = new conditionVariable( );
   }
   public void pickUp(int i) {
      state[i] = hungry;
      test(i);         // change state to eating if philosopher i is able to eat
      if (state[i] != eating)
      self[i].wait();
   }
   public void putDown(int i) {
      state[i] = thinking;
      test((i-1) % n);     // check the left neighbor
      test((i+1) % n);    // check the right neighbor
   }
   private void test(int k) {
   // if philosopher k is hungry and can eat, change her state and signal her queue.
      if (( state[k] == hungry) && (state[(k+n-1) %  n] != eating ) &&
            (state[(k+1) %  n] != eating )) {
         state[k] = eating;
         self[k].signal(); // no affect if philosopher k is not waiting on self[k]
      }
   }
}

Philosopher i executes:
while (true) {
   /* thinking  */
   dp1.pickUp(i);
   /* eating */
   dp1.putDown(i)
}
```

Listing 4.8 Class *diningPhilosopher1*.

```
class diningPhilosopher2 extends monitor {
   final int n = …;       // number of philosophers
   final int thinking = 0; final int hungry = 1;
   final int starving = 2; final int eating = 3;
   int state[] = new int[n];   // state[i] indicates the state of philosopher i
   // philosopher i blocks herself on self[i] when she is hungry, but unable to eat
   conditionVariable[] self  = new conditionVariable[n];
   diningPhilosopher2() {
      for (int i = 0; i < n; i++) state[i] = thinking;
      for (int j = 0; j < n; j++) self[j] = new conditionVariable( );
   }
   public void pickUp(int i) {
      state[i] = hungry;
      test(i);
      if (state[i] != eating)
          self[i].wait();
   }
   public void  putDown(int i) {
      state[i] = thinking;
      test((i-1) % n);
      test((i+1) % n);
   }
   private void test(int k) {
   // Determine whether the state of philosopher k should be changed to
   // eating or starving. A hungry philosopher is not allowed to eat if she has a
   // neighbor that's starving or eating.
      if (( state[k] == hungry || state[k] == starving ) &&
          (state[(k+n-1) % n] != eating  && state[(k+n-1) % n] != starving ) &&
          (state[(k+1) % n] != eating  && state[(k+1) % n] !=starving)) {
              state[k] = eating;
              self[k].signal();   // no effect if phil. k is not waiting on self[k],
      }                    // which is the case if test() was called from pickUp().
      // a hungry philosopher enters the "starving" state if she cannot eat and her
      // neighbors are not starving
      else if   ((state[k] == hungry) && (state[(k+n-1) % n] != starving ) &&
                 (state[(k+1) % n] != starving )) {
          state[k] = starving;
      }
   }
}
```

Listing 4.9 Class *diningPhilosopher2*.

4.2.4 Readers and Writers

Listing 4.10 is an SC monitor implementation of strategy R>W.1, which allows concurrent reading and gives readers a higher priority than writers (see Section 3.5.4.) Reader and writer threads have the following form:

```
r_gt_w.1 rw;

Reader Threads:              Writer Threads:
  rw.startRead();              rw.startWrite();
  /* read shared data */       /* write to shared data */
  rw.endRead();                rw.endWrite();
```

Writers are forced to wait in method *startWrite()* if any writers are writing or any readers are reading or waiting.

In method *endWrite()*, all the waiting readers are signaled since readers have priority.

However, one or more writers may enter method *startWrite()* before the signaled readers reenter the monitor. Variable *signaledReaders* is used to prevent these barging writers from writing when the signaled readers are waiting in the entry queue and no more readers are waiting in *readerQ*.

Notice above that the shared data is read outside the monitor. This is necessary in order to allow concurrent reading.

```
class r_gt_w_1 extends monitor {
   int readerCount = 0;     // number of active readers
   boolean writing = false;   // true if a writer is writing
   conditionVariable readerQ = new conditionVariable();
   conditionVariable writerQ = new conditionVariable();
   int signaledReaders = 0;  // number of readers signaled in endWrite
   public void startRead() {
      if (writing) {             // readers must wait if a writer is writing
         readerQ.wait();
         --signaledReaders;// another signaled reader has started reading
      }
      ++readerCount;
   }
   public void endRead() {
      --readerCount;
      if (readerCount == 0 && signaledReaders==0)
         // signal writer if no more readers are reading and the signaledReaders
         //    have  read
         writerQ.signal();
   }
   public void startWrite() {
   // the writer waits if another writer is writing, or a reader is reading or waiting,
   // or the writer is barging
      while (readerCount > 0 || writing || !readerQ.empty() || signaledReaders>0)
          writerQ.wait();
      writing = true;
   }
   public void endWrite() {
      writing = false;
      if (!readerQ.empty()) { // priority is given to waiting readers
          signaledReaders = readerQ.length();
          readerQ.signalAll();
      }
      else writerQ.signal();
   }
}
```

Listing 4.10 Class *r_gt_w_1* allows concurrent reading and gives readers a higher priority than writers.

4.3 Monitors in Java

Java's wait, notify, and notifyAll operations combined with synchronized methods and user-defined classes enables the construction of objects that have some of the characteristics of monitors.

Adding synchronized to the methods of a Java class automatically provides mutual exclusion for threads accessing the data members of an instance of this class.

However, if some or all of the methods are inadvertently not synchronized, a data race may result. This enables the very types of bugs that monitors were designed to eliminate!

There are no explicit condition variables in Java. When a thread executes a wait operation, it can be viewed as waiting on a single, implicit condition variable associated with the object.

Operations wait, notify, and notifyAll use SC signaling:

- A thread must hold an object's lock before it can execute a wait, notify, or notifyAll operation. Thus, these operations must appear in a synchronized method or synchronized block (see below); otherwise, an *IllegalMonitorStateException* is thrown.

- Every Java object has a lock associated with it. Methods wait, notify, and notifyAll are inherited from class *Object*, the base class for all Java objects.

- When a thread executes wait, it releases the object's lock and waits in the "wait set" that is associated with the object:
  - A notify operation awakens a single waiting thread in the wait set.
  - A notifyAll operation awakens all the waiting threads.
  - Operations notify and notifyAll are not guaranteed to wake up the thread that has been waiting the longest.

- A waiting thread T may be removed from the wait set due to any one of the following actions: a notify or notifyAll operation; an interrupt action being performed on T; a timeout for a timed wait (e.g., wait(1000) allows T to stop waiting after one second); or a “spurious wakeup”, which removes T without any explicit Java instructions to do so (Huh?!).

- A notified thread must reacquire the object’s lock before it can begin executing in the method. Furthermore, notified threads that are trying to reacquire an object’s lock compete with any threads that have called a method of the object and are trying to acquire the lock for the first time. The order in which these notified and calling threads obtain the lock is unpredictable.

- If a waiting thread T is interrupted at the same time that a notification occurs, then the result depends on which version of Java is being used:

In versions before J2SE 5.0 [JSR-133 2004], the notification may be “lost”.

- Suppose that thread T and several other threads are in the wait set and thread T is notified and then interrupted before it reacquires the monitor lock. Then the wait operation that was executed by T throws *InterruptedException* and the notification gets lost, i.e., no waiting thread is allowed to proceed.
- Thus, it is recommended that the catch block for *InterruptedException* execute an additional notify or notifyAll to make up for any lost notifications [Hartley 1999]. Alternately, the programmer should use notifyAll instead of notify to wake up all waiting threads even when just one thread can logically proceed.

J2SE 5.0 removes the possibility of lost notifications. If the interrupt of T occurs before T is notified then T’s interrupt status is set to false, wait throws *InterruptedException*, and some other waiting thread (if any exist at the time of the notification) receives the notification. If the notification occurs first, then T eventually returns normally from the wait with its interrupt status set to true.

(A thread can determine its interrupt status by invoking the static method *Thread.isInterrupted(*), and it can observe and clear its interrupt status by invoking the static method *Thread.interrupted()*.)

4.3.1 A Better *countingSemaphore*

Class *countingSemaphore* in Listing 4.11 has been revised to handle interrupts and spurious wakeups. It assumes that J2SE 5.0 interrupt semantics are used, which prevents notifications from being lost when a waiting thread is interrupted right before it is notified.

```
public final class countingSemaphore {
   private int permits = 0; int waitCount = 0; int notifyCount=0;
   public countingSemaphore(int initialPermits) {
      if (initialPermits>0) permits = initialPermits;
   synchronized public void P() throws InterruptedException {
      if (permits <= waitCount) {
         waitCount++;          // one more thread is waiting
         try {
            do { wait(); }     // spurious wakeups do not increment
            while (notifyCount == 0);//   notifyCount
         }
         finally { waitCount--; }   // one waiting thread notified or interrupted
         notifyCount--;    // one notification has been consumed
      }
      else {
         if (notifyCount > waitCount)   // if some notified threads were
            notifyCount--;      //     interrupted, adjust notifyCount
      }
      permits--;
   }
   synchronized public void V() {
      permits++;
      if (waitCount > notifyCount) { // if there are waiting threads yet to be
         notifyCount++;     //    notified, notify one thread
         notify();
      }
   }
}
```

Listing 4.11 Java Class *countingSemaphore* that handles interrupts and spurious wakeups.

Variable *notifyCount* counts notifications that are made in method V():

- The if-statement in method *V()* ensures that only as many notifications are done as there are waiting threads.
- A thread that awakens from a wait operation in *P()* executes wait again if *notifyCount* is zero since a spurious wakeup must have occurred (i.e., a notification must have been issued outside of *V()*.)

Assume that two threads are blocked on the wait operation in method *P()* and the value of *permits* is 0.

- Suppose that three *V()* operations are performed and all three *V()* operations are completed before either of the two notified threads can reacquire the lock.
- Then the value of *permits* is 3 and the value of *waitCount* is still 2.
- A thread that then calls *P()* and barges ahead of the notified threads is not required to wait and does not execute wait since the condition (*permits* <= *waitCount*) in the if-statement in method *P()* is false.

Assume that two threads are blocked on the wait operation in method *P()*

- Suppose two *V()* operations are executed: *notifyCount* and *permits* become 2.
- Suppose further that the two notified threads are interrupted so that *waitCount* becomes 0 (due to the interrupted threads decrementing *waitCount* in their finally blocks).
- If three threads now call method *P()*, two of the threads will be allowed to complete their *P()* operations, and before they complete *P()*, they will each decrement *notifyCount* since both will find that the condition *(notifyCount* > *waitCount*) is true.
- Now *permits*, *notifyCount* and *waitCount* are all 0.
- If another thread calls *P()* it will be blocked by the wait operation in *P()*, and if it is awakened by a spurious wakeup, it will execute wait again since *notifyCount* is zero.

Class *Semaphore* in J2SE 5.0 package *java.util.concurrent* provides methods *acquire()* and *release()* instead of *P()* and *V()*, respectively. The implementation *of acquire()* handles interrupts as follows.

- If a thread calling *acquire()* has its interrupted status set on entry to *acquire()*, or is interrupted while waiting for a permit, then InterruptedException is thrown and the calling thread's interrupted status is cleared.
- Any permits that were to be assigned to this thread are instead assigned to other threads trying to acquire permits.

Class *Semaphore* also provides method *acquireUninterruptibly()*:

- If a thread calling *acquireUninterruptibly()* is interrupted while waiting for a permit then it will continue to wait until it receives a permit.
- When the thread does return from this method its interrupt status will be set.

4.3.2 notify vs. notifyAll

A Java monitor only has a single (implicit) condition variable available to it so notify operations must be handled carefully.

Listing 4.12 shows Java class *binarySemaphore*.

- Threads that are blocked in *P*() or *V*() are waiting in the single queue associated with the implicit condition variable.
- Any execution of notifyAll awakens all the waiting threads, even though one whole group of threads, either those waiting in *P*() or those waiting in *V*(), cannot possibly continue.
- The first thread, if any, to find that its loop condition is false, exits the loop and completes its operation. The other threads may all end up blocking again.

If notify were used instead of notifyAll, the single thread that was awakened might be a member of the wrong group. If so, the notified thread would execute another wait operation and the notify operation would be lost, potentially causing deadlock.

Use notifyAll instead of notify unless the following requirements are met:

1. all waiting threads are waiting on conditions that are signaled by the same notifications. If one condition is signaled by a notification, then the other conditions are also signaled by this notification. Usually, when this requirement is met, all the waiting threads are waiting on the exact same condition.
2. each notification is intended to enable exactly one thread to continue. (In this case, it would be useless to wake up more than one thread.)

Example: In class *countingSemaphore* in Listing 3.15, all threads waiting in *P*() are waiting for the same condition (*permits* ≥ 0), which is signaled by the notify operation in *V*(). Also, a notify operation enables one waiting thread to continue.

Even though both of these requirements might be satisfied in a given class, they may not be satisfied in subclasses of this class, so using notifyAll may be safer. (Use the `final` keyword to prevent subclassing.)

Monitor *boundedBuffer* in Listing 4.13 uses notifyAll operations since *Producers* and *Consumers* wait on the same implicit condition variable and they wait for different conditions that are signaled by different notifications.

```
final class boundedBuffer {
   private int fullSlots=0; private int capacity = 0;
   private int[] buffer = null; private int in = 0, out = 0;
   public boundedBuffer(int bufferCapacity) {
      capacity = bufferCapacity; buffer = new int[capacity];
   }
   public synchronized void deposit (int value) {
      while (fullSlots == capacity) // assume no interrupts are possible
         try { wait(); } catch (InterruptedException ex) {}
      buffer[in] = value; in = (in + 1) % capacity;
      if (fullSlots++ == 0) // note the use of post-increment.
         notifyAll();   // it is possible that Consumers are waiting for "not empty"
   }
   public synchronized int withdraw () {
      int value = 0;
      while (fullSlots == 0)   // assume no interrupts are possible
         try { wait(); } catch (InterruptedException ex) {}
      value = buffer[out];  out = (out + 1) % capacity;
      if (fullSlots-- == capacity)   // note the use of post-decrement.
         notifyAll(); // it is possible that Producers are waiting for "not full"
      return value;
   }
}
```

Listing 4.13 Java monitor *boundedBuffer*.

4.3.3 Simulating Multiple Condition Variables

It is possible to use simple Java objects to achieve an effect that is similar to the use of multiple condition variables.

Listing 4.14 shows a new version of class *binarySemaphore* that uses objects *allowP* and *allowV* in the same way that condition variables are used.

Notice that methods *P*() and *V*() are not synchronized since it is objects *allowP* and *allowV* that must be synchronized in order to perform wait and notify operations on them. (Adding synchronized to methods *P()* and *V()* would synchronize the *binarySemaphore2* object, not objects *allowP* and *allowV*, and would result in a deadlock.)

The use of a synchronized block:

```
synchronized (allowP) {
   /* block of code */
}
```

creates a block of code that is synchronized on object *allowP*.

- A thread must acquire *allowP's* lock before it can enter the block.
- The lock is released when the thread exits the block.

Note that a synchronized method:

```
public synchronized void F() {
   /* body of F */
}
```

is equivalent to a method whose body consists of a single synchronized block:

```
public void F() {
   synchronized(this) {
      /* body of F */
   }
}
```

```
public final class binarySemaphore2 {
   int vPermits = 0;   int pPermits = 0;
   Object allowP = null;    // queue of threads waiting in P()
   Object allowV = null;   // queue of threads waiting in V()
   public binarySemaphore2(int initialPermits) {
      if (initialPermits != 0 && initialPermits != 1) throw new
       IllegalArgumentException("initial binary semaphore value must be 0 or 1");
      pPermits = initialPermits;  // 1 or 0
      vPermits = 1 – pPermits;   // 0 or 1
      allowP = new Object(); allowV = new Object();
   }
   public void P() {
      synchronized (allowP) {
         --pPermits;
         if (pPermits < 0) // assume no interrupts are possible
            try { allowP.wait(); } catch (InterruptedException e) {}
      }
      synchronized (allowV) {
         ++vPermits;
         if (vPermits <=0)
            allowV.notify(); // signal thread waiting in V()
      }
   }
    public void V() {
       synchronized (allowV) {
         --vPermits;
         if (vPermits < 0) // assume no interrupts are possible
            try { allowV.wait(); } catch (InterruptedException e) {}
       }
       synchronized (allowP) {
          ++pPermits;
          if (pPermits <= 0)
            allowP.notify(); // signal thread waiting in P()
       }
   }
}
```

Listing 4.14 Java class *binarySemaphore2*.

4.4 Monitors in Pthreads

Pthreads does not provide a monitor construct, but it provides condition variables, which enables the construction of monitor-like objects.

4.4.1 Pthreads Condition Variables

The operations that can be performed on a Pthreads condition variable are *pthread_cond_wait()*, *pthread_cond_signal()*, and *pthread_cond_broadcast()*.

The *signal* and *broadcast* operations are similar to Java's notify and notifyAll operations, respectively.

Also like Java, *wait* operations are expected to be executed inside a critical section. Thus, a condition variable is associated with a *mutex* and this *mutex* is specified when a *wait* operation is performed.

- A condition variable is initialized by calling *pthread_cond_init*().

- When a thread waits on a condition variable it must have the associated *mutex* locked. This means that a *wait* operation on a condition variable must be preceded by a lock operation on the *mutex* associated with the condition variable.

- Each condition variable must be associated at any given time with only one mutex. On the other hand, a mutex may have any number of condition variables associated with it.

- A *wait* operation automatically unlocks the associated *mutex* if the calling thread is blocked and automatically tries to lock the associated *mutex* when the blocked thread is awakened by a *signal* or *broadcast* operation. The awakened thread competes with other threads for the *mutex*.

- A *signal* operation wakes up a single thread while a *broadcast* operation wakes up all waiting threads.

- A thread can *signal* or *broadcast* a condition variable without holding the lock for the associated *mutex*. (This is different from Java since a Java *notify* or *notifyAll* operation must be performed in a synchronized method or block.) A thread that executes a *signal* or *broadcast* continues to execute. If the thread holds the lock for the associated *mutex* when it performs a *signal* or *broadcast*, it should eventually release the lock.

Listing 4.15 shows a C++/Pthreads monitor class named *boundedBuffer* that solves the bounded buffer problem.

Producers wait on condition variable *notFull* and Consumers wait on condition variable *notEmpty*. The use of two explicit condition variables makes this Pthreads version similar to the solution in Listing 4.3.

Note that the *wait* operations appear in loops since Pthreads uses the SC signaling discipline.

Also, it is recommended to always use loops because of the possibility of "spurious wakeups", i.e., threads can be awakened without any *signal* or *broadcast* operations being performed.

```
#include <iostream>
#include <pthread.h>
#include "thread.h"
class boundedBuffer  {
private:
    int fullSlots;                  // # of full slots in the buffer
    int capacity; int* buffer; int in, out;
    pthread_cond_t notFull;  // Producers wait on notFull
    pthread_cond_t notEmpty;  // Consumers wait on notEmpty
    pthread_mutex_t mutex;  // exclusion for deposit() and withdraw()
public:
    boundedBuffer(int capacity_) : capacity(capacity_), fullSlots(0), in(0),
       out(0), buffer(new int[capacity_]) {
          pthread_cond_init(&notFull,NULL); pthread_cond_init(&notEmpty,NULL);
          pthread_mutex_init(&mutex,NULL);
    }
    ~boundedBuffer() {
       delete [] buffer;
       pthread_cond_destroy(&notFull); pthread_cond_destroy(&notEmpty);
       pthread_mutex_destroy(&mutex);
    }
    void deposit(int value) {
       pthread_mutex_lock(&mutex);
       while (fullSlots == capacity)
           pthread_cond_wait(&notFull,&mutex);
       buffer[in] = value;
       in = (in + 1) % capacity;   ++fullSlots;
       pthread_mutex_unlock(&mutex);
       pthread_cond_signal(&notEmpty);
    }
    int withdraw() {
       pthread_mutex_lock(&mutex);
       int value;
       while (fullSlots == 0)
           pthread_cond_wait(&notEmpty,&mutex);
       value = buffer[out];  out = (out + 1) % capacity;  --fullSlots;
       pthread_mutex_unlock(&mutex);
       pthread_cond_signal(&notFull);
       return value;
    }
};
```

```
class Producer : public Thread {
private:
    boundedBuffer& b;
    int num;
public:
    Producer (boundedBuffer* b_, int num_) : b(*b_),  num(num_) { }
    virtual void* run () {
       std::cout << "Producer Running" << std::endl;
       for (int i = 0; i < 3; i++) {
           b.deposit(i);
           std::cout << "Producer # " << num << " deposited " << i << std::endl;
       }
       return NULL;
    }
};

class Consumer : public Thread {
private:
    boundedBuffer& b;
    int num;
public:
    Consumer (boundedBuffer* b_, int num_) : b(*b_), num(num_) { }
    virtual void* run () {
       std::cout << "Consumer Running" << std::endl;
       int value = 0;
       for (int i = 0; i < 3; i++) {
           value = b.withdraw();
           std::cout << "Consumer # " << num << " withdrew " <<
             value << std::endl;
       }
       return NULL;
    }
};

int main ( ) {
    boundedBuffer* b1 = new boundedBuffer(3);
    Producer p1(b1, 1); Consumer c1(b1, 1);
    p1.start(); c1.start();
    p1.join(); c1.join();
    delete b1;
    return 0;
}
```
Listing 4.15 C++/Pthreads class *boundedBuffer*.

4.4.2 Condition Variables in J2SE 5.0

Package *java.util.concurrent.locks* in Java release J2SE 5.0, contains a lock class called *ReentrantLock* (see Section 3.6.4) and a condition variable class called *Condition*.

A *ReentrantLock* replaces the use of a synchronized method, and operations *await* and *signal* on a *Condition* replace the use of methods *wait* and *notify*.

A *Condition* object is bound to its associated *ReentrantLock* object. Method *newCondition()* is used to obtain a *Condition* object from a *ReentrantLock*:

```
ReentrantLock mutex = new ReentrantLock();
// notFull and notEmpty are bound to mutex
Condition notFull = mutex.newCondition();
Condition notEmpty = mutex.newCondition();
```

Conditions provide operations *await*, *signal*, and *signallAll*, including those with timeouts.

Listing 4.16 shows a Java version of class *boundedBuffer* using *Condition* objects.

- The try-finally clause ensures that *mutex* is unlocked no matter how the try block is executed.
- If an interrupt occurs before a *signal* then the *await* method must, after re-acquiring the lock, throw InterruptedException. (If the interrupted thread is signaled, then some other thread (if any exist at the time of the *signal*) receives the signal.)
- if the interrupted occurs after a *signal*, then the *await* method must return without throwing an exception, but with the current thread's interrupt status set.

```
import java.util.concurrent.locks;
final class boundedBuffer {
    private int fullSlots=0; private int capacity = 0; private int in = 0, out = 0;
    private int[] buffer = null;
    private ReentrantLock mutex;
    private Condition notFull;
    private Condition notEmpty;

    public boundedBuffer(int bufferCapacity) {
       capacity = bufferCapacity; buffer = new int[capacity];
       mutex = new ReentrantLock();
       // notFull and notEmpty are both attached to mutex
       notFull = mutex.newCondition(); notEmpty = mutex.newCondition();
    }
    public void deposit (int value) throws InterruptedException {
       mutex.lock();
       try {
          while (fullSlots == capacity)
             notFull.await();
          buffer[in] = value;
          in = (in + 1) % capacity;
          notEmpty.signal();
       } finally {mutex.unlock();}
    }
    public synchronized int withdraw () throws InterruptedException {
       mutex.lock();
       try {
          int value = 0;
          while (fullSlots == 0)
             notEmpty.await();
          value = buffer[out];
          out = (out + 1) % capacity;
          notFull.signal();
          return value;
       } finally {mutex.unlock();}
    }
}
```

Listing 4.16 Java class *boundedBuffer* using *Condition* objects.

4.5 Signaling Disciplines

4.5.1 Signal-and-Urgent-Wait (SU)

When a thread executes *cv.signal()*:

- if there are no threads waiting on condition variable *cv*, this operation has no effect.
- otherwise, the thread executing signal (which is called the "signaler" thread) awakens one thread waiting on *cv*, and blocks itself in a queue, called the reentry queue. The signaled thread reenters the monitor immediately.

When a thread executes *cv.wait()*:

- if the reentry queue is not empty, the thread awakens one signaler thread from the reentry queue and then blocks itself on the queue for *cv*.
- otherwise, the thread releases mutual exclusion (to allow a new thread to enter the monitor) and then blocks itself on the queue for *cv*.

When a thread completes and exits a monitor method:

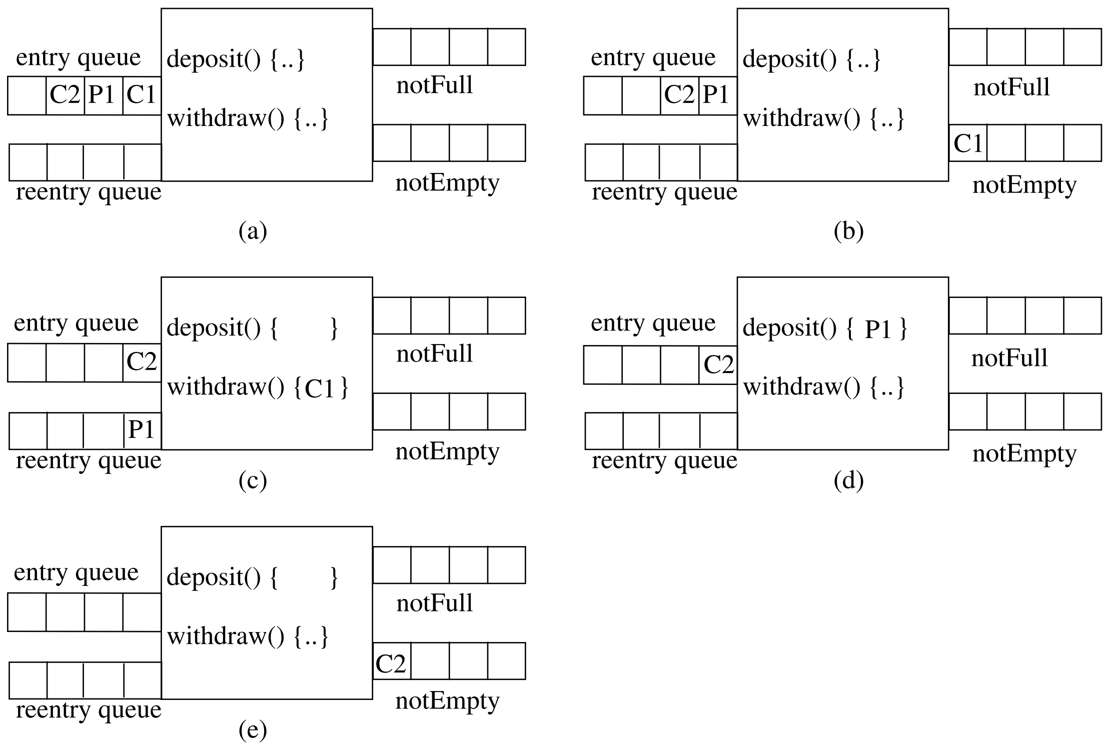
- if reentry queue is not empty, it awakens one signaler thread from the reentry queue.
- otherwise, it releases mutual exclusion to allow a new thread to enter the monitor.

In an SU monitor, the threads waiting to enter a monitor have three levels of priority (from highest to lowest):

- the awakened thread (A), which is the thread awakened by a signal operation
- signaler threads (S), which are the threads waiting in the reentry queue
- calling threads (C), which are the threads that have called a monitor method and are waiting in the entry queue.

The relative priority associated with the three sets of threads is A > S > C.

Considering again the *boundedBuffer* monitor in Listing 4.3. Assume that SU signals are used instead of SC signals. Assume we start with Fig. 4.17a.



When Consumer$_1$ enters method *withdraw()*, it executes the statement

```
while (fullSlots == 0)
    notEmpty.wait();
```

Since the buffer is empty, Consumer$_1$ is blocked by the *wait()* operation on condition variable *notEmpty* (Fig. 4.17b).

Producer$_1$ then enters the monitor:

- Since the buffer is not full, Producer$_1$ deposits its item, and executes *notEmpty.signal()*.
- This signal awakens Consumer$_1$ and moves Producer$_1$ to the Reentry queue (Fig. 4.17c).

Consumer$_1$ can now consume an item.

- When Consumer$_1$ executes *notFull.signal()*, there are no Producers waiting so none are signaled and Consumer$_1$ does not move to the Reentry queue.
- When Consumer$_1$ exits the monitor, Producer$_1$ is allowed to reenter the monitor since the Reentry queue has priority over the Entry queue (Fig. 4.17d).

Producer$_1$ has no more statements to execute so Producer$_1$ exits the monitor.

- Since the Reentry queue is empty, Consumer$_2$ is now allowed to enter the monitor.
- Consumer$_2$ finds that the buffer is empty and blocks itself on condition variable *notEmpty* (Fig. 4.17e).

Unlike the scenario that occurred when SC signals were used, Consumer$_2$ was not allowed to barge ahead of Consumer$_1$ and consume the first item.

Since signaled threads have priority over new threads, a thread waiting on a condition variable in an SU monitor can assume that the condition it is waiting for will be true when it reenters the monitor.

In the *boundedBuffer* monitor, we can replace the while-loops with if-statements and avoid the unnecessary reevaluation of the loop condition after a *wait()* operation returns.

As another example, Listing 4.18 shows an SU monitor implementation of strategy R>W.1. This implementation is simpler than the SC monitor in Section 4.2.4 since there is no threat of barging - when a writer signals waiting readers, these waiting readers are guaranteed to reenter the monitor before any new writers are allowed to enter *startWrite()*.

```
class r_gt_w_1SU extends monitorSU {
   int readerCount = 0;// number of active readers
   boolean writing = false;   // true if a writer is writing
   conditionVariable readerQ = new conditionVariable();
   conditionVariable writerQ = new conditionVariable();

   public void startRead() {
      if (writing)          // readers must wait if a writer is writing
         readerQ.wait();
      ++readerCount;
      readerQ.signal();  // continue cascaded wakeup of readers
   }
   public void endRead() {
      --readerCount;
      if (readerCount == 0)
         writerQ.signal();   // signal writer if there are no more readers reading
   }
   public void startWrite() {
   // writers wait if a writer is writing, or a reader is reading or waiting, or the
   // writer is barging
      while (readerCount > 0 || writing || !readerQ.empty())
          writerQ.wait();
      writing = true;
   }
   public void endWrite() {
      writing = false;
      if (!readerQ.empty()) { // priority is given to waiting readers
          readerQ.signal();  // start cascaded wakeup of readers
      }
      else
          writerQ.signal();
   }
}
```

Listing 4.18 Class *r_gt_w_1SU* allows concurrent reading and gives readers a higher priority than writers.

4.5.2 Signal-and-Exit (SE)

Signal-and-Exit (SE) is a special case of Signal-and-Urgent-Wait (SU).

When a thread executes an SE *signal* operation it does not enter the reentry queue; rather, it exits the monitor immediately.

- An SE *signal* statement is either the last statement of a method or it is followed immediately by a return statement.
- As with SU signals, the thread awakened by a signal operation is always the next thread to enter the monitor.

Since there are no signaling threads that want to remain in or reenter the monitor, the relative priority of the sets of awakened (A) and calling (C) threads is A > C.

When a *signal* statement appears in a monitor method, it is very often the last statement of the method, regardless of the type of *signal*. This was the case for all of the SC monitor examples in Section 4.2 and for the SU monitor in Listing 4.18.

Using SE semantics for these special SU *signal* operations avoids the extra cost of having a thread exit a monitor and join the reentry queue, and then later reenter the monitor only to immediately exit the monitor again.

The Java and C++ monitor classes shown later have a Signal-and-Exit operation that allows SE *signals* to be used at the end of SU monitor methods.

4.5.3 Urgent-Signal-and-Continue (USC)

- A thread that executes an USC *signal* operation continues to execute just as it would for an SC signal.
- A thread awakened by a *signal* operation has priority over threads waiting in the entry queue.

When *signal* operations appear only at the end of monitor methods, which is usually the case, this discipline is the same as the SE discipline, which is a special case of the SU discipline.

A thread waiting in the entry queue is allowed to enter a monitor only when no other threads are inside the monitor and there are no signaled threads waiting to reenter.

Thus, the relative priority associated with the three sets of threads is S > A > C.

Table 4.1 lists the signaling disciplines and shows the relative priorities associated with the three sets of threads. Another signaling discipline is described in Exercise 4.17.

Relative priority	Name
S > A = C	Signal-and-Continue [Lampson and Redell 1980]
A > S > C	Signal-and-Urgent-Wait [Hoare 1974]
A > C	Signal-and-Exit [Brinch Hansen 1975]
S > A > C	Urgent-Signal-and-Continue [Howard 1976]

Table 4.1 Signaling Disciplines

4.5.4 Comparing SU and SC signals

If a thread executes an SU *signal* to notify another thread that a certain condition is true, this condition remains true when the signaled thread reenters the monitor.

A *signal* operation in an SC monitor is only a “hint” to the signaled thread that it may be able to proceed. Other threads may “barge” into the monitor and make the condition false before the signaled thread reenters the monitor.

That is why SC monitors use a while-loop instead of an if-statement.

- Using a while-loop instead of an if-statement in an SC monitor requires an extra evaluation of condition (*permits == 0*) after a *wait()*.
- On the other hand, the execution of an SU monitor requires additional context switches for managing the signaler threads in the Reentry queue.

Using Signal-and-Exit semantics for a signal operation that appears at the end of an SU method avoids the costs of the extra condition evaluation and the extra context switches.

4.6 Using Semaphores to Implement Monitors

Semaphores can be used to implement monitors with SC, SU, or SE signaling.

Even though Java provides built-in support for monitors, our custom Java monitor classes are still helpful since it is not easy to test and debug built-in Java monitors. Also, we can choose which type of monitor - SC, SU, SE, or USC - to use in our Java programs.

4.6.1 SC Signaling.

The body of each public monitor method is implemented as

```
public returnType F(…) {
   mutex.P();
   /* body of F */
   mutex.V();
}
```

Semaphore *mutex* is initialized to 1. The calls to *mutex.P()* and *mutex.V()* ensure that monitor methods are executed with mutual exclusion.

Java class *conditionVariable* in Listing 4.20 implements condition variables with SC signals:

- since it is not legal to overload final method *wait*() in Java, methods named *waitC*() and *signalC*() are used instead of *wait()* and *signal().*
- the *signalCall()* operation behaves the same as Java's notifyAll
- operations *empty()* and *length()* are also provided

```
final class conditionVariable {
   private countingSemaphore threadQueue = new countingSemaphore(0);
   private int numWaitingThreads = 0;
   public void waitC() {
     numWaitingThreads++;   // one more thread is waiting in the queue
     threadQueue.VP(mutex); // release exclusion and wait in threadQueue
     mutex.P();             // wait to reenter the monitor
   }
   public void signalC() {
     if (numWaitingThreads > 0) {  // if any threads are waiting
        numWaitingThreads--;//      // wakeup one thread in the queue
        threadQueue.V();
     }
   }
   public void signalCall() {
     while (numWaitingThreads > 0) {   // if any threads are waiting
        --numWaitingThreads;           //  wakeup all the threads in the queue
        threadQueue.V();               //    one-by-one
     }
   }
   // returns true if the queue is empty
   public boolean empty() { return (numWaitingThreads == 0); }
   // returns the length of the queue
   public int length() { return numWaitingThreads; }
}
```

Listing 4.20 Java class *conditionVariable*.

Each *conditionVariable* is implemented using a semaphore named *threadQueue*:

- When a thread executes *waitC()*, it releases mutual exclusion, and blocks itself using threadQueue.VP(mutex). VP() guarantees that threads executing *waitC()* are blocked on semaphore *threadQueue* in the same order that they entered the monitor.
- An integer variable named *numWaitingThreads* is used to count the waiting threads.

The value of *numWaitingThreads* is incremented in *waitC()* and decremented in *signalC()* and *signalCall()*:

- *signalC()* executes *threadQueue.V()* to signal one waiting thread.
- *signalCall()* uses a while-loop to signal all the waiting threads one-by-one.

4.6.2 SU Signaling.

Java class *conditionVariable* in Listing 4.21 implements condition variables with SU signals.

Each SU monitor has a semaphore named *reentry* (initialized to 0), on which signaling threads block themselves.

If *signalC()* is executed when threads are waiting on the condition variable, *reentry.VP(threadQueue)* is executed to signal a waiting thread, and block the signaler in the reentry queue.

Integer *reentryCount* is used to count the number of signaler threads waiting in *reentry*.

- When a thread executes *waitC*(), if signalers are waiting, the thread releases a signaler by executing *reentry.V()*; otherwise, the thread releases mutual exclusion by executing *mutex.V()*.
- Method *signalC()* increments and decrements *reentryCount* as threads enter and exit the *reentry* queue.

The body of each public monitor method is implemented as

```
public returnType F(…) {
  mutex.P();
  /* body of F */
  if (reentryCount >0)
     reentry.V();      // allow a signaler thread to reenter the monitor
  else mutex.V();      // allow a calling thread to enter the monitor
}
```

```
final class conditionVariable {
   private countingSemaphore threadQueue = new countingSemaphore(0);
   private int numWaitingThreads = 0;
   public void signalC() {
      if (numWaitingThreads > 0) {
         ++reentryCount;
         reentry.VP(threadQueue);  // release exclusion and join reentry queue
         --reentryCount;
      }
   }
   public void waitC() {
      numWaitingThreads++;
      if (reentryCount > 0) threadQueue.VP(reentry); // the reentry queue has
      else threadQueue.VP(mutex);       //   priority over entry queue
      --numWaitingThreads;
   }
   public boolean empty() { return (numWaitingThreads == 0); }
   public int length() { return numWaitingThreads; }
}
```

Listing 4.21 Java class *conditionVariable* for SU monitors.

4.7 A Monitor Toolbox for Java

A monitor toolbox is a program unit that is used to simulate the monitor construct. The Java monitor toolboxes are class *monitorSC* for SC monitors and class *monitorSU* for SU monitors.

Classes *monitorSC* and *monitorSU* implement operations *enterMonitor* and *exitMonitor*, and contain a member class named *conditionVariable* that implements *waitC* and *signalC* operations on condition variables.

A regular Java class can be made into a monitor class by doing the following:

1. extend class *monitorSC* or *monitorSU*
2. use operations *enterMonitor()* and *exitMonitor()* at the start and end of each public method
3. declare as many *conditionVariables* as needed
4. use operations *waitC()*, *signalC()*, *signalCall()*, *length()*, and *empty()*, on the *conditionVariables*.

Listing 4.22 shows part of a Java *boundedBuffer* class that illustrates the use of class *monitorSC*.

Simulated monitors are not as easy to use or as efficient as real monitors, but they have some advantages:

- A monitor toolbox can be used to simulate monitors in languages that do not support monitors directly, e.g., C++/Win32/Pthreads.
- Different versions of the toolbox can be created for different types of signals, e.g., an SU toolbox can be used to allow SU signaling in Java.
- The toolbox can be extended to support testing and debugging.

```
final class boundedBuffer extends monitorSC {
   …
   private conditionVariable notFull = new conditionVariable();
   private conditionVariable notEmpty = new conditionVariable();
   …
   public void deposit(int value) {
      enterMonitor();
      while (fullSlots == capacity)
         notFull.waitC();
      buffer[in] = value;
         in = (in + 1) % capacity;
      ++fullSlots;
      notEmpty.signalC();
      exitMonitor();
   }
   …
}
```

Listing 4.22 Using the Java monitor toolbox class *monitorSC*.

4.7.1 A Toolbox for SC Signaling in Java

Listing 4.23 shows a monitor toolbox that uses semaphores to simulate monitors with SC signaling.

Class *conditionVariable* is nested inside class *monitor*, which gives class *conditionVariable* access to member object *mutex* in the *monitorSC* class.

```
public class monitorSC { // monitor toolbox with SC signaling
   private binarySemaphore mutex = new binarySemaphore(1);
   protected final class conditionVariable {
      private countingSemaphore threadQueue = new countingSemaphore(0);
      private int numWaitingThreads = 0;
      public void signalC() {
         if (numWaitingThreads > 0) {
             numWaitingThreads--;
             threadQueue.V();
         }
      }
      public void signalCall() {
         while (numWaitingThreads > 0) {
            --numWaitingThreads;
            threadQueue.V();
         }
      }
      public void waitC() {
         numWaitingThreads++; threadQueue.VP(mutex);   mutex.P();
      }
      public boolean empty() { return (numWaitingThreads == 0); }
      public int length() { return numWaitingThreads; }
   }
   protected void enterMonitor() { mutex.P(); }
   protected void exitMonitor() { mutex.V(); }
}
```

Listing 4.23 Java monitor toolbox *monitorSC* with SC signaling.

4.7.2 A Toolbox for SU Signaling in Java

Listing 4.24 shows a Java monitor toolbox with SU signaling. The SU toolbox provides method *signalC_and_exitMonitor()*, which can be used when a signal operation is the last statement in a method (other than a return statement). When this method is called, the signaler does not wait in the *reentry* queue.

For example, method *deposit*() using *signalC_and_exitMonitor*() becomes:

```
public void deposit(int value) {
   enterMonitor();
   if (fullSlots == capacity)
      notFull.waitC();
   buffer[in] = value;
   in = (in + 1) % capacity;
   ++fullSlots;
   notEmpty.signalC_and_exitMonitor();
}
```

```
public class monitorSU { // monitor toolbox with SU signaling

   private binarySemaphore mutex = new binarySemaphore(1);
   private binarySemaphore reentry = new binarySemaphore(0);
   private int reentryCount = 0;
   proteced final class conditionVariable {
      private countingSemaphore threadQueue = new countingSemaphore(0);
      private int numWaitingThreads = 0;
      public void signalC() {
         if (numWaitingThreads > 0) {
            ++reentryCount;
            reentry.VP(threadQueue);
            --reentryCount;
         }
      }
      public void signalC_and_exitMonitor() { // does not execute reentry.P()
         if (numWaitingThreads > 0)threadQueue.V();
         else if (reentryCount > 0) reentry.V();
         else mutex.V();
      }
      public void waitC() {
         numWaitingThreads++;
         if (reentryCount > 0)   threadQueue.VP(reentry);
         else threadQueue.VP(mutex);
         --numWaitingThreads;
      }
      public boolean empty() { return (numWaitingThreads == 0); }
      public int length() { return numWaitingThreads; }
   }
   public void enterMonitor() { mutex.P(); }
   public void exitMonitor() {
      if (reentryCount > 0) reentry.V();
      else mutex.V();
   }
}
```

Listing 4.24 Java monitor toolbox *monitorSU* with SU signaling.

4.8 A Monitor Toolbox for Win32/C++/Pthreads

Listing 4.25 shows how to use the C++ *monitorSC* toolbox class to define a monitor for the bounded buffer problem.

```
class boundedBuffer : private monitorSC { // Note the use of private inheritance –
private:    // methods of monitorSC cannot be called outside of boundedbuffer.
    int fullSlots;    // # of full slots in the buffer
    int capacity;     // # of slots in the buffer
    int* buffer;
    int in, out;
    conditionVariable notFull;
    conditionVariable notEmpty;
public:
    boundedBuffer(int bufferCapacity);
    ~boundedBuffer();
    void deposit(int value, int ID);
    int withdraw(int ID);
};
```

Listing 4.25 Using the C++ *monitorSC* Toolbox class.

The *conditionVariable* constructor receives a pointer to the monitor object that owns the variable and uses this pointer to access the *mutex* object of the monitor.

```
   boundedBuffer(int bufferCapacity_) : fullSlots(0), capacity(bufferCapacity),
         in(0), out(0), notFull(this), notEmpty(this), buffer( new int[capacity]) {}
```

Monitor methods are written just as they are in Java:

```
    void boundedBuffer::deposit(int value) {
       enterMonitor();
       while (fullSlots == capacity)
          notFull.waitC();
       buffer[in] = value;
       in = (in + 1) % capacity;
       ++fullSlots;
       notEmpty.signalC();
       exitMonitor();
    }
```

4.8.1 A Toolbox for SC Signaling in C++/Win32/Pthreads

Listing 4.26 shows a C++/Win32/Pthreads monitor toolbox with SC signaling. Class *conditionVariable* is a friend of class *monitorSC*. This gives *conditionVariables* access to private member *mutex* of class *monitor*.

4.8.2 A Toolbox for SU Signaling in C++/Win32/Pthreads

Listing 4.27 shows a C++/Win32/Pthreads monitor toolbox with SU signaling.

```
class monitorSC { // monitor toolbox with SC signaling
protected:
   monitorSC() : mutex(1) {}
   void enterMonitor() { mutex.P(); }
   void exitMonitor() { mutex.V(); }
private:
   binarySemaphore mutex;
   friend class conditionVariable;   // conditionVariable needs access to mutex
};
class conditionVariable {
private:
   binarySemaphore threadQueue;
   int numWaitingThreads;
   monitorSC& m;  // reference to monitor that owns this conditionVariable
public:
   conditionVariable(monitorSC* mon) : threadQueue(0),numWaitingThreads(0),
      m(*mon) {}
   void signalC();
   void signalCall();
   void waitC();
   bool empty() { return (numWaitingThreads == 0); }
   int length() { return numWaitingThreads; }
};

void conditionVariable::signalC() {
   if (numWaitingThreads > 0) {
      --numWaitingThreads;
      threadQueue.V();
   }
}
void conditionVariable::signalCall() {
   while (numWaitingThreads > 0) {
      --numWaitingThreads;
      threadQueue.V();
   }
}
void conditionVariable::waitC(int ID) {
   numWaitingThreads++;
   threadQueue.VP(&(m.mutex));
   m.mutex.P();
}
```

Listing 4.26 C++ monitor toolbox for SC signaling.

```
class monitorSU { // monitor toolbox with SU signaling
protected:
   monitorSU() : reentryCount(0), mutex(1), reentry(0) { }
   void enterMonitor() { mutex.P(); }
   void exitMonitor(){
      if (reentryCount > 0) reentry.V();
      else mutex.V();
   }
private:
   binarySemaphore mutex;  binarySemaphore reentry;
   int reentryCount; friend class conditionVariable;
};
class conditionVariable {
private:
   binarySemaphore threadQueue;
   int numWaitingThreads;
   monitorSU& m;
public:
   conditionVariable(monitorSU* mon):threadQueue(0),numWaitingThreads(0),
      m(*mon){}
   void signalC();    void signalC_and_exitMonitor();
   void waitC();
   bool empty() { return (numWaitingThreads == 0); }
   int length() { return numWaitingThreads; }
};
void conditionVariable::signalC() {
   if (numWaitingThreads > 0) {
      ++(m.reentryCount);
      m.reentry.VP(&threadQueue);
      --(m.reentryCount);
   }
}
void conditionVariable::signalC_and_exitMonitor() {
   if (numWaitingThreads > 0) threadQueue.V();
   else if (m.reentryCount > 0) m.reentry.V();
   else m.mutex.V();
}
void conditionVariable::waitC() {
   numWaitingThreads++;
   if (m.reentryCount > 0) threadQueue.VP(&(m.reentry));
   else threadQueue.VP(&(m.mutex));
   --numWaitingThreads;
}
```
Listing 4.27 C++ monitor toolbox for SU signaling.

4.9 Nested Monitor Calls

A thread T executing in a method of monitor M1 may call a method in another monitor M2. This is called a nested monitor call.

If thread T releases mutual exclusion in M1 when it makes the nested call to M2, it is said to be an *open* call. If mutual exclusion is not released, it is a *closed* call. Closed calls are prone to create deadlocks.

```
    class First extends monitorSU {  class Second extends monitorSU {
       Second M2;
       public void A1() {               public void A2() {
         …                                …
         M2.A2();                         wait();            // Thread A is blocked
         …                                …
       }                                }
       public void B1() {               public void B2() {
         M2.B2();                         …
         …                                signal-and-exit();  // wakeup Thread A
       }                                }
}
```

Suppose that we create an instance M1 of the First monitor and that this instance is used by two threads:

Thread A	Thread B
M1.A1();	M1.B1();

- Assume that Thread *A* enters method *A1()* of monitor *M1* first and makes a closed monitor call to *M2.A2()*.
- Assume that Thread *A* is then blocked on the *wait* statement in method *A2()*.
- Thread *B* intends to signal Thread *A* by calling method *M1.B1*, which issues a nested call to *M2.B2()* where the signal is performed.
- But this is impossible since Thread *A* retains mutual exclusion for monitor *M1* while Thread *A* is blocked on the wait statement in monitor *M2*.
- Thus, Thread *B* is unable to enter *M1* and a deadlock occurs.

Open monitor calls are implemented by having the calling thread release mutual exclusion when the call is made and reacquire mutual exclusion when the call returns.

The monitor toolboxes described in the previous section make this easy to do. For example, method *A1()* above becomes:

```
public void A1() {
   enterMonitor();// acquire mutual exclusion
   …
   exitMonitor();  // release mutual exclusion
   M2.A2();
   enterMonitor();// reacquire mutual exclusion
   …
   exitMonitor();  // release mutual exclusion
}
```

This gives equal priority to the threads returning from a nested call and the threads trying to enter the monitor for the first time, since both groups of threads call *enterMonitor()*.

Open calls can create a problem if shared variables in monitor *M1* are used as arguments and passed by reference on nested calls to *M2*.

This allows shared variables of *M1* to be accessed concurrently by a thread in *M1* and a thread in *M2*, violating the requirement for mutual exclusion.

5. Message-Passing

Threads can communicate and synchronize by sending and receiving messages across channels.

A channel is an abstraction of a communication path between threads:

- If shared memory is available, channels can be implemented as objects that are shared by threads.
- Without shared memory, channels can be implemented using kernel routines that transport messages across a communication network (Chapter 6)

Outline:

- Basic message passing using send and receive commands.
- Rendezvous.
- Testing and debugging message-passing programs.

5.1 Channel Objects

Threads running in the same program (or process) can access channel objects in shared memory.

```
channel requestChannel = new channel();
channel replyChannel = new channel();

          Thread1                                   Thread2
requestChannel.send(request);     ⇒   request = requestChannel.receive();
reply = replyChannel.receive();   ⇐   replyChannel.send(reply);
```

A thread that calls *send()* or *receive()* may be blocked. Thus, send and receive operations are used both for communication and synchronization.

Types of send and receive operations:

- blocking send: the sender is blocked until the message is received (by a *receive()* operation).
- buffer-blocking send: messages are queued in a bounded message buffer, and the sender is blocked only if the buffer is full.
- non-blocking send: messages are queued in an unbounded message buffer, and the sender is never blocked.
- blocking receive: the receiver is blocked until a message is available.
- non-blocking receive: the receiver is never blocked. A receive command returns an indication of whether or not a message was received.

Asynchronous message passing: blocking receive operations are used with either non-blocking or buffer-blocking send operations.

S*ynchronous message passing:*

- The send and receive operations are both blocking.
- Either the sender or the receiver thread will be blocked, whichever one executes its operation first.
- Can be simulated using asynchronous message passing: the sender can issue a buffer-blocking send followed immediately by a blocking receive.

5.1.1 Channel Objects in Java

Threads in the same Java Virtual Machine (JVM) can communicate and synchronize by passing messages through user-defined *channels* that are implemented as shared objects.

```
public abstract class channel {
    public abstract void send(Object m); // send a message object
    public abstract void send();         // acts as a signal to the receiver
    public abstract Object receive();    // receive an object.
}
```

Three types of channels:

- *mailbox*: many senders and many receivers may access a mailbox object
- *port*: many senders but only one receiver may access a port object
- *link*: only one sender and one receiver may access a link object

Each type has a synchronous version and an asynchronous version.

A synchronous *mailbox* class is shown in Listing 5.1:

- A sending thread copies its message into the channel's *message* object and issues *sent.V()* to signal that the message is available.
- The sending thread executes *received.P()* to wait until the message is received.
- The receiving thread executes *sent.P()* to wait for a message from the sender.
- When the sender signals that a message is available, the receiver makes a copy of the *message* object and executes *received.V()* to signal the sender that the message has been received.

```
public class mailbox extends channel {
   private Object message = null;
   private final Object sending = new Object();
   private final Object receiving = new Object();
   private final binarySemaphore sent = new binarySemaphore(0);
   private final binarySemaphore received = new binarySemaphore(0);
   public final void send(Object sentMsg) {
      if (sentMsg == null) {throw new
         NullPointerException("Null message passed to send()");
      }
      synchronized (sending) {
      message = sentMsg;
      sent.V();                 // signal that the message is available
      received.P();             // wait until the message is received
      }
   }
   public final void send() {
      synchronized (sending) {
         message = new Object();// send a null message
         sent.V();                 // signal that message is available
         received.P();             // wait until the message is received
      }
   }
   public final Object receive() {
      Object receivedMessage = null;
      synchronized (receiving) {
        sent.P();                  // wait for message to be sent
        receivedMessage = message;
        received.V();              // signal the sender that the message has
      }                            //    been received
      return receivedMessage;
   }
}
```

Listing 5.1 A synchronous *mailbox* class.

Note: A *send()* operation with no message acts as a signal to the receiver that some event has occurred. Such a send is analogous to a *V()* operation on a semaphore.

The *send()* methods for classes *port* and *mailbox* are the same.

The *receive()* methods for classes *port* and *link* are also the same.

Only one thread can ever execute a *receive* operation on a *port* or *link* object. In Listing 5.2, an exception is thrown if multiple receivers are detected.

Since a *link* can have only one sender, a similar check is performed in the *send()* method of class *link*.

```
public final Object receive() {
   synchronized(receiving) {
        if (receiver == null) // save the first thread to call receive
            receiver = Thread.currentThread();
        // if currentThread() is not first thread to call receive, throw an exception
        if (Thread.currentThread() != receiver) throw new
              InvalidLinkUsage("Attempted to use link with multiple receivers");
        Object receivedMessage = null;
        sent.P();                        // wait for the message to be sent
        receivedMessage = message;
        received.V();                    // signal the sender that the message has
        return receivedMessage; // been received
   }
}
```

Listing 5.2 Synchronous *receive* method for the *link* and *port* classes.

An asynchronous *mailbox* class is shown in Listing 5.3:

- The implementation of the buffer-blocking *send()* operation is based on the bounded-buffer solution in Section 3.5.2.
- The *send()* will block if the message buffer is full.

- The messages that a thread sends to a particular mailbox are guaranteed to be received in the order they are sent.
- Also, If Thread1 executes a *send()* operation on a particular mailbox before Thread2 executes a *send()* operation on the same mailbox, then Thr*ead1's message will be received from that mailbox before Thread2's message.

Listing 5.4 shows how to use the *link* class.

- *Producer* and *Consumer* threads exchange messages with a *Buffer* thread using links *deposit* and *withdraw*.
- The *Buffer* thread implements a 1-slot bounded buffer.
- The *Producer* builds a *Message* object and sends it to the *Buffer* over the *deposit* link.
- The *Buffer* then sends the *Message* to the *Consumer* over the *withdraw* link.

```
public final class asynchMailbox extends channel {

   private final int capacity = 100;
   private Object messages[] = new Object[capacity]; // message buffer
   private countingSemaphore messageAvailable = new countingSemaphore(0);
   private countingSemaphore slotAvailable = new
      countingSemaphore(capacity);
   private binarySemaphore senderMutex = new binarySemaphore(1);
   private binarySemaphore receiverMutex = new binarySemaphore(1);
   private int in = 0, out = 0;

   public final void send(Object sentMessage) {
      if (sentMessage == null) {
         throw new NullPointerException("null message passed to send()");
      }
      slotAvailable.P();
      senderMutex.P();
      messages[in] = sentMessage;   in = (in + 1) % capacity;
      senderMutex.V();
      messageAvailable.V();
   }
```

Listing 5.3 Asynchronous *asynchMailbox* class.

```
   public final void send() {
   /* same as send(Object sentMessage) above except that the line
      “messages[in] = sentMessage;” becomes “messages[in] = new Object();” */
   }

   public final Object receive() {
      messageAvailable.P();
      receiverMutex.P();
      Object receivedMessage = messages[out];   out = (out + 1) % capacity;
      receiverMutex.V();
      slotAvailable.V();
      return receivedMessage;
  }
}
```

Listing 5.3 (cont.) Asynchronous *asynchMailbox* class.

```
public final class boundedBuffer {
   public static void main (String args[]) {
       link deposit = new link();    link withdraw = new link();
       Producer producer = new Producer(deposit);
       Consumer consumer = new Consumer(withdraw);
       Buffer buffer = new Buffer(deposit,withdraw);
       // buffer will be terminated when producer and consumer are finished
       buffer.setDaemon(true);   buffer.start();
       producer.start();  consumer.start();
   }
}

final class Message {
   public int number;
   Message(int number ) {this.number = number;}
}
final class Producer extends Thread {
   private link deposit;
   public Producer (link deposit) { this.deposit = deposit; }
   public void run () {
      for (int i = 0; i<3; i++) {
          System.out.println("Produced " + i);
          deposit.send(new Message(i));
      }
   }
}
```

Listing 5.4 Java bounded buffer using channels.

```
final class Consumer extends Thread {
   private link withdraw;
   public Consumer (link withdraw) { this.withdraw = withdraw; }
   public void run () {
      for (int i = 0; i<3; i++) {
         Message m = (Message) withdraw.receive(); // message from Buffer
         System.out.println("Consumed " + m.number);
      }
   }
}

final class Buffer extends Thread {
   private link deposit, withdraw;
   public Buffer (link deposit, link withdraw) { this.deposit = deposit;
      his.withdraw = withdraw; }
   public void run () {
      while (true) {
         Message m = ((Message) deposit.receive()); // message from Producer
         withdraw.send(m);                          // send message to Consumer
      }
   }
}
```

Listing 5.4 Java bounded buffer using channels.

Note: The receiver expects mutually exclusive access to the message. If the sender needs to access a message after it is sent, or just to be completely safe, the sender should send a clone (copy) of the message:

```
   final class Message implements Cloneable {
      public int number;
      Message(int number ) {this.number = number;}
   }

   Message m(1);
   deposit.send((Message) m.clone());
```

This prevents the sending and receiving threads from referencing the same message objects.

5.1.2 Channel Objects in C++/Win32

Listing 5.5 shows a C++ version of the synchronous *mailbox* class.

Methods *send()* and *receive()* operate on "smart pointer" objects of type *message_ptr*<T>.

- Smart pointers mimic simple pointers by providing pointer operations like dereferencing (using operator *) and indirection (using operator ->).
- A message_ptr<T> object contains a pointer to a message object of type T.
- Ownership and memory management are handled by maintaining a count of the *message_ptr* objects that point to the same message. Copying a *message_ptr* object adds one to the count. The message is deleted when count becomes zero.

Sending and receiving *message_ptr*<T> objects simulates message passing in Java.

- Messages are shared by the sending and receiving threads
- messages are automatically deleted when they are no longer being referenced.
- virtually any type T of message object can be used.

Example: A message containing an integer:
```
class Message {
public:
   int contents;
   Message(int contents_) : contents(contents_){}
   Message(const Message& m) : contents(m.contents) {}
   Message* clone() const {return new Message(*this);}
};
```

A *Message* object can be sent over channel *deposit* using:
```
mailbox<Message> deposit;                     // create a mailbox for Message objects
message_ptr<Message> m(new Message(i));       // create a message object m
deposit.send(m);                              // send message m
```

Message objects are received using:
```
message_ptr<Message> m = deposit.receive();
std::cout << "Received:" << m->contents << std::endl;
```

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the sender should send a copy of the message:

```
   Message m1(1);
   message_ptr<Message> m2(m1.clone());    // send copy of m1
   deposit.send(m2);
```

The *link* and *port* classes can easily be produced from C++ class *mailbox*<T> and the Java *link* and *port* counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in Listing 5.4.

```
class Message {
public:
   int contents;
   Message(int contents_) : contents(contents_){}
   Message(const Message& m) : contents(m.contents) {}
   Message* clone() const {return new Message(*this);}
};

class Producer : public Thread {
private:
   mailbox<Message>& deposit;
public:
   Producer (mailbox<Message>& deposit_) : deposit(deposit_) {}
   virtual void* run () {
      for (int i=0; i<3; i++) {
         message_ptr<Message> m(new Message(i));
         std::cout << "Producing " << i << std::endl;
         deposit.send(m);
      }
      return 0;
   }
};
```

```
class Consumer : public Thread {
private:
   mailbox<Message>& withdraw;
public:
   Consumer (mailbox<Message>& withdraw_) : withdraw(withdraw_) {}
   virtual void* run () {
      for (int i=0; i<3; i++) {
         message_ptr<Message> m = withdraw.receive();
         std::cout << "Consumed " << m->contents << std::endl;
      }
      return 0;
   }
};

class Buffer : public Thread {
private:
   mailbox<Message>& deposit; mailbox<Message>& withdraw;
public:
   Buffer (mailbox<Message>& deposit_, mailbox<Message>& withdraw_)
      : deposit(deposit_),  withdraw(withdraw_) {}
   virtual void* run () {
      for (int i=0; i<3; i++) {
         message_ptr<Message> m = deposit.receive(); withdraw.send(m);
      }
      return 0;
   }
};

int main () {
   mailbox<Message> deposit;    mailbox<Message> withdraw;
   std::auto_ptr<Producer> producer(new Producer(deposit));
   std::auto_ptr<Consumer> consumer(new Consumer(withdraw));
   std::auto_ptr<Buffer> buffer(new Buffer(deposit,withdraw));
   producer->start(); consumer->start(); buffer->start();
   producer->join();  consumer->join(); buffer->join();
   return 0;
}
```

Listing 5.6 C++ bounded buffer using channels.

5.2 Rendezvous

The following message-passing paradigm is common in a client-server environment:

```
           Client_i                                   Server
                                              loop {
  request.send(clientRequest);   ⇒               clientRequest = request.receive();
                                                 /* process clientRequest and compute result */
  result = reply_i.receive();    ⇐               reply_i.send(result);
                                              }
```

Implementing this with basic message passing:

- one channel that the server can use to receive client requests
- one channel for each client's reply. That is, each client uses a separate channel to receive a reply from the server.

This paradigm can be implemented instead as follows:

```
  Client                         Server
                                 entry E;
                                 loop {
                                    E.accept(clientRequest, result) {
  E.call(clientRequest,result);       ⇔ /* process Client's request and compute result */
                                    } // end accept()
                                 } // end loop
```

The server uses a new type of channel called an *entry*.

In the client, the pair of send and receive statements:

```
request.send(clientRequest);
result = reply.receive();
```

is combined into the single entry call statement: E.call(clientRequest, result);

This call on entry *E* is very similar to a procedure call:

- Object *clientRequest* holds the message being sent to the server. When the call returns, object *result* will hold the server's reply.

In the server, the code that handles the client's request is in the form an accept statement for entry *E*:

```
E.accept(clientRequest,result) {
   /* process the clientRequest and compute result*/
   result = ...;
 }
```

Only one thread can accept the entry calls made to a given entry. When a server thread executes an accept statement for entry *E*:

- if no entry call for entry *E* has arrived, the server waits
- if one or more entry calls for *E* have arrived, the server accepts one call and executes the body of the accept statement. When the execution of the accept statement is complete, the entry call returns to the client with the server's reply, and the client and server continue execution.

This interaction is referred to as a *rendezvous*.

- Rendezvous are a form of synchronous communication.
- A client making an entry call is blocked until the call is accepted and a reply is returned.
- Executing an accept statement blocks the server until an entry call arrives.

Listing 5.7 shows a Java class named *entry* that simulates a rendezvous [. Class *entry* uses the *link* and *port* channels of section 5.1.

A client issues an entry call to entry *E* as follows:

```
reply = E.call(clientRequest);
```

```
class entry {
   private port requestChannel = new port();
   private callMsg cm;
   public Object call(Object request) throws InterruptedException {
       link replyChannel = new link();
       requestChannel.send(new callMsg(request,replyChannel));
       return replyChannel.receive();
   }
   public Object call() throws InterruptedException {
       link replyChannel = new link();
       requestChannel.send(new callMsg(replyChannel));
       return replyChannel.receive();
   }
   public Object accept() throws InterruptedException {
   // check the for multiple callers is not shown
       cm = (callMsg) requestChannel.receive();
       return cm.request;
   }
   public void reply(Object response) throws InterruptedException {
      cm.replyChannel.send(response);
   }
   public void reply() throws InterruptedException { cm.replyChannel.send(); }
   public Object acceptAndReply() throws InterruptedException {
   // the check for multiple callers is not shown
      cm = (callMsg) requestChannel.receive();
      cm.replyChannel.send(new Object()); // send empty reply back to client
      return cm.request;
   }
   private class callMsg {
      Object  request;
      link replyChannel;
      callMsg(Object m, link c) {request=m; replyChannel=c;}
      callMsg(link c) {request = new Object(); replyChannel=c;}
   }
}
```

Listing 5.7 Java class *entry*.

Implementation of method *call()*:

- a *send*() operation is issued on a port named *requestChannel*.
- The *send*() operation sends the *clientRequest*, along with a *link* named *replyChannel*, to the server.
- A new *replyChannel* is created on each execution of *call()*, so that each client sends its own *replyChannel* to the server.
- The *call()* operation ends with *replyChannel.receive*(), allowing the client to wait for the server's reply.

The server accepts entry calls from its client and issues a reply:

```
request = E.accept();       // accept client's call to entry E
…
E.reply(response);   // reply to the client
```

The *accept*() method in class *Entry* is implemented using a *receive*() operation on the *requestChannel* (see Fig. 5.8):

- *accept*() receives the *clientRequest* and the *replyChannel* that was sent with it.
- *accept*() saves the *replyChannel* and returns the *clientRequest* to the server thread.
- method *reply*() sends the server's response back to the client using *replyChannel.send*().
- the client waits for the server's response by executing r*eplyChannel.receive*() in method *call()*.

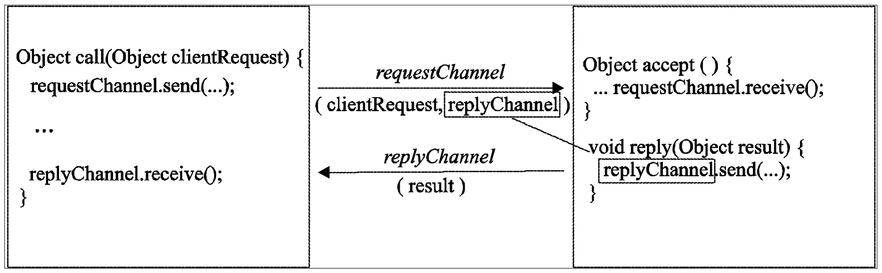

Object call(Object clientRequest) {
requestChannel.send(...);
...
replyChannel.receive();
}
requestChannel
(clientRequest, replyChannel)
replyChannel
(result)
Object accept ( ) {
... requestChannel.receive();
}
void reply(Object result) {
replyChannel.send(...);
}

Figure 5.8 Implementing entries.

If the server does not need to compute a reply for the client, the server can execute method *acceptAndReply()*:

- accepts the client's request and sends an empty reply back to the client so that the client is not delayed.
- The client simply ignores the reply.

Listing 5.9 shows a client and server program using Java *entries* and rendezvous.

```
public final class clientServer {
    public static void main (String args[]) {
       entry E = new entry();
       Client c1 = new Client(E, 2);   // send value 2 to the server using entry E
       Client c2 = new Client(E, 4);   // send value 4 to the server using entry E
       Server s = new Server(E);
       s.setDaemon(true);
       s.start(); c1.start(); c2.start();
    }
}
final class Message {
       public int number;
       Message(int number) { this.number = number; }
}
final class Client extends Thread {
    private entry E;
    int number;
    public Client (entry E, int number) { this.E = E; this.number = number;     }
    public void run () {
       try {
          // send number and wait for reply
          int i = ((Integer)E.call(new Message(number))).intValue();
          System.out.println (number + " x " + number + " = " + i); // e.g., 2x2=4.
       }
       catch(InterruptedException e) {}
    }
}
final class Server extends Thread {
    private entry E;
    public Server (entry E) { this.E = E; }
    public void run () {
       Message m;  int number;
       while (true) {
          try {
             m = ((Message) E.accept());  // accept number from Client
             number = m.number;
             E.reply(new Integer(number*number));// reply to client
          }
          catch(InterruptedException e) {}
       }
    }
}
```
Listing 5.9 Client and server using Java *entries* and rendezvous.

5.3 Selective Wait

Assume that server thread *boundedBuffer* has entries *deposit* and *withdraw*. Two possible implementations of the *run()* method of thread *boundedBuffer* are shown below:

```
Implementation 1                  Implementation 2
while (true) {                    while (true) {
   if (buffer is not full) {         if (buffer is not empty) {
      item = deposit.acceptAndReply();  withdraw.accept();
      …                                 …
   }                                    withdraw.reply(item);
                                     }
   if (buffer is not empty) {
      withdraw.accept();             if (buffer is not full) {
      …                                 item = deposit.acceptAndReply();
      withdraw.reply(item);             …
   }                                 }
}                                 }
```

Both implementations create unnecessary delays for threads calling entries *deposit* and *withdraw*:

- Implementation 1: while *boundedBuffer* is blocked waiting to accept an entry call to *deposit*, it is possible that a call to *withdraw* has arrived and is waiting to be accepted.
- Implementation 2: while *boundedBuffer* is blocked waiting to accept an entry call to *withdraw*, it is possible that a call to *deposit* has arrived and is waiting to be accepted.

⇒ allow a thread to wait for a set of entries instead of a single entry:

```
select:
    when (the buffer is not full) => accept a call to entry deposit and deposit the item;
or
    when (the buffer is not empty) => accept a call to entry withdraw and return item;
end select;
```

When one of the entries is acceptable and the other is not, then the acceptable entry is selected for execution. When both entries are acceptable, then one of them is selected for execution.

The Ada language provides a select statement just like the one above.

A select statement can optionally contain either a delay or else alternative:

- A delay t alternative is selected if no entry can be accepted within t seconds.
- An else alternative is executed immediately if no entries are acceptable.

Ada's selective wait statement can be simulated in Java by a class named *selectiveWait:*

1. Create a *selectiveWait* object

```
selectiveWait select = new selectiveWait();
```

2. Add one or more *selectableEntry* objects to the *selectiveWait*:

```
selectableEntry deposit = new selectableEntry();
selectableEntry withdraw = new selectableEntry();
select.add(deposit);
select.add(withdraw);
```

selectableEntry objects are *entry* objects that have been extended so they can be used as alternatives in a *selectiveWait*.

A *selectiveWait* can also contain one *delayAlternative*:

```
selectiveWait.delayAlternative delayA = select.new delayAlternative(500);   // .5 sec.
```

or one *elseAlternative* (but not both):

```
selectiveWait.elseAlternative elseA = select.new elseAlternative();
```

A *delay* or *else* alternative is added to a selective wait in the same way as a *selectableEntry*:   select.add(delayA);

Each *selectableEntry* and *delayAlternative* is associated with a condition, called a *guard*, which determines whether the alternative is allowed to be selected.

The guard for each *selectableEntry* and *delayAlternative* must be evaluated before a selection takes place.

Method *guard*() is called with a *boolean* expression that sets the guard to *true* or *false*:

```
deposit.guard (fullSlots<capacity);  // guard set to boolean value (fullSlots<capacity)
withdraw.guard(fullSlots>0);         // guard set to boolean value (fullSlots>0)
delayA.guard(true);                  // guard is always set to true
```

Method *choose*() selects one of the alternatives with a true guard:

```
switch (select.choose()) {
   case 1:  deposit.acceptAndReply();  /* alternative 1 */
            …
            break;
   case 2:  withdraw.accept();         /* alternative 2 */
            …
            withdraw.reply(value);
            break;
   case 3:  delayA.accept();           /* alternative 3 */
            break;
}
```

Method *choose()* returns the alternative number of the selected alternative. Alternative numbers are based on the order in which the alternatives are added to the selective wait.

In the example above,

- *selectableEntry* object *deposit* was added first, thus its alternative number is 1.
- the alternative number for *withdraw* is 2
- the number for *delayA* is 3.

A switch statement uses the alternative number to execute the appropriate alternative.

When the *selectiveWait* contains no *delay* or *else* alternatives:

- *choose*() will select an open *accept()* alternative (i.e., one with a true guard) that has a waiting entry call.
- if several *accept()* alternatives are open and have waiting entry calls, the one whose entry call arrived first is selected.
- if one or more *accept()* alternatives are open but none have a waiting entry call, *choose()* blocks until an entry call arrives for one of the open *accept()* alternatives.

When the *selectiveWait* has an *else* or *delay* alternative:

- an e*lse* alternative is executed if all the *accept()* alternatives are closed, or all the open *accept()* alternatives have no waiting entry calls.
- an open *delay* alternative is selected when its expiration time is reached if no open *accept()* alternatives can be selected prior to the expiration time.

When all of the guards of the accept alternatives are false, and there is no delay alternative with a true guard and no else alternative, method *choose()* throws a *SelectException* indicating that a deadlock has been detected.

Listing 5.10 shows a *boundedBuffer* server class that uses a selective wait. The *delayAlternative* simply displays a message when it is accepted.

Note that the guards of all the alternatives are evaluated each iteration of the while-loop.

- Changes made to variables *fullSlots* and *emptySlots* in the alternatives of the switch statement may change the values of the guards for entries *deposit* and *withdraw*.
- This requires each guard to be reevaluated before the next selection occurs.

```
final class boundedBuffer extends Thread {
   private selectableEntry deposit, withdraw;
   private int fullSlots=0; private int capacity = 0;
   private Object[] buffer = null; private int in = 0, out = 0;
   public boundedBuffer(selectableEntry deposit, selectableEntry withdraw, int capacity)
   {  this.deposit = deposit; this.withdraw = withdraw; this.capacity = capacity;
      buffer = new Object[capacity];
   }

   public void run() {
      try {
         selectiveWait select = new selectiveWait();
         selectiveWait.delayAlternative delayA = select.new delayAlternative(500);
         select.add(deposit);           // alternative 1
         select.add(withdraw);          // alternative 2
         select.add(delayA);            // alternative 3
         while(true) {
            withdraw.guard(fullSlots>0);
            deposit.guard (fullSlots<capacity);
            delayA.guard(true);
            switch (select.choose()) {
               case 1:  Object o = deposit.acceptAndReply();
                        buffer[in] = o;
                        in = (in + 1) % capacity;  ++fullSlots;
                        break;
               case 2:  withdraw.accept();
                        Object value = buffer[out];
                        withdraw.reply(value);
                        out = (out + 1) % capacity; --fullSlots;
                        break;
               case 3:  delayA.accept();
                        System.out.println("delay selected");
                        break;
            }
         }
      }catch (InterruptedException e) {}
       catch (SelectException e) {
         System.out.println("deadlock detected"); System.exit(1);
       }
   }
}
```

Listing 5.10 Java bounded buffer using a *selectiveWait*.

5.4 Message-Based Solutions to Concurrent Programming Problems

5.4.1 Readers and Writers

Listing 5.11 shows a solution to the readers and writers problem for strategy R>W.1 (many readers or one writer, with readers having a higher priority).

Class *Controller* uses a *selectiveWait* with *selectablePort* objects *startRead*, *endRead*, *startWrite*, and *endWrite*.

(A *selectablePort* object is a synchronous port that can be used in selective waits.)

- The guard for *startRead* is (*!writerPresent*), which ensures that no writers are writing when a reader is allowed to start reading.
- The guard for *startWrite* is (*!writerPresent && readerCount == 0 && startRead.count() == 0*). This allows a writer to start writing only if no other writer is writing, no readers are reading, and no reader is waiting for its call to *startRead* to be accepted.
- Note: the call to *startRead.count()* returns the number of *startRead.send()* operations that are waiting to be received.

The *selectablePorts* are private members of *Controller* and thus cannot be directly accessed by reader and writer threads:

- readers and writers call public methods *read*() and *write*(), respectively.
- public methods *read*() and *write*() their calls on to the private entries, ensuring that the entries are called in the correct order (i.e., *startRead* is called before *endRead*, and *startWrite* is called before *endWrite*).

```
final class Controller extends Thread {
//Strategy R>W.1 : Many readers or one writer; readers have a higher priority
    private selectablePort startRead = new selectablePort ();
    private selectablePort endRead = new selectablePort ();
    private selectablePort startWrite = new selectablePort ();
    private selectablePort endWrite = new selectablePort ();
    private boolean writerPresent = false;
    private int readerCount = 0;    private int sharedValue = 0;
    public int read() {
       try {startRead.send();} catch(Exception e) {}
       int value = sharedValue;
       try {endRead.send();} catch(Exception e) {}
       return value;
    }
    public void write(int value) {
       try {startWrite.send();} catch(Exception e) {}
       sharedValue = value;
       try {endWrite.send();} catch(Exception e) {}
    }
    public void run() {
       try {
          selectiveWait select = new selectiveWait();
          select.add(startRead);     // alternative 1
          select.add(endRead);       // alternative 2
          select.add(startWrite);    // alternative 3
          select.add(endWrite);      // alternative 4
          while(true) {
             startRead.guard(!writerPresent);
             endRead.guard(true);
             startWrite.guard(!writerPresent && readerCount == 0 &&
                startRead.count() == 0);
             endWrite.guard(true);
             switch (select.choose()) {
                case 1: startRead.receive(); ++readerCount; break;
                case 2: endRead.receive(); --readerCount; break;
                case 3: startWrite.receive();    writerPresent = true;    break;
                case 4: endWrite.receive(); writerPresent = false;   break;
             }
          }
       } catch (InterruptedException e) {}
    }
}
```

Listing 5.11 Readers and writers using a *selectiveWait*.

5.4.2 Resource Allocation

Listing 5.12 shows a solution to the resource allocation problem.

- A *resourceServer* manages three resources.
- A client calls entry *acquire* to get a resource and entry *release* to return the resource.
- Vector *resources* contains the IDs of the available resources.
- The integer *available* is used to count the number of available resources.
- A resource can be acquired if the guard (*available>0*) is true.
- A resource's ID is given to the client that *acquires* the resource. The client gives the resource ID back when it *releases* the resource.

Listing 5.13 shows an SU monitor solution for this same resource allocation problem. Monitor *resourceMonitor* and thread *resourceServer* demonstrate a mapping between monitors and server threads:

- Thread *resourceServer* is an active object that executes concurrently with the threads that call it.
- The *resourceMonitor* is a passive object, not a thread, which does not execute until it is called.
- A monitor cannot prevent a thread from entering one of its methods (although the monitor can force threads to enter one at a time). However, once a thread enters the monitor, the thread may be forced to wait on a condition variable until a condition becomes true.
- Condition synchronization in a server thread works in the opposite way. A server thread will prevent an entry call from being accepted until the condition for accepting the call becomes true.

It has been shown elsewhere that a program that uses shared variables with semaphores or monitors can be transformed into an equivalent program that uses message passing, and vice versa.

```
final class resourceServer extends Thread {
    private selectableEntry acquire;
    private selectableEntry release;
    private final int numResources = 3;
    private int available = numResources;
    Vector resources = new Vector(numResources);
    public resourceServer(selectableEntry acquire, selectableEntry release) {
        this.acquire = acquire;
        this.release = release;
        resources.addElement(new Integer(1));
        resources.addElement(new Integer(2));
        resources.addElement(new Integer(3));
    }
    public void run() {
       int unitID;
       try {
          selectiveWait select = new selectiveWait();
          select.add(acquire);     // alternative 1
          select.add(release);     // alternative 2
          while(true) {
             acquire.guard(available > 0);
             release.guard(true);
             switch (select.choose()) {
                case 1: acquire.accept();
                        unitID = ((Integer) resources.firstElement()).intValue();
                         // Replying early allows the client to proceed as soon as possible
                        acquire.reply(new Integer(unitID));
                        --available;
                        resources.removeElementAt(0);
                        break;
                case 2: unitID = ((Integer)
                           release.acceptAndReply()).intValue();
                        ++available;
                        resources.addElement(new Integer(unitID));
                        break;
             }
          }
       } catch (InterruptedException e) { }
    }
}
```

Listing 5.12 Resource allocation using a *selectiveWait*.

```
final class resourceMonitor extends monitorSU {
   private conditionVariable freeResource = new conditionVariable();
   private int available = 3;
   Vector resources = new Vector(3);
   public resourceMonitor() {
      resources.addElement(new Integer(1));
      resources.addElement(new Integer(2));
      resources.addElement(new Integer(3));
   }
   public int acquire() {
      int unitID;
      enterMonitor();
      if (available == 0)
          freeResource.waitC();
      else
          --available;
      unitID = ((Integer)resources.firstElement()).intValue();
      resources.removeElementAt(0);
      exitMonitor();
      return unitID;
   }
   public void release(int unitID) {
       enterMonitor();
       resources.addElement(new Integer(unitID));
       if (freeResource.empty()) {
           ++available;
           exitMonitor();
       }
       else
           freeResource.signalC_and_exitMonitor();
   }
}
```

Listing 5.13 Resource allocation using a monitor.

5.4.3 Simulating Counting Semaphores

Listing 5.14 shows an implementation of a *countingSemaphore* that uses a *selectiveWait* with *selectablePorts* named *P* and *V*.

Clients call public methods *P()* and *V()*, which pass the calls on to the (private) ports.

A *P()* operation can be performed only if the guard *(permits>0)* is true.

```
final class countingSemaphore extends Thread {
   private selectablePort V, P;
   private int permits;
   public countingSemaphore(int initialPermits) {permits = initialPermits;}
   public void P() { P.send();}
   public void V() {V.send();}
   public void run() {
      try {
          selectiveWait select = new selectiveWait();
          select.add(P);              // alternative1
          select.add(V);              // alternative 2
          while(true) {
              P.guard(permits>0);
              V.guard(true);
              switch (select.choose()) {
                 case 1:  P.receive();
                          --permits;
                          break;
                 case 2:  V.receive();
                          ++permits;
                          break;
              }
          }
      } catch (InterruptedException e) {}
   }
}
```

Listing 5.14 Using a s*electiveWait* to simulate a counting semaphore.

6. Message-Passing in Distributed Programs

A distributed program is a collection of concurrent processes that run on a network of computers:

- Typically, each process is a concurrent program that executes on a single computer.
- A process (or program) on one computer communicates with processes on other computers by passing messages across the network.
- The threads in a single process all execute on the same computer, so they can use message passing and/or shared variables to communicate.

Outline:

- Mechanisms for message passing in distributed programs.
- Java message passing classes
- Distributed solutions to several classical synchronization problems.\
- Tracing, testing, and replaying distributed programs.

6.1 TCP Sockets

Communication channels are formed across a communications network with help from the operating system.

- Each thread creates an endpoint object that represents its end of the network channel.
- The operating system manages the hardware and software that is used to transport messages between the endpoints, i.e., “across the channel”.

The endpoint objects are called *sockets* (Fig. 6.1):

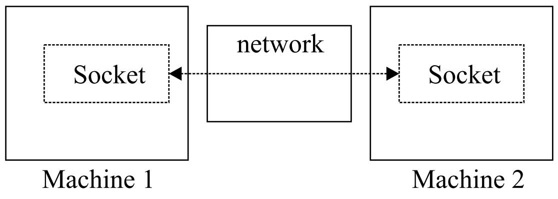


Figure 6.1 Sockets are the endpoints of channels across a network.

- The client thread’s socket specifies a local IO port to be used for sending messages (or the IO port can be chosen by the operating system).
- The client’s socket also specifies the address of the destination machine and the port number that is expected to be bound to the server thread’s socket.
- The server’s socket specifies a local IO port for receiving messages. Messages can be received from any client that knows both the server’s machine address and the port number bound to the server’s socket.
- The client issues a request to the server to form a connection between the two sockets. Once the server accepts the connection request, messages can be passed in either direction across the channel.

6.1.1 Channel Reliability

Messages that travel across a network may be lost or corrupted, or they may arrive out of order. Some applications may be able to tolerate this, e.g., streaming music.

An application can choose how reliably its messages are transmitted by selecting a transport protocol:

- Transmission Control Protocol (TCP) ensures the reliable transmission of messages. TCP guarantees that messages are not lost or corrupted, and that messages are delivered in the correct order. However, this adds overhead to message transport.

- User Data Protocol (UDP) is a fast but unreliable method for transporting messages. Messages sent using UDP will not be corrupted, but they may be lost or duplicated, and they may arrive in an order different from the order in which they were sent.

Both TCP and UDP utilize the Internet Protocol (IP) to carry packets of data. The IP standard defines the packet format, which includes formats for specifying source and destination addresses and IO ports.

6.1.2 TCP Sockets in Java

The java.net class library provides classes *Socket* and *ServerSocket* for TCP-based message passing.

6.1.2.1 Class Socket.

A client's first step is to create a TCP socket and try to connect to the server:

```
InetAddress host;                        // server's machine address
int serverPort = 2020;                   // port number bound to server's socket
Socket socket;                           // client's socket
try {host = InetAddress.getByName("www.cs.gmu.edu"); }
catch (UnknownHostException) { … }
try {socket = new Socket(host,serverPort); }  // create a socket and request a
catch (IOException e) { … }                   // connection to host
```

The client assumes that the server is listening for TCP connection requests on port *serverPort*. The *Socket* constructor throws an *IOException* if it cannot make a connection.

Client creates an input stream to receive data from its socket and an output stream to send data to the socket at the server's end of the channel. (Looks like file I/O!)

```
PrintWriter toServer = new PrintWriter(socket.getOutputStream(),true);
BufferedReader fromServer = new BufferedReader(new
       inputStreamReader(socket.getInputStream()));
toServer.println("Hello");               // send a message to the server
String line = fromServer.readLine();     // receive the server's reply
System.out.println ("Client received: " + line + " from Server");
toServer.close(); fromServer.close(); socket.close();
```

A read operation on the *InputStream* associated with a *Socket* normally blocks. To set a one second timeout:

```
socket.setSoTimeout(1000);           // set a one second timeout
```

When the timeout expires, a *java.net.SocketTimeoutException* is raised and the *Socket* is still valid.

6.1.2.2 Class ServerSocket. The server begins by creating a *ServerSocket:*

```
int serverPort = 2020;
ServerSocket listen;
try { listen = new ServerSocket(serverPort); }
catch (IOException e) { … }
```

The server then calls the *accept*() method of the *ServerSocket*:

- method *accept*() waits until a client requests a connection,
- it then returns a *Socket* that connects the client to the server.
- server gets input and output streams and communicates with the client.
- client, server, or both, close the connection, and the server waits for a connection request from another client.

```
try {
   listen = new ServerSocket(serverPort);
   while (true) {
      Socket socket = listen.accept();   // wait for a client request
      toClient = new PrintWriter(socket.getOutputStream(),true);
      fromClient = new BufferedReader(new
           InputStreamReader(socket.getInputStream()));
      String line = fromClient.readLine();// receive a message from the client
      System.out.println("Server received " + line);
      toClient.println("Good-bye");  // send a reply to the client
   }
}
catch (IOException e) { … }
finally { if (listen != null) try { listen.close();}
                                   catch (IOException e) { e.printStackTrace(); }}
```

The server can create a separate thread to handle the client's requests.

```
Socket socket = listen.accept();
clientHandler c = new clientHandler(socket);  // clientHandler extends Thread
c.start();
```

The *run*() method of class *clientHandler* uses input and output streams obtained from the *socket* to communicate with the client, exactly as shown above.

Listing 6.2 shows a client and server program using TCP sockets:

- *Message* objects are serialized and passed between the client and the server.
- If the client program is executed using “java Client 2”, then the client will send the value 2 to the server, receive the value 4 from the server, and display the message “2 x 2 = 4”.

```
import java.net.*; import java.io.*;
public final class Client {
   public static void main (String args[]) {
      int serverPort = 2020; Socket socket = null;
      ObjectOutputStream toServer = null; ObjectInputStream fromServer=null;
      try {
         if (args.length != 1) {
            System.out.println("need 1 argument"); System.exit(1);
         }
         int number = Integer.parseInt(args[0]);
         // client and server run on the same machine, known as the Local Host
         InetAddress serverHost = InetAddress.getLocalHost();
         socket = new Socket(serverHost,serverPort);
         // send a value to the server
         toServer = new ObjectOutputStream(new BufferedOutputStream
               (socket.getOutputStream()));
         Message msgToSend = new Message(number);
         toServer.writeObject(msgToSend); toServer.flush();
         // This will block until the corresponding ObjectOutputStream in the
         // server has written an object and flushed the header.
         fromServer = new ObjectInputStream(new
         BufferedInputStream(socket.getInputStream()));
         Message msgFromReply = (Message) fromServer.readObject();
         System.out.println (number + " x " + number + " = " +
            msgFromReply.number);
      }
      catch (IOException e) { e.printStackTrace(); System.exit(1); }
      catch (ClassNotFoundException e) {e.printStackTrace(); System.exit(1); }
      finally { if (socket != null) try { socket.close();} catch (IOException e) {
         e.printStackTrace(); }}
   }
}
```

```
public final class Server {
   public static void main (String args[]) {
      int serverPort = 2020; ServerSocket listen=null;
      ObjectOutputStream toClient; ObjectInputStream fromClient;
      try {
         listen = new ServerSocket(serverPort);
         while (true) {
            Socket socket = listen.accept();
            toClient = new ObjectOutputStream(new BufferedOutputStream
                (socket.getOutputStream()));
            // This will block until the corresponding ObjectOutputStream in the
            // client has written an object and flushed the header.
            fromClient = new ObjectInputStream(new BufferedInputStream
                (socket.getInputStream()));
            Message msgRequest = (Message) fromClient.readObject();
            // compute a reply and send it back to the client
            int number = msgRequest.number;
            toClient.writeObject(new Message(number*number)); toClient.flush();
         } }
      catch (IOException e) {e.printStackTrace(); System.exit(1); }
      catch (ClassNotFoundException e) {e.printStackTrace(); System.exit(1); }
      finally { if (listen != null) try { listen.close();} catch (IOException e) {
         e.printStackTrace(); }}
   }
}
public final class Message implements Serializable {
   public int number;
   Message(int number) {this.number =  number;   }
}
```

Listing 6.2 Client and server using TCP sockets.

6.2 Java TCP Channel Classes

The custom TCP channel classes:

- provide asynchronous and synchronous message passing
- can be used with the selective wait statement presented in Chapter 5
- hide the complexity of using TCP
- support tracing, testing, and replay in a distributed environment.
- based on the notion of a mailbox that holds messages deposited by senders and withdrawn by receivers. (They are not based on the client and server paradigm.)

6.2.1 Classes *TCPSender* and *TCPMailbox*

Classes *TCPSender* and *TCPMailbox* provide asynchronous message passing using a buffer-blocking *send()* method and a blocking *receive()* method, respectively.

Listing 6.3 illustrates the use of these classes in a distributed solution to the bounded buffer problem.

- Program *Producer* creates a *TCPSender* object named *deposit* for sending items to program *Buffer*.
- Program *Buffer* creates a *TCPMailbox* object named *deposit* for receiving items from the *Producer*, and a *TCPSender* object named *withdraw* for sending items to the *Consumer*.
- Program *Buffer* acts as a one-slot bounded buffer, receiving items from the *Producer* and forwarding them to the *Consumer*.
- Program *Consumer* has a *TCPMailbox* object named *withdraw* to receive the messages sent by *Buffer*.

Method *connect()*:

- must be called before a *TCPSender* can be used to send messages.
- opens a TCP connection using the host and port number specified when the *TCPSender* object is constructed.

When there are no more messages to send, method *close()* is called and the TCP connection is closed.

Methods *send()* and *receive()* both operate on *messageParts* objects.

- A *messageParts* object packages the message to be sent with the return address of the sender. (The return address is optional.)
- The return address information includes the sender’s host address and a port number that the sender will use to wait for a reply.

A thread that calls *receive()* receives a *messageParts* object.

- The return address in this object can be saved and used later to reply to the sender.
- If no return address is needed, a *messageParts* object can be constructed without a return address.

```
import java.net.*;
import java.io.*;
public final class Producer {
    public static void main (String args[]) {
        final int bufferPort = 2020; String bufferHost = null;
        try {
            bufferHost = InetAddress.getLocalHost().getHostName();
            TCPSender deposit = new TCPSender(bufferHost,bufferPort);
            deposit.connect();
            for (int i=0; i<3; i++) {
                System.out.println("Producing" + i);
                messageParts msg = new messageParts(new Message(i));
                deposit.send(msg);
            }
            deposit.close();
        }
        catch (UnknownHostException e) {e.printStackTrace();}
        catch (TCPChannelException e) {e.printStackTrace();}
    }
}
public final class Buffer {
    public static void main (String args[]) {
        final int bufferPort = 2020; final int consumerPort = 2022;
        try {
            String consumerHost = InetAddress.getLocalHost().getHostName();
            TCPMailbox deposit = new TCPMailbox(bufferPort,"deposit");
            TCPSender withdraw = new TCPSender(consumerHost,consumerPort);
            withdraw.connect();
            for (int i=0; i<3; i++) {
                messageParts m = (messageParts) deposit.receive();
                withdraw.send(m);
            }
            withdraw.close(); deposit.close();
        }
        catch (UnknownHostException e) {e.printStackTrace();}
        catch (TCPChannelException e) {e.printStackTrace();}
    }
}
```

```
public final class Consumer {
   public static void main (String args[]) {
      final int consumerPort = 2022;
      try {
         TCPMailbox withdraw = new TCPMailbox(consumerPort,"withdraw");
         for (int i=0; i<3; i++) {
            messageParts m = (messageParts) withdraw.receive();
            Message msg = (Message) m.obj;
            System.out.println("Consumed " + msg.number);
         }
         withdraw.close();
      }
      catch (TCPChannelException e) {e.printStackTrace();}
   }
}
public final class messageParts implements Serializable {
   public Object obj;    // message to be sent
   public String host;   // host address of the sender
   public int port;      // port where sender will wait for a reply, if any
   messageParts(Object obj, String host, int port) {
      this.obj = obj;  this.host = host;  this.port = port;
   }
   // no return address
   messageParts(Object obj) {this.obj = obj;  this.host = "";  this.port = 0;}
}
```

Listing 6.3 Distributed bounded buffer using *TCPSender* and *TCPMailbox*.

Listing 6.4 shows classes *TCPSender* and *TCPMailbox*.

Class *TCPSender*:

- method *send()* is used to send messages to a particular *TCPMailbox*. The host address and port number of the destination *TCPMailbox* are specified when the *TCPSender* object is constructed.
- Method *connect()* is used to connect the *TCPSender* to the *TCPMailbox*. Once the connection is made, each call to *send()* uses the same connection.
- The connection is closed with a call to method *close ()*.

A single *TCPMailbox* object may receive connection requests from any number of *TCPSenders*:

- A *TCPMailbox* objects begins listening for connection requests when the *TCPMailbox* object is constructed.
- If multiple *TCPSenders* connect to the same *TCPMailbox*, the connections are handled concurrently.
- When *close()* is called on a *TCPMailbox* object, it stops listening for new connection requests.

Method *receive()* of class *TCPMailbox* returns a *messageParts* object:

- The *messageParts* object returned by *receive()* is withdrawn from a *messageBuffer* called *buffer*.
- A *messageBuffer* is simply a bounded buffer of *messagePart* objects implemented as an SU monitor

A *TCPMailbox* object is an "active object" -- during construction it automatically starts an internal thread to receive messages:

```
public class TCPMailbox implements Runnable {
  …
  public TCPMailbox( int port, String channelName ) throws
        TCPChannelException {
    this.port = port;
    this.channelName = channelName;
    try {listen = new ServerSocket( port ); }
    catch (IOException e) { e.printStackTrace();
        throw new TCPChannelException(e.getMessage());
    }
    buffer = new messageBuffer(100);
    Thread internal = new Thread(this);
    internal.start();   // internal thread executes method run()
  }
  …
}
```

The *run()* method of *TCPMailbox* accepts connection requests from *TCPSender* objects and starts a *connectionHandler* thread to handle the connection:

```
public void run() {
   while (true) {
      Socket socket = listen.accept(); // listen for senders
      (new connectionHandler(socket)).start();
   }
}
```

The *connectionHandler* thread obtains an input stream from the *socket*:

```
connectionHandler (Socket socket) throws IOException {
   this.socket = socket;
   from = new ObjectInputStream(socket.getInputStream());
}
```

and then uses the input stream to receive *messageParts* objects. The *messagePart* objects are deposited into the *messageBuffer*:

```
while (true) { // read objects until get EOF
   messageParts msg = null;
   try {
      msg = (messageParts) from.readObject();   // receive messageParts object
      buffer.deposit(msg);              // deposit messageParts object into buffer
   }
   catch (EOFException e) { break;}
}
```

If *buffer* becomes full, method *deposit()* will block. This will prevent any more *messagePart* objects from being received until a *messagePart* object is withdrawn using method *receive().*

Note: The preferred way to use a *TCPSender* object S is to issue an *S.close()* operation only after all the messages have been sent:

- An *S.connect()* operation appears at the beginning of the program and an *S.close()* operation appears at the end (refer back to Listing 6.3.).
- All the messages sent over S will use a single connection.
- If multiple *TCPSender* objects connect to the same *TCPMailbox*, the *TCPMailbox* will handle the connections concurrently.

6.2.2 Classes *TCPSynchronousSender* and *TCPSynchronousMailbox*

Classes *TCPSynchronousSender* and *TCPSynchronousMailbox* implement synchronous channels:

- method *send() of TCPSynchronousSender* waits for an acknowledgment that the sent message has been received by the destination thread.
- The *receive()* method of *TCPSynchronousMailbox* sends an acknowledgement when the message is withdrawn from the *messageBuffer*, indicating that the destination thread has received the message.

6.2.3 Class *TCPSelectableSynchronousMailbox*

A *TCPSelectableSynchronousMailbox* object is used just like a *selectableEntry* or *selectablePort* object in Chapter 5.

Listing 6.6 shows bounded buffer program *Buffer*, which uses a *selectiveWait* object and *TCPSelectableSynchronousMailbox* objects *deposit* and *withdraw*.

Notice that *Buffer* selects *deposit* and *withdraw* alternatives in an infinite loop. One way to terminate this loop is to add a *delay* alternative to the selective wait, which would give *Buffer* a chance to timeout and terminate after a period of inactivity.

Detecting the point at which a distributed computation has terminated is not trivial since no process has complete knowledge of the global state of the computation, and neither global time nor common memory exists in a distributed system.

```
public final class Buffer {
   public static void main (String args[]) {
       final int depositPort = 2020; final int withdrawPort = 2021;
       final int withdrawReplyPort = 2022; int fullSlots=0; int capacity = 2;
       Object[] buffer = new Object[capacity];   int in = 0, out = 0;
       try {
           TCPSelectableSynchronousMailbox deposit = new
              TCPSelectableSynchronousMailbox(depositPort);
           TCPSelectableSynchronousMailbox withdraw = new
           TCPSelectableSynchronousMailbox (withdrawPort);
           String consumerHost = InetAddress.getLocalHost().getHostName();
           TCPSender withdrawReply = new
           TCPSender(consumerHost,withdrawReplyPort);
           selectiveWait select = new selectiveWait();
           select.add(deposit);            // alternative 1
           select.add(withdraw);           // alternative 2
           while(true) {
              withdraw.guard(fullSlots>0);
              deposit.guard (fullSlots<capacity);
              switch (select.choose()) {
                  case 1:Object o = deposit.receive(); // item from Producer
                         buffer[in] = o; in = (in + 1) % capacity;    ++fullSlots;
                         break;
                  case 2:messageParts withdrawRequest = withdraw.receive();
                         messageParts m = (messageParts) buffer[out];
                         try {// send an item back to the Consumer
                            withdrawReply.send(m);
                         } catch (TCPChannelException e)
                               {e.printStackTrace();}
                            out = (out + 1) % capacity; --fullSlots;
                            break;
              }
           }
       }
       catch (InterruptedException e) {e.printStackTrace();System.exit(1);}
       catch (TCPChannelException e) {e.printStackTrace();System.exit(1);}
       catch (UnknownHostException e) {e.printStackTrace();}
   }
}
```
Listing 6.6 Using a *selectiveWait* statement in a distributed program.

6.3 Timestamps and Event Ordering

In a distributed environment, it is difficult to determine the execution order of events.

Distributed mutual exclusion: Distributed processes that need access to a shared resource must send each other requests to obtain exclusive access to the resource. Processes can access the shared resource in the order of their requests, but the request order is not easy to determine.

Event ordering is also a critical problem during testing and debugging.

- the event order observed during tracing must be consistent with an event order that actually occurred.
- reachability testing depends on accurate event ordering to identify concurrent events and generate race variants.

6.3.1 Event Ordering Problems

Consider the following program, which uses asynchronous communication:

```
Thread1                  Thread2          Thread3
(a) send A to Thread2;  (b) receive X;   (d) send B to Thread2;
                        (c) receive Y;
```

The possible executions of this program are represented by diagrams (D1) and (D2) in Fig. 6.7.

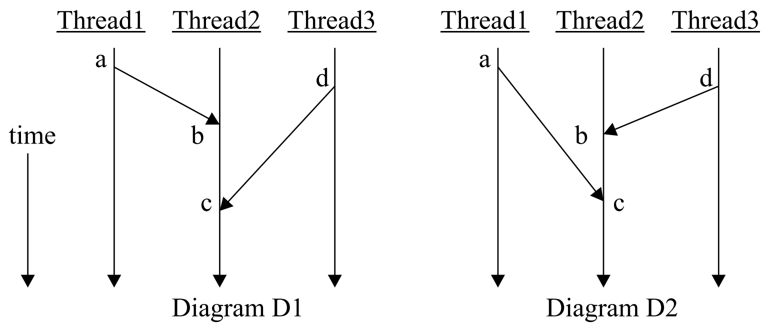


Figure 6.7 Diagrams D1 and D2.

Assume that the threads in diagram D1 send asynchronous trace messages to the controller whenever they execute a message passing event.

Fig. 6.8 illustrates two “observability problems” that can occur when the controller relies on the arrival order of the trace messages to determine the order of events.

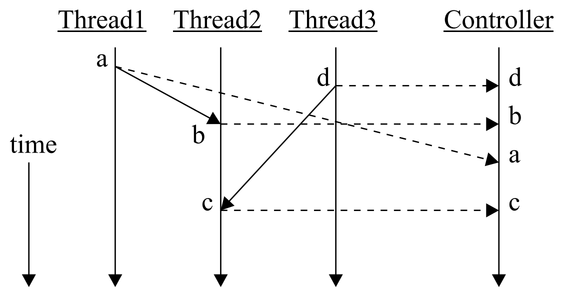


Figure 6.8 Observability Problems for Diagram D1

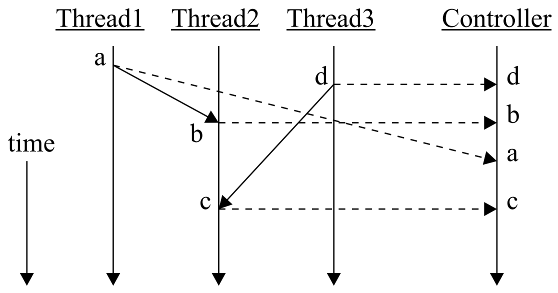


Figure 6.8 Observability Problems for Diagram D1

Incorrect orderings: The controller observes event *b* occur before event *a*, which is not what happened.

Arbitrary orderings:

- The controller observes event *d* occur before event *b*. Since *d* and *b* are concurrent events they can occur in either order, and the order the controller will observe is non-deterministic.
- The controller cannot distinguish non-deterministic orderings from orderings that are enforced by the program.
  - the programmer may mistakenly conclude that *d must* precede *b*.
  - the programmer may feel the need to create a test case where the order of events *d* and *b* is reversed, even though this change is not significant.

The controller can use timestamps to accurately order events. In Table 6.1 An 'X' indicates that a timestamp mechanism has a particular observability problem.

Observability Problems	**Timestamp Mechanisms**				
	Arrival order	Local real-time clocks	Global real-time clock	Totally-ordered logical clocks	Partially-ordered logical clocks
Incorrect orderings	X	X			
Arbitrary orderings	X	X	X	X	

Table 6.1 Effectiveness of timestamping mechanisms [Fidge 1996]. An 'X' indicates that a problem exists.

6.3.2 Local Real-Time Clocks

The real-time clock available on each processor can be used as the source of the timestamp.

Since the real-time clocks on different processors are not synchronized, incorrect orderings may be seen, and concurrent events are arbitrarily ordered.

Fig. 6.9 shows two ways in which the events in diagram D1 of Fig. 6.7 can be timestamped.

- On the left, the clock of Thread 1's processor is ahead of the clock of Thread 2's processor so event *b* erroneously appears to occur before event *a*.
- The ordering of events *d* and *b* is arbitrary, and depends on the relative speeds of the threads and the amount by which the processor's real-time clocks differ.

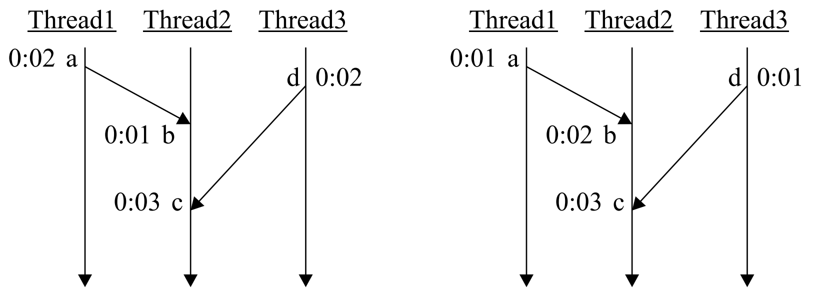


Figure 6.9 Timestamps using unsynchronized local real-time clocks.

6.3.3 Global Real-Time Clocks

If local real-time clocks are synchronized, there is a global reference for real time.

- avoids incorrect orderings, but as shown in Fig. 6.10, arbitrary orderings are still imposed on concurrent events *d* and *b*.
- Sufficiently accurate clock synchronization is difficult and sometimes impossible to achieve.

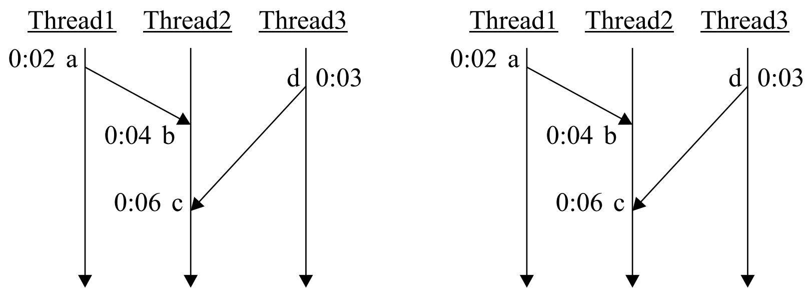


Figure 6.10 Timestamps using a real-time global clock.

6.3.4 Causality

The remaining two schemes use logical clocks instead of real-time clocks.

Logical clock schemes rely on the semantics of program operations to determine whether one event occurred before another event:

- if events A and B are local events in the same thread and A is executed before B, then A's logical timestamp will indicate that A happened before B.
- if S is an asynchronous send event in one thread and R is the corresponding receive event in another thread, then S's logical timestamp will indicate that S occurred before R.

(More accurately, S occurred before the *completion* of R, since the send operation S might have occurred long after the receive operation R started waiting for a message to arrive.)

It is important that any event ordering is consistent with the cause and effect relationships between the events.

The causality or "happened before" relation "$\Rightarrow$" for an execution of a message-passing program is defined as follows:

(C1) If events e and f are events in the same thread and e occurs before f, then $e \Rightarrow f$.

(C2) If there is a message $e \rightarrow f$ (i.e., e is a non-blocking send and f is the corresponding receive), then $e \Rightarrow f$.

(C3) If there is a message $e \leftrightarrow f$ or $f \leftrightarrow e$ (i.e., one of e or f is a blocking send and the other is the corresponding blocking receive), then for event g such that $e \Rightarrow g$, we have $f \Rightarrow g$, and for event h such that $h \Rightarrow f$, we have $h \Rightarrow e$.

(C4) If $e \Rightarrow f$ and $f \Rightarrow g$, then $e \Rightarrow$g. (Thus, "$\Rightarrow$" is transitive.)

It is easy to visually examine a space-time diagram and determine the causal relations:

> For two events *e* and *f* in a space-time diagram, $e \Rightarrow f$ if and only if there is no message $e \leftrightarrow f$ or $f \leftrightarrow e$ and there exists a path from *e* to *f* that follows the vertical lines and arrows in the diagram.

(A double headed arrow allows a path to cross in either direction.).

For events *e* and *f* of an execution, if neither $e \Rightarrow f$ nor $f \Rightarrow e$, then *e* and *f* are said to be concurrent, denoted as $e \| f$:

- if there is a message $e \leftrightarrow f$ or $f \leftrightarrow e$, then *e* and *f* are concurrent events.
- Since $e \| f$ and $f \| e$ are equivalent, the "||" relation is symmetric.
- the "||" relation is not transitive.

In diagram (D1), $a \rightarrow b$, $b \rightarrow c$, $d \rightarrow c$, $a \rightarrow c$, $a \| d$, and $d \| b$, but *a* and *b* are not concurrent events.

In diagram (D2), $a \rightarrow c$, $b \rightarrow c$, $d \rightarrow c$, $d \rightarrow c$, b||*a*, and $a \| d$, but *b* and *d* are not concurrent events.

6.3.5 Integer Timestamps

Each event receives a logical (integer) timestamp and these timestamps are used to order the events.

Consider a message-passing program that uses asynchronous communication and contains threads Thread_1, Thread_2, ..., and Thread_n. Thread_i, $1 \leq i \leq n$, contains a logical clock C_i, which is simply an integer variable initialized to 0.

During execution, logical time flows as follows:

(IT1) Thread$_i$ increments C$_i$ by one immediately before each event it executes.

(IT2) When Thread$_i$ sends a message, it also sends the value of C$_i$ as the timestamp for the send event.

(IT3) When Thread$_i$ receives a message with *ts* as its timestamp, if $ts \geq C_i$, then Thread$_i$ sets C$_i$ to *ts*+1 and assigns *ts*+1 as the timestamp for the receive event. Hence, $C_i = \max(C_i, ts+1)$.

Diagram (D3) in Fig. 6.11 represents an execution of three threads that use asynchronous communication.

Notice that the integer timestamp for event *v* is less than the integer timestamp for event *b*, but $v \Rightarrow b$ does not hold. (There is no path from *v* to *b* in diagram D3.)

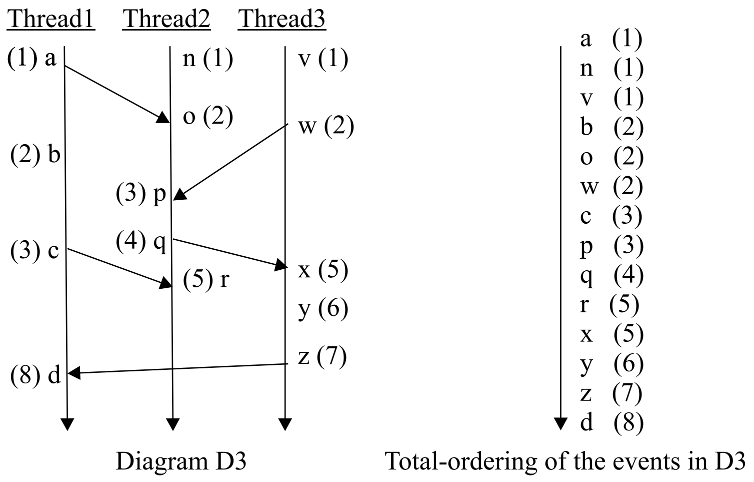


Figure 6.11 Diagram D3 and a total-ordering for D3.

Denote the integer timestamp recorded for event *e* as IT(e) and let *s* and *t* be two events of an execution.

- If $s \Rightarrow t$, then IT(*s*) will definitely be less than IT(*t*).
- The fact that IT(*s*) is less than IT(*t*) does not imply that $s \Rightarrow t$.
- If *s* and *t* are concurrent, then their timestamps will be consistent with one of their two possible causal orderings.

⇒ we cannot determine whether or not $s \Rightarrow t$ by using the integer timestamps recorded for *s* and *t*.

Although integer timestamps cannot tell us the causality relationships that hold between the events, we can use integer timestamps to produce one or more total orders that preserve the causal order:

- Order the events in ascending order of their integer timestamps. For the events that have the same integer timestamp, break the tie in some consistent way.
- A method for tie breaking: order events with the same integer timestamps in increasing order of their thread identifiers (refer again to Fig. 6.11).

From Table 6.1:

- integer timestamps avoid incorrect orderings
- but independent events are arbitrary ordered.

6.3.6 Vector Timestamps

Integer timestamps cannot be used to determine that two events are *not* causally related.

To do this, each thread maintains a vector clock, which is a vector of integer clock values. The vector clock for Thread_i, $1 \leq i \leq n$, is denoted as VC_i, where $VC_i[j]$, $1 \leq j \leq n$, refers to the jth element of vector clock VC_i.

During execution, vector time is maintained as follows:

(VT1) Thread_i increments $VC_i[i]$ by one before each event of Thread_i.

(VT2) When Thread_i executes a non-blocking send, it sends the value of its vector clock VC_i as the timestamp for the send operation.

(VT3) When Thread_i receives a message with timestamp VT_m from a non-blocking send of another thread, Thread_i sets VC_i to the maximum of VC_i and VT_m and assigns VC_i as the timestamp for the receive event. Hence, $VC_i = \max(VC_i, VT_m)$. That is: for (k = 1; k<= n; k++) $VC_i[k] = \max(VC_i[k], VT_m[k])$;

(VT4) When one of Thread_i or Thread_j executes a blocking send that is received by the other, Thread_i and Thread_j exchange their vector clock values, set their vector clocks to the maximum of the two vector clock values, and assign their new vector clocks (which now have the same value) as the timestamps for the send and receive events. Hence, Thread_i performs the following operations:

- Thread_i sends VC_i to Thread_j and receives VC_j from Thread_j;
- Thread_i sets $VC_i = \max(VC_i, VC_j)$;
- Thread_i assigns VC_i as the timestamp for the send or receive event that Thread_i performed.

Thread_j performs similar operations.

The value $VC_i[j]$, where $j \neq i$, denotes the best estimate $Thread_i$ is able to make about $Thread_j$'s current logical clock value VCj[j], i.e., the number of events in $Thread_j$ that $Thread_i$ "knows about" through direct communication with $Thread_j$ or through communication with other threads that have communicated with $Thread_j$ and $Thread_i$.

Examples:

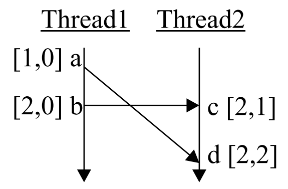


Figure 6.12 Diagram D4.

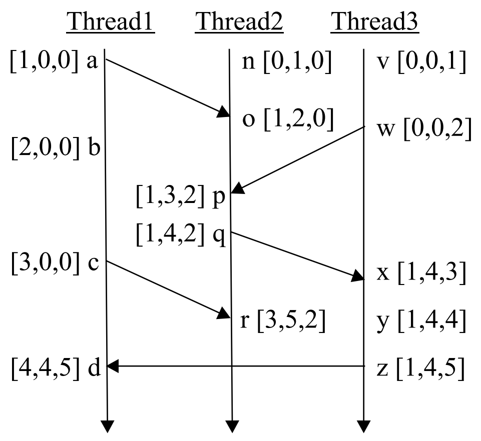


Figure 6.13 Diagram D5.

Rule HB1:

- Denote the vector timestamp recorded for event *e* as VT(e).
- For a given execution, let e_i be an event in Thread_i and e_j an event in (possibly the same thread) Thread_j.
- Threads are permitted to use asynchronous or synchronous communication, or a mix of both.
- $e_i \Rightarrow e_j$ if and only if there exists a path from e_i to e_j in the space-time diagram of the execution and there is no message $e_i \leftrightarrow e_j$ or $e_j \leftrightarrow e_i$. Thus:

  (HB1) $e_i \Rightarrow e_j$ if and only if (for $1 \le k \le n$, $VT(e_i)[k] \le VT(e_j)[k]$) and ($VT(e_i) \ne VT(e_j)$).

Note that if there is a message $e_i \leftrightarrow e_j$ or $e_j \leftrightarrow e_i$ then $VT(e_i) = VT(e_j)$ and (HB1) cannot be true.

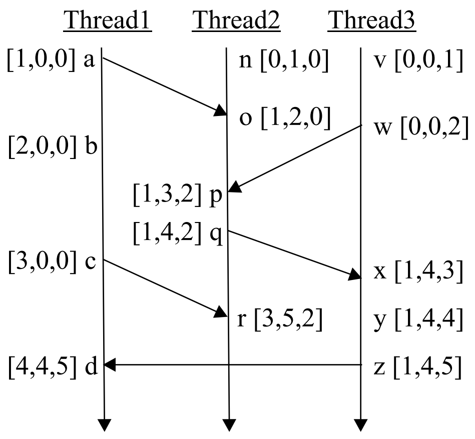


Figure 6.13 Diagram D5.

Rule HB2: Actually, we only need to compare two pairs of values, as the following rule shows:

(HB2) $e_i \Rightarrow e_j$ if and only if $(VT(e_i)[i] \leq VT(e_j)[i])$ and $(VT(e_i)[j] < VT(e_j)[j])$.

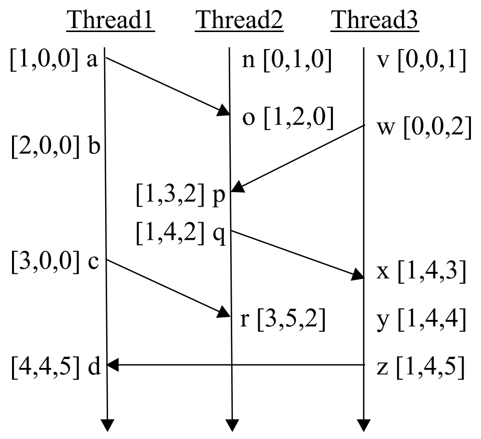


Figure 6.13 Diagram D5.

If there is a message $e_i \leftrightarrow e_j$ or $e_j \leftrightarrow e_i$ then $e_i \Rightarrow e_j$ is not true and $(VT(e_i)[j] < VT(e_j)[j])$ cannot be true (since the timestamps of e_i and e_j will be the same). This is also true if e_i and e_j are the same event.

In Fig. 6.13, events *v* and *p* are in Thread3 and Thread2, respectively, where $VT(v) = [0,0,1]$ and $VT(p) = [1,3,2]$:

- Since $(VT(v)[3] \leq VT(p)[3])$ and $(VT(v)[2] < VT(p)[2])$ we conclude $v \Rightarrow p$.
- For event *w* in Thread3 we have $VT(w) = [0,0,2]$. Since there is a message $w \leftrightarrow p$, the timestamps for *w* and *p* are the same, which means that $VT(v)[3] \leq VT(w)[3]$ must be true and that $VT(v)[2] < VT(w)[2]$ cannot be true.

  Hence, $w \Rightarrow p$ is not true, as expected.

In general, suppose that the value of VT(e_i)[j] is *x* and the value of VT(e_j)[j] is *y*.

- the only way for VT(e_i)[j] < VT(e_j)[j] to be false is if $Thread_i$ knows (through communication with $Thread_j$ or with other threads that have communicated with $Thread_j$) that $Thread_j$ already performed its x^{th} event, which was either e_j or an event that happened after e_j (as $x \geq y$).
- In either case, $e_i \Rightarrow e_j$ can't be true (otherwise, we would have $e_i \Rightarrow e_j \Rightarrow e_i$, which is impossible).

Rule HB3: If events e_i and e_j are in different threads and only asynchronous communication is used, then the rule can be further simplified to:

(HB3) $e_i \Rightarrow e_j$ if and only if VT(e_i)[i] ≤ VT(e_j)[i].

Example:

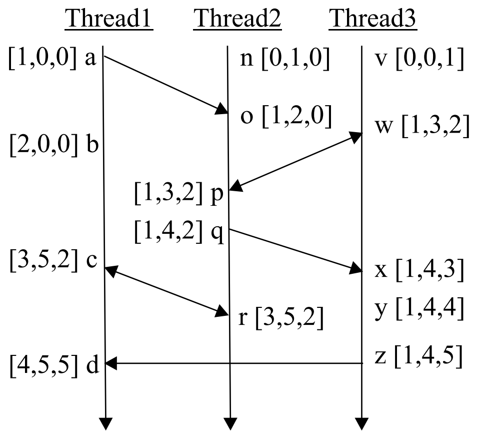


Figure 6.14 Diagram D6.

Referring to Table 6.1, using vector timestamps avoids incorrect orderings and independent events are not arbitrarily ordered.

6.3.7 Timestamps for Programs using Messages and Shared Variables

In programs that use both message passing and shared variable communication, vector timestamps must be assigned to send and receive events and also to events involving shared variables, such as read and write events or entering a monitor.

For two read/write events *e* and *f* on a shared variable V, let $e \xrightarrow{V} f$ denote that event *e* occurs before *f* on V.

(C1–C4) Same as rules C1 – C4 above for send and receive events.

(C5) For two different events *e* and *f* on shared variable V such that at least one of them is a write event, if $e \xrightarrow{V} f$ then $e \Rightarrow f$.

Timestamps for send and receive events are assigned as described earlier.

For each shared variable V, we maintain two vector timestamps:

- VT_LastWrite(V): contains the vector timestamp of the last write event on V, initially all zeros.
- VT_Current(V): is the current vector clock of V, initially all zeros.

When Thread_i performs an event *e*:

(VT1–VT4) If *e* is a send or receive event, same as (VT1–VT4) in Section 6.3.6

(VT5) If *e* is a write operation on shared variable V, Thread_i performs the following operations after performing write operation *e*:

(VT5.1) VC_i = max(VC_i, VT_Current(V))

(VT5.2) VT_LastWrite(V) = VC_i

(VT5.3) VT_Current(V) = VC_i

(VT6) If *e* is a read operation on shared variable V, Thread_i performs the following operations after performing read operation *e*:

(VT6.1) VC_i = max(VC_i, VT_LastWrite(V))

(VT6.2) VT_Current(V) = max(VC_i, VT_Current(V))

For write event *e* in rule (VT5), VC_i is set to max(VC_i, VT_Current(V)).
For read event *e* in rule (VT6), VC_i is set to max(VC_i, VT_LastWrite(V)).

The reason for the difference is the following.

- A write event on V causally precedes all the read and write events on V that follow it.
- A read event on V is concurrent with other read events on V that happen after the most recent write event on V and happen before the next write event, provided that there are no causal relations between these read events due to messages or accesses to other shared variables.

Example:

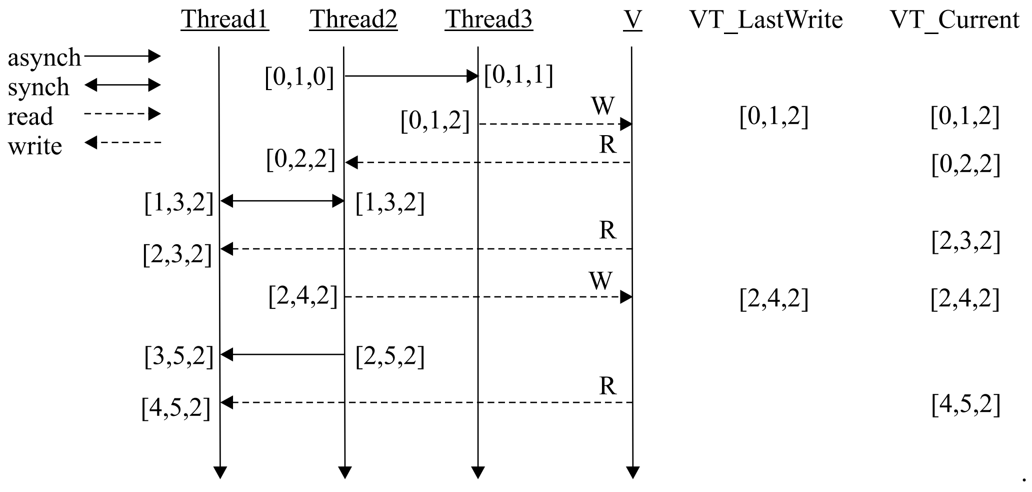


Figure 6.15 Timestamps for a program that uses message passing and shared variables.

6.4 Message-Based Solutions to Distributed Programming Problems

Distributed mutual exclusion and distributed readers and writers: Both problems involve events that must be totally-ordered, such as requests to read or write, or requests to enter a critical section. Requests are ordered using integer timestamps (Section 6.3.5).

Alternating bit protocol (ABP). The ABP is used to ensure the reliable transfer of data over faulty communication lines. Protocols similar to ABP are used by TCP.

6.4.1 Distributed Mutual Exclusion

When a process wishes to enter its critical section, it requests and waits for permission from all the other processes. When a process receives a request:

- if the process is not interested in entering its critical section, the process gives its permission by sending a reply as soon as it receives the request.
- if the process does want to enter its critical section, then it may defer its reply, depending on the relative order of its request among requests by other processes.

Requests are ordered based on a sequence number that is associated with each request and a sequence number maintained locally by each process:

- Sequence numbers are essentially integer timestamps.
- If a process receives a request having a sequence number that is the same as the local sequence number of the process, then the tie is resolved in favor of the process with the lowest ID.
- Since requests can be totally ordered using sequence numbers and IDs, there is always just one process that can enter its critical section next.

In program *distributedMutualExclusion* in Listing 6.16, each *distributedProcess* is started with a user assigned ID:

- assume there are three *distributedMutualExclusion* programs running at the same time, on the same computer.
- *numberOfProcesses* is three, and the IDs of the processes are 0, 1, and 2.

Each *distributedProcess* uses an array of three *TCPSender* objects for sending requests, and an array of three *TCPSender* objects for sending replies:

- When a *TCPSender* object is constructed, it is associated with the host address and port number of a *TCPMailbox* object owned by one of the distributed processes.
- All messages sent through the *TCPSender* object are addressed to the associated *TCPMailbox*.

Each *distributedProcess* uses *TCPMailbox* objects named *receiveRequests* and *receiveReplies* to receive request and reply messages from the other processes.

Connections between all the *TCPSender* and *TCPMailbox* objects are made by calling the *connect()* method of each *TCPSender* object at the start of the *run()* method for each *distributedProcess*.

The port numbers used for *TCPMailbox* objects *receiveRequests* and *receiveReplies* are as follows:

- *distributedProcess0* uses 2020 and 2021 for its two *TCPMailbox* objects
- *distributedProcess1* uses 2022 and 2023 for its two *TCPMailbox* objects
- *distributedProcess2* uses 2024 and 2025 for its two *TCPMailbox* objects.

Example:

- *distributedProcess0* sends requests ports 2022 and 2024
- Requests from the other two processes to *distributedProcess0* are addressed to port 2020

- Replies from the other processes to *distributedProcess0* are addressed to 2021
- Replies from *distributedProcess0* to *distributedProcess1* and *distributedProcess2* are addressed to ports 2023 and 2025, respectively.

When a *distributedProcess* wants to enter its critical section in method *run()* it:

- computes a sequence number
- sets flag *requestingOrExecuting* to true
- sends a request to each of the other processes
- waits for each of the processes to reply.

Each *distributedProcess* has a *Helper* thread that handles requests received from the other processes:

- If the *Helper* for *distributedProcess i* receives a *requestMessage* from *distributedProcess j*, the *Helper* replies immediately if
  - the sequence number in *j*'s request < the sequence number stored at *i*, or
  - if d*istributedProcess i* is not trying to enter its critical section.
- The *Helper* defers the reply if *distributedProcess i* is in its critical section, or if *distributedProcess i* wants to enter its critical section and the *requestMessage* from *distributedProcess j* has a higher sequence number.
- If the sequence numbers are the same, then the tie is broken by comparing process identifiers. (Each request message contains a sequence number and the identifier of the sending process.)

When a *distributedProcess* sends its request, it computes a sequence number by adding one to the highest sequence number it has received in requests from other processes.

Class *Coordinator* is a monitor that synchronizes a *distributedProcess* thread and its *Helper* thread.

```
import java.io.*; import java.net.*;
class distributedMutualExclusion {
   public static void main(String[] args) {
      if (args.length != 1) {
         System.out.println("need 1 argument: process ID (IDs start with 0)");
         System.exit(1);
      }
      int ID = Integer.parseInt(args[0]); new distributedProcess(ID).start();
   }
}
class distributedProcess extends TDThreadD {
   private int ID;                  // process ID
   private int number;  // the sequence number sent in request messages
   private int replyCount;          // number of replies received so far
   final private int numberOfProcesses = 4;
   final private int basePort = 2020;    String processHost = null;
   // requests sent to other processes
   private TCPSender[] sendRequests = new TCPSender[numberOfProcesses];
   private TCPSender[] sendReplies = new TCPSender[numberOfProcesses];
   private TCPMailbox receiveRequests = null;
   private TCPMailbox receiveReplies = null;
   private boolean[] deferred = null; // true means reply was deferred
   private Coordinator C; // monitor C coordinates distributedProcess and helper
   private Helper helper; // manages incoming requests for distributedProcess
   distributedProcess(int ID) {
      this.ID = ID;
      try {processHost = InetAddress.getLocalHost().getHostName(); }
      catch (UnknownHostException e) {e.printStackTrace();System.exit(1);}
      for (int i = 0; i < numberOfProcesses; i++) {
         sendRequests[i] = new TCPSender(processHost,basePort+(2*i));
         sendReplies[i] = new TCPSender(processHost,basePort+(2*i)+1);
      }
      receiveRequests = new TCPMailbox(basePort+(2*ID),"RequestsFor"+ID);
      receiveReplies = new TCPMailbox(basePort+(2*ID)+1,"RepliesFor"+ID);
      C = new Coordinator();    deferred = new boolean[numberOfProcesses];
      for (int i=0; i<numberOfProcesses; i++) deferred[i] = false;
      helper = new Helper(); helper.setDaemon(true); // start helper in run() method
   }
   class Helper extends TDThreadD {
   // manages requests from other distributed processes
      public void run() { // handle requests from other distributedProcesses
         while (true) {
            messageParts msg = (messageParts) receiveRequests.receive();
            requestMessage m = (requestMessage) msg.obj;
```

```
          if (!(C.deferrMessage(m))) { // if no deferral, then send a reply
             messageParts reply = new messageParts(new Integer(ID),
                processHost,basePort+(2*ID));
             sendReplies[m.ID].send(reply);
          }
       }
    }
}
class Coordinator extends monitorSC {
// Synchronizes the distributed process and its helper
   // requestingOrExecuting true if in or trying to enter CS
   private boolean requestingOrExecuting = false;
   private int highNumber; // highest sequence number seen so far
   public boolean deferrMessage(requestMessage m) {
      enterMonitor("decideAboutDeferral");
      highNumber = Math.max(highNumber,m.number);
      boolean deferMessage = requestingOrExecuting &&
         ((number < m.number) ||| (number == m.number && ID < m.ID));
      if (deferMessage)
         deferred[m.ID] = true; // remember that the reply was deferred
      exitMonitor();
      return deferMessage;
   }


   public int chooseNumberAndSetRequesting() {
   // choose sequence number and indicate process has entered or is
   // requesting to enter critical section
      enterMonitor("chooseNumberAndSetRequesting");
      requestingOrExecuting = true;
      number = highNumber + 1; // get the next sequence number
      exitMonitor();
      return number;
   }
   public void resetRequesting() {
      enterMonitor("resetRequesting");
      requestingOrExecuting = false;
      exitMonitor();
   }
}
```

```
public void run() {
   int count = 0;
   try {Thread.sleep(2000);}  // give other processes time to start
   catch (InterruptedException e) {e.printStackTrace();System.exit(1);}
   System.out.println("Process " + ID + " starting");
   helper.start();
   for (int i = 0; i < numberOfProcesses; i++) {
   // connect to the mailboxes of the other processes
      if (i != ID) {sendRequests[i].connect(); sendReplies[i].connect();}
   }
   while (count++<3) {
      System.out.println(ID + " Before Critical Section"); System.out.flush();
      int number = C.chooseNumberAndSetRequesting();
      sendRequests(); waitForReplies();
      System.out.println(ID + " Leaving Critical Section-"+count);  System.out.flush();
      try {Thread.sleep(500);} catch (InterruptedException e) {}
      C.resetRequesting();
      replytoDeferredProcesses();
   }
   try {Thread.sleep(10000);}  // let other processes finish
   catch (InterruptedException e) {e.printStackTrace();System.exit(1);}
   for (int i = 0; i < numberOfProcesses; i++)  // close connections
      if (i != ID) {sendRequests[i].close(); sendReplies[i].close();}
}
public void sendRequests() {
   replyCount = 0;
   for (int i = 0; i < numberOfProcesses; i++) {
      if (i != ID) {
         messageParts msg = new messageParts(new requestMessage(number,ID));
         sendRequests[i].send(msg); // send sequence number and process ID
         System.out.println(ID + " sent request to Thread " + i);
         try {Thread.sleep(1000);} catch (InterruptedException e) {}
      }
   }
}
public void replytoDeferredProcesses() {
   System.out.println("replying to deferred processes");
   for (int i=0; i < numberOfProcesses; i++)
      if (deferred[i]) {
         deferred[i] = false; // ID sent as a convenience for identifying sender
         messageParts msg = new messageParts(new Integer(ID));
         sendReplies[i].send(msg);
      }
}
```

```
    public void waitForReplies() {    // wait for all the other processes to reply
       while (true) {
          messageParts m = (messageParts) receiveReplies.receive();
          // ID of replying thread is available but not needed
          int receivedID = ((Integer)m.obj).intValue();
          replyCount++;
          if (replyCount == numberOfProcesses-1)
             break;              // all replies have been received
       }
    }
}
class requestMessage implements Serializable {
    public int ID;               // process ID
    public int number;           // sequence number (integer timestamp)
    public requestMessage(int number, int ID) {
        this.ID = ID; this.number = number;}
}
```

Listing 6.16 Permission-based algorithm for distributed mutual exclusion.

6.4.2 Distributed Readers and Writers

We implement strategy R=W.2, which allows concurrent reading and gives readers and writers equal priority.

Mutual exclusion is provided using the permission-based distributed mutual exclusion algorithm described previously.

When a process wants to perform its read or write operation:

- It sends a request to each of the other processes and waits for replies.
- A request consists of the same pair of values (sequence number and ID) used in the mutual exclusion algorithm, along with a flag that indicates the type of operation (read or write) being requested.
- When process *i* receives a request from process *j*, process *i* sends *j* an immediate reply if:
  - process *i* is not executing or requesting to execute its read or write operation
  - process *i* is executing or requesting to execute a "compatible" operation. Two read operations are compatible, but two write operations, or a read and a write operation, are not compatible.
  - process *i* is also requesting to execute a non-compatible operation but process *j*'s request has priority over process *i*'s request.

The differences between *distributedReadersAndWriters* and program *distributedMutualExclusion* in Listing 6.16:

1. Each *distributedProcess* is either a reader or a writer.

```
java distributedReadersAndWriters 0 Reader      // Reader process with ID 0
```

2. In method *decideAboutDeferral()* (Listing 6.17)

- The flag *requestingOrExecuting* is true when a *distributedProcess* is requesting to execute or is executing its read or write operation.
- The *requestMessage m* contains the type of *operation* being requested, and the sequence *number* and *ID* of the requesting process.
- If the receiving process is executing its operation or is requesting to execute its operation, method *compatible*() is used to determine whether or not the two operations are compatible.

Note: if process *i* is executing read or write operation OP when it receives a request *m* from process *j*, then the condition (*number* < *m.number*) in method *decideAboutDeferral()* must be true. That is:

- process *j* must have already received process *i*'s request for operation OP, sent its permission to *i*, and updated its sequence number to be higher than the number in process *i*'s request.
- In this case, if process *j*'s operation is not compatible, it will definitely be deferred.

```
public void decideAboutDeferral(requestMessage m) {
    enterMonitor();
    highNumber = Math.max(highNumber,m.number);
    boolean deferMessage = requestingOrExecuting &&
       !compatible(m.operation) && ((number < m.number) ||
           (number == m.number && ID < m.ID));
    if (deferMessage) {
       deferred[m.ID] = true;
    }
    else {
       messageParts msg = new messageParts(new Integer(ID));
       sendReplies[m.ID].send(msg);
    }
    exitMonitor();
}

private boolean compatible(int requestedOperation) {
    if (readerOrWriter == READER && requestedOperation == READER)
       return true;   // only READER operations are compatible
    else // READER/WRITER and WRITER/WRITER operations incompatible
       return false;
}
```

Listing 6.17 Methods *decideAboutDeferral()* and *compatible()* in program *distributedReadersAndWriters*.

6.4.3 Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is designed to ensure the reliable transfer of data over an unreliable communication medium.

The name of the protocol refers to the method used - messages are sent tagged with the bits 1 and 0 alternately, and these bits are also sent as acknowledgments.

Listing 6.18 shows classes *ABPSender* and *ABPReceiver* and two client threads.

- Thread *client1* is a source of messages for an *ABPSender* thread called *sender*.
- The *sender* receives messages from *client1* and sends the messages to an *ABPReceiver* thread called *receiver*.
- The *receiver* thread passes each message it receives to thread *client2*, which displays the message.

We assume that messages sent between an *ABPSender* and an *ABPReceiver* will not be corrupted, duplicated, or reordered, but they may be lost. The ABP will handle the detection and retransmission of lost messages.

An *ABPSender S* works as follows.

- After accepting a message from its client, *S* sends the message and sets a timer.
- *S* appends a one-bit sequence number (initially 1) to each message it sends out.
- There are then three possibilities:
  - *S* receives an acknowledgement from *ABPReceiver R* with the same sequence number. If this happens, the sequence number is incremented (modulo 2), and *S* is ready to accept the next message from its client.
  - *S* receives an acknowledgment with the wrong sequence number. In this case *S* resends the message (with the original sequence number), sets a timer, and waits for another acknowledgement from *R*.
  - *S* gets a timeout from the timer while waiting for an acknowledgement. In this case, *S* resends the message (with the original sequence number), sets a timer, and waits for an acknowledgement from *R*.

An *ABPReceiver R* receives a message and checks that the message has the expected sequence number (initially 1). There are two possibilities:

- *R* receives a message with a sequence number that matches the sequence number that R expects. If this happens,
  - R delivers the message to its client and sends an acknowledgement to *S*. The acknowledgment contains the same sequence number that *R* received.
  - *R* then increments the expected sequence number (modulo 2) and waits for the next message.
- *R* receives a message but the sequence number does not match the sequence number that *R* expects. In this case,
  - *R* sends *S* an acknowledgement that contains the sequence number that R received (i.e, the unexpected number),
  - R waits for *S* to resend the message.

Note that in both cases, the acknowledgement sent by R contains the sequence number that R received.

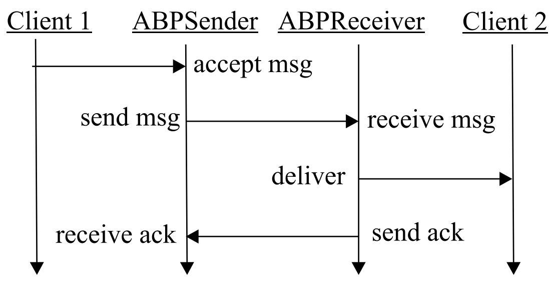
Communication between the sender and its client, and the receiver and its client, is through shared *link* (Chapter 5) channels.

The sender and receiver threads communicate using *TCPSender* and *TCPMailbox* objects, and a selectable asynchronous mailbox called *TCPSelectableMailbox.*

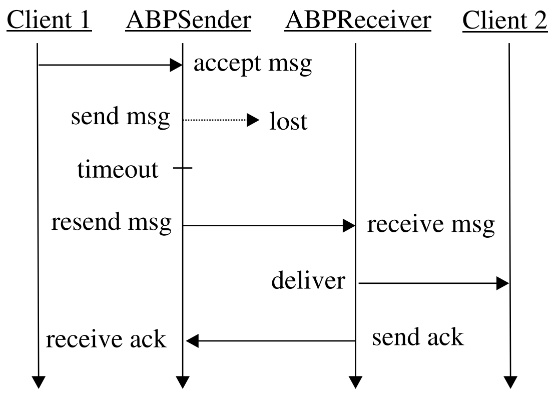
The *delayAlternative* in the *sender's* selective wait statement allows the *sender* to timeout while it is waiting to receive an acknowledgement from the *receiver*.

TCPUnreliableMailbox and *TCPUnreliableSelectableMailbox* randomly discard messages and acknowledgements sent between the *sender* and *receiver*.

Fig. 6.19 shows one possible flow of messages through the ABP program. In this scenario, no message or acknowledgement is lost.

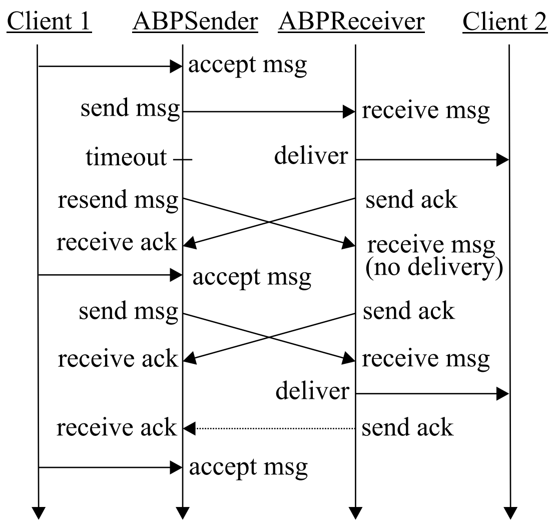


In Fig. 6.20, the *ABPSender's* message is lost. In this case, the *ABPSender* receives a timeout from its timer and resends the message.



In Fig. 6.21, the *ABPSender's* message is not lost, but the *ABPSender* still receives a timeout before it receives an acknowledgement. In this case,

- the *ABPReceiver* delivers and acknowledges the first message but only acknowledges the second.
- The *ABPSender* receives two acknowledgements, but it ignores the second one.



7. Testing and Debugging Concurrent Programs

The purpose of testing is to find program failures => A successful test is a test that causes a program to fail.

Ideally, tests are designed *before* the program is written.

The conventional approach to testing a program:

- execute the program with each selected test input once
- compare the test results with the expected results.

The term *failure* is used when a program produces unexpected results.

A failure is an observed departure of the external result of software operation from software requirements or user expectations [IEE90].

Failures can be caused by hardware or software faults.

Ways in which concurrent programs can fail: deadlock, livelock, starvation, and data races.

A software *fault* (or "bug") is a defective, missing, or extra instruction, or a set of related instructions that is the cause of one or more actual or potential failures.

Example: an error in writing an if-else statement may result in a fault that causes an execution to take a wrong branch:

- If this execution produces the wrong result, then it is said to fail;
- otherwise, the result is "coincidentally correct", and the internal state and path of the execution must be examined to detect the error.

If a test input causes a program to fail, the program is executed again, with the same input, in order to collect debugging information.

Debugging is the process of locating and correcting faults.

Since it is not possible to anticipate the information that will be needed to pinpoint the location of a fault, debugging information is collected and refined over the course of many executions until the problem is understood.

Regression testing: After the fault has been located and corrected, the program is executed again with each of the previously tested inputs to verify that the fault has been corrected and that in doing so no new faults have been introduced.

This cyclical process of testing, followed by debugging, followed by more testing, breaks down when it is applied to concurrent programs.

Let CP be a concurrent program. Multiple executions of CP with the *same* input may produce *different* results. This non-deterministic execution behavior creates the following problems during the testing and debugging cycle of CP:

- Problem 1. When testing CP with input X, a single execution is insufficient to determine the correctness of CP with X. Even if CP with input X has been executed successfully many times, it is possible that a future execution of CP with X will fail.

- Problem 2. When debugging a failed execution of CP with input X, there is no guarantee that this execution will be repeated by executing CP with X.

- Problem 3. After CP has been modified to correct a fault detected during a failed execution of CP with input X, one or more successful executions of CP with X during regression testing does not imply that the detected fault has been corrected.

7.1 Synchronization Sequences of Concurrent Programs

An execution of CP is characterized by CP's inputs and the sequence of synchronization events that CP exercises, referred to as the *synchronization sequence* (or *SYN-sequence*) of the execution.

The definition of a SYN-sequence can be *language-based* or *implementation-based*

- A language-based definition is based on the concurrent programming constructs available in a given programming language.
- An implementation-based definition is based on the implementation of these constructs, including the interface with the run-time system, virtual machine, and operating system.

Threads in a concurrent program synchronize by performing synchronization operations (e.g., P, V, send, receive) on synchronization objects (e.g., semaphores, channels).

A synchronization event, or "SYN-event", refers to the execution of one of these operations.

The order in which SYN-events are executed is non-deterministic.

7.1.1 Complete Events vs. Simple Events

For a concurrent programming language or construct, its *complete SYN-event set* is the set of all types of SYN-events.

In general, the complete SYN-event format contains the following information:

(thread name(s), event type, object name, additional information)

This indicates that a specific thread, or a pair of synchronizing threads, executes a SYN-event of a specific type on a specific SYN-object.

Some additional information may be recorded to capture important details about the event, such as the event's vector timestamp (Section 6.3.6).

The information recorded about a SYN-event may not include the values of the messages that are received or the values of the shared variables that are read or written:

- These values are not needed for program replay, since they will be (re)computed during the normal course of execution.
- They may be needed, say, to assess the correctness of an execution.

The result of an execution of CP is determined by the text of CP and the input and complete SYN-sequence of this execution.

Example 1.

```
 mailbox C1, C2; // synchronous mailboxes

   Thread1            Thread2              Thread3              Thread4
 C1.send(msg1);   msg = C1.receive();   msg = C1.receive();   C1.send(msg1);
 C2.send(msg2);   msg = C2.receive();   msg = C2.receive();   C2.send(msg2);
```

Listing 7.1 Message passing using synchronous mailboxes

SR-events: sending and receiving messages through *mailboxes*.

The complete format of an SR-event is:

(sending thread, receiving thread, mailbox name, event type).

One possible complete SR-sequence of this program is:

(Thread1, Thread2, C1, SendReceive-synchronization),

(Thread4, Thread3, C1, SendReceive-synchronization),

(Thread1, Thread2, C2, SendReceive-synchronization),

(Thread4, Thread3, C2, SendReceive-synchronization).

End Example 1.

Some of the information recorded in a complete SYN-event is not needed for program replay. We can use "*simple SYN-sequences*" for replay.

Example 2:

For the program in Listing 7.1, the format of a simple SR-event is: (Sender, Receiver).

The simple SR-sequence corresponding to the complete SR-sequence in Example 1 is
(Thread1, Thread2),
(Thread4, Thread3),
(Thread1, Thread2),
(Thread4, Thread3).
end Example 2.

The result of an execution of CP is determined by CP and the input and simple SYN-sequence of this execution (same as for complete SYN-sequences).

Why does the channel name appear in a complete SYN-sequence but not a simple SYN-sequence?

In Example 2 the first channel that Thread1 and Thread2 access is guaranteed to be C1 and does not need to be controlled or checked during replay; thus, the channel name "C1" does not appear in the first simple SR-event.

During replay: there is a need to control which pair of threads, (Thread1 and Thread2) or (Thread4 and Thread3), access channel C1 first:

- Thus, thread names appear in the simple SR-events recorded during an execution.
- During replay, the threads are forced to access the channels in the order given in the recorded sequence.

During testing: need to determine whether the threads are synchronizing on the correct channels, in the correct order. In this case, whether Thread1 and Thread2 can and should synchronize on channel C1 is a question that we need to answer.

- Use complete SR-events, which specify the threads and the channel name for each synchronization event, and a complete SR-sequence, which specifies a synchronization order.
- During execution, try to force the threads to synchronize in the specified order, on the specified channels. If the results are not as expected, then either there is a problem with the program or we made a mistake when we generated the test sequence.

In general, it takes less information to replay an execution than to determine whether an execution is allowed by a program.

Define different types of SYN-sequences for the different activities that occur during testing and debugging.

7.1.2 Total Ordering vs. Partial Ordering

A SYN-sequence can be a *total* or *partial* ordering of SYN-events. The SYN-sequences in Examples 1 and 2 were totally-ordered.

When a concurrent program is tested or debugged using totally-ordered SYN-sequences, events must be executed one-by-one, in sequential order, which can have a significant impact on performance.

A partially-ordered SYN-sequence is actually a collection of sequences – there is one sequence for each thread or SYN-object in the program.

In a partially-ordered sequence, SYN-events that are concurrent are unordered, so that concurrent events can be executed at the same time, which speeds up execution.

Assume that a concurrent program CP consists of threads T_1, T_2, ..., T_m, m>0, and SYN-objects O_1, O_2, ..., and O_n, n>0. A partially-ordered SYN-sequence of CP may be thread-based or object-based:

Thread-based sequences: The general format of a thread-based, partially-ordered SYN-sequence of CP is $(S_1,S_2,...,S_m)$, where sequence S_i, $1 \le i \le m$, denotes a totally-ordered sequence of SYN-events of thread T_i.

- Each event in S_i has thread T_i as the executing thread and has the general format:
  (object name, object order number, event type, other necessary information).
- An object's order number is used to indicate the relative order of this event among all the events executed on the object (i.e., an event with object order number *i* is the *i*th event executed on the object).

Object-based sequences: The general format of an object-based, partially-ordered SYN-sequence of CP is $(S_1, S_2, ..., S_n)$, where sequence S_j, $1 \le j \le n$, denotes a totally-ordered sequence of SYN-events on object O_j.

- Each event in S_j has object O_j as the SYN-object and has the general format:
  (thread name(s), thread order number(s), event type, other necessary information).
- A thread's order number is used to indicate the relative order of this event among all the events executed by the thread.

Thread-based sequences are a natural way to visualize message passing programs. The space-time diagrams in Chapter 6 are partially-ordered, thread-based sequences.

Object-based sequences are helpful for understanding the behavior of an individual SYN-object. For example, object-based M-sequences were defined in Chapter 4 for monitors.

Program replay can be implemented using either thread-based or object-based sequences. Implementation details and the desired user interface may favor one approach over the other.

Example 3.

```
 mailbox C1, C2; // synchronous mailboxes

   Thread1            Thread2              Thread3              Thread4
 C1.send(msg1);   msg = C1.receive();   msg = C1.receive();   C1.send(msg1);
 C2.send(msg2);   msg = C2.receive();   msg = C2.receive();   C2.send(msg2);
```

Listing 7.1 Message passing using synchronous mailboxes

The synchronization objects are synchronous *mailboxes*.

The format of an SR-event in an object-based sequence is:

(sending thread, receiving thread, sender's order number, receiver's order number, eventType)

where:

- the *sending thread* executed the send operation
- the *receiving thread* executed the receive operation
- the *sender's order number* gives the relative order of this event among all of the sending thread's events
- the *receiver's order number* gives the relative order of this event among all of the receiving thread's events
- *eventType* is the type of this send-receive event

One feasible object-based SR-sequence of this program is:

Sequence of mailbox C1: (Thread1, Thread2, 1, 1, SendReceive-synchronization),
(Thread4, Thread3, 1, 1, SendReceive-synchronization).

Sequence of mailbox C2: (Thread1, Thread2, 2, 2, SendReceive-synchronization),
(Thread4, Thread3, 2, 2, SendReceive-synchronization).

A thread-based SR-event for thread *T* is denoted by:

(channel name, channel's order number, eventType)

where:

- the *channel name* is the name of the channel
- the *channel order number* gives the relative order of this event among all of the channel's events
- *eventType* is the type of this send-receive event

The thread-based SR-sequence corresponding to the object-based sequence above is:

Sequence of Thread1: (C1, 1, SendReceive-synchronization), (C2, 1, SendReceive-synchronization).
Sequence of Thread2: (C1, 1, SendReceive-synchronization), (C2, 1, SendReceive-synchronization).
Sequence of Thread3: (C1, 2, SendReceive-synchronization), (C1, 2, SendReceive-synchronization).
Sequence of Thread4: (C2, 2, SendReceive-synchronization), (C2, 2, SendReceive-synchronization).

end Example 3.

Totally-ordered SYN-sequences can be converted into object- and thread-based, partially-ordered SYN-sequences. Object- and thread-based sequences can be converted into each other.

Note that the totally-ordered and partially-ordered SYN-sequences of an execution should have the same "happened before" relation.

Chapter 6 described how to use integer timestamps to translate a partially-ordered sequence into a totally-ordered sequence.

7.2 Paths of Concurrent Programs

What is the relationship between the paths and SYN-sequences of a concurrent program?

7.2.1 Defining a Path

An execution of a sequential program exercises a sequence of statements, referred to as a *path* of the program.

The result of an execution of a sequential program is determined by the input and the sequence of statements executed during the execution. However, this is not true for a concurrent program.

```
    port M; // synchronous port

    Thread1             Thread2                 Thread3
(1) M.send(A);      (2) M.send(B);          (3) X = M.receive();
                                            (4) Y = M.receive();
                                            (5) output the difference (X – Y) of X and Y
```

Listing 7.2 Program CP using synchronous communication.

Assume that an execution of CP with input A=1 and B=2 exercises the totally-ordered sequence of statements (1), (2), (3), (4), (5).

This is not information to determine the output of the execution.

A totally-ordered path of a concurrent program is a totally-ordered sequence of statements plus additional information about any synchronization events that are generated by these statements.

For example, a totally-ordered path of program CP in Listing 7.2 is

((1), (2), (3, Thread1), (4, Thread2), (5)).

Events (3, Thread1) and (4, Thread2) denote that the receive statements in (3) and (4) receive messages from Thread1 and Thread2, respectively.

Information about the synchronization events of a path can also be specified separately in the form of a SYN-sequence. Thus, a totally-ordered path of CP is associated with a SYN-sequence of CP, referred to as the SYN-sequence of this path.

Assume that CP contains threads T_1, T_2, ..., and T_n. A partially-ordered path of CP is (P_1, P_2, ..., P_n), where P_i, $1 \leq i \leq n$, is a totally-ordered path of thread T_i. A partially-ordered path of CP is associated with the partially-ordered SYN-sequence of this path.

- A path (SYN-sequence) of CP is said to be *feasible for CP with input X* if this path (SYN-sequence) can be exercised by some execution of CP with input X.

- A path (SYN-sequence) of CP is said to be *feasible for CP* if this path (SYN-sequence) can be exercised by some execution of CP.

- The *domain* of a path or SYN-sequence S of CP is a set of input values. Input X is in the domain of a path or SYN-sequence S if S is feasible for CP with input X. The domain of an infeasible path or SYN-sequence is empty.

The following relationships exist between the paths and SYN-sequences of CP:

(a) If a path is feasible for CP with input X, the SYN-sequence of this path is feasible for CP with input X.

(b) If a partially-ordered SYN-sequence S is feasible for CP with input X, there exists only one partially-ordered, feasible path of CP with input X such that the partially-ordered SYN-sequence of this path is S. Thus, there exists a one-to-one mapping between partially-ordered, feasible paths of CP with input X and partially-ordered, feasible SYN-sequences of CP with input X.

(c) If a totally-ordered SYN-sequence S is feasible for CP with input X, there exists at least one totally-ordered, feasible path of CP with input X such that the totally-ordered SYN-sequence of this path is S.

(d) If two or more totally-ordered, feasible paths of CP with input X have the same totally- or partially-ordered SYN-sequence, these paths produce the same result and thus are considered to be equivalent.

(e) The domains of two or more different partially-ordered, feasible paths of CP are not necessarily mutually disjoint. This statement is also true for two or more totally-ordered, feasible paths of CP. The reason is that CP with a given input may have two or more different partially- or totally-ordered, feasible SYN-sequences.

(f) If two or more different partially-ordered, feasible paths of CP have the same partially-ordered SYN-sequence, then their input domains are mutually disjoint. However, this statement is not true for totally-ordered, feasible paths of CP.

We will illustrate relationship (e) with an example. Consider the following program:

```
     Thread1                 Thread2                Thread3
 (1) p1.send();          (1) p2.send();        (1)  input(x);
                                               (2)  if (x)
                                               (3)  output(x);
                                               (4)  select
                                               (5)     p1.receive();
                                               (6)     p2.receive();
                                               (7)  or
                                               (8)     p2.receive();
                                               (9)     p1.receive();
                                               (10) end select;
```

One partially-ordered path of this program is

Thread1: (1)
Thread2: (1)
Thread3: (1), (2), (3), (5,Thread1), (6,Thread2)

and another path is

Thread1: (1)
Thread2: (1)
Thread3: (1), (2), (3), (8,Thread2), (9,Thread1)

These paths are different, but the value of input *x* is *true* in both paths so their input domains are not disjoint.

In sequential programs, paths that are different have disjoint domains.

7.2.2 Path-based Testing and Coverage Criteria

Coverage criteria are used to determine when testing can stop and to guide the generation of input values for test cases.

Structural coverage criteria focus on the paths in a program.

The *all-paths* criterion requires every path to be executed at least once. Since the number of paths in a program may be very large or even infinite, it may be impractical to cover them all.

The minimum structural coverage criterion is *statement coverage*, which requires every statement in a program to be executed at least once.

Some stronger criteria focus on the predicates in a program.

- The predicates in if-else and loop statements divide the input domain into partitions and define the paths of a program.
- Simple predicates contain a single condition which is either a single Boolean variable (e.g., if (B)) or a relational expression (e.g., if (e1 < e2)), possibly with one or more negation operators (!).
- Compound predicates contain two or more conditions connected by the logical operators AND (∧) and OR (∨), (e.g., if ((e1 < e2) ∧ (e2 < e3))).

Predicate coverage criteria require certain types of tests for each predicate:

- *decision coverage* requires every (simple or compound) predicate to evaluate to true at least once and to false at least once. Decision coverage is also known as *branch coverage*.

- *condition coverage* requires each condition in each predicate to evaluate to true at least once and to false at least once. Note that decision coverage can be satisfied without testing both outcomes of each condition in the predicate.

  For example, decision coverage for the predicate ($A \wedge B$) is satisfied by two tests, the first being (A=true, B=true) and the second being (A=true, B=false). But neither of these tests causes A to be false. Condition coverage requires A to be false at least once.

- *decision/condition coverage* requires both decision coverage and condition coverage to be satisfied. Note that condition coverage can be satisfied without satisfying decision coverage.

  For example, for the predicate ($A \vee B$), condition coverage is satisfied by two tests: (A=true, B=false) and (A=false, B=true). But neither of these tests causes the predicate to be false. Decision/condition coverage requires the predicate to be false at least once.

- *multiple-condition coverage* requires all possible combinations of condition outcomes in each predicate to occur at least once. Note that for a predicate with N conditions, there are 2^N possible combinations of outcomes for the conditions.

These criteria can be compared based on the *subsumes* relation.

A coverage criterion C_1 is said to *subsume* another criterion C_2 if and only if any set of paths that satisfies criterion C_1 also satisfies criterion C_2.

Example: decision coverage subsumes statement coverage since covering all decisions necessarily covers all statements.

A coverage criterion that subsumes another is <u>not</u> always more effective at detecting failures. Whether or not a failure occurs may also depend on the particular input values that are chosen for a test.

Fig. 7.3 shows a hierarchy of criteria based on the subsumes relation. A path from criterion X to Y indicates that X subsumes Y.

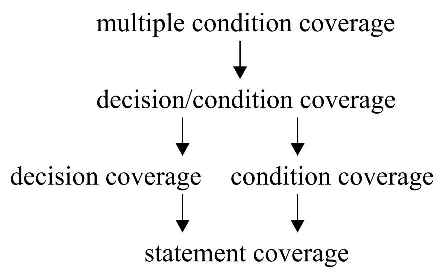


Figure 7.3 Hierarchy of sequential, structural coverage criteria based on the subsumes relation.

Instead of focusing on the control characteristics of a program, other structural coverage criteria focus on the patterns in which data is defined and used.

The *all-du-paths* criterion requires tests for "*d*efinition-*u*se (du)" pairs: if a variable is defined in one statement and used in another, there should be at least one test path that passes through both statements.

Uses may occur in predicates or in computations.

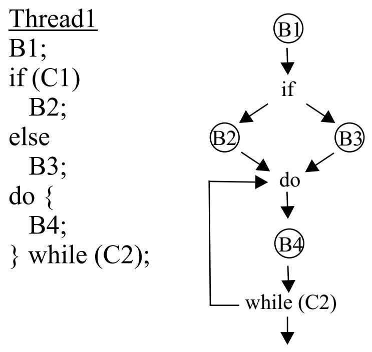
Under certain assumptions, all-du-paths subsumes decision coverage.

Structural coverage criteria are often defined with respect to a flowgraph model of a program.

In a flowgraph:

- Each circular node represents a statement or a collection of sequential statements that will be executed as a block. That is, if the first statement in the block is executed, then all the statements in the block will be executed.
- The edges between the nodes represent the flow of control from one block of code to the next.

Fig. 7.4 shows an example flowgraph for a thread that contains an if-else statement and a do-while loop.



```
Thread1
B1;
if (C1)
   B2;
else
   B3;
do {
   B4;
} while (C2);
```

Figure 7.4 A thread and its control-flow graph.

Note that some paths through a flowgraph may represent program paths that are not executable. The predicates in the if-else and loop statements must be examined to determine which paths are executable.

In a flowgraph model, statement coverage is achieved by covering *all-nodes*. Note that when a node is executed, each statement in the block represented by that node is guaranteed to be executed.

Decision coverage is achieved by covering *all-edges* of the flowgraph.

```
     port M; // synchronous port

     Thread1           Thread2              Thread3
(1) M.send(A);    (2) M.send(B);        (3) X = M.receive();
                                        (4) Y = M.receive();
                                        (5) output the difference (X – Y) of X and Y
```

Listing 7.2 Program CP using synchronous communication.

In Listing 7.2, none of the threads in CP contain any branches. Thus, any single execution of the program will cover all the statements in CP and all the paths in each of the threads. (Each thread has one path.)

However, based on the definition of a path in Section 7.2.1, there are two partially-ordered paths of CP:

- one path in which T3 receives T1's message first
- one path in which T3 receives T2's message first.

Path-based coverage criteria for concurrent programs should consider the statements exercised within threads and also the synchronization between threads, since both are used to define the paths of a concurrent program.

The paths of a concurrent program can be presented by a graph structure called a reachability graph.

The flowgraphs of individual threads can be used to build a reachability graph of a program.

The flowgraph constructed for a thread contains only the nodes and edges necessary to capture the thread's synchronization activity, e.g., sending and receiving messages, selecting alternatives in a selective wait, and ignore thread activities unrelated to synchronization.

Flowgraphs of the individual threads are analyzed to derive a "concurrency graph" of the program:

- A concurrency graph contains nodes that represent the concurrency states of the program and edges representing transitions between the states.
- A concurrency state specifies the next synchronization activity to occur in each of the program's threads.

Fig. 7.5 shows the flowgraphs for the threads in Listing 7.2 and the concurrency graph for the program. Note that the concurrency graph captures both paths in the program.

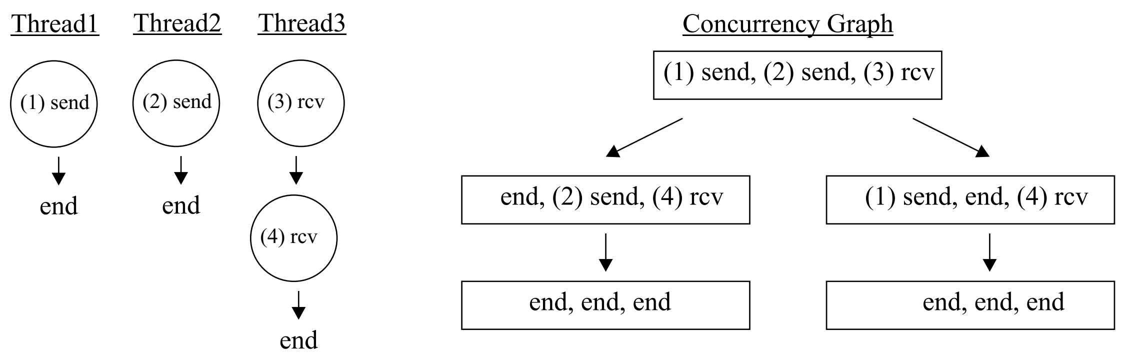


Figure 7.5 Concurrency graph for the program in listing 7.2.

Since concurrency-graphs ignore statements that are unrelated to synchronization, a path through a concurrency-graph corresponds to a SYN-sequence of a program, not a path of the program.

This is because two or more different program paths may exercise the same SYN-sequence.

In general, the concurrency-graph of program CP may not be an accurate representation of the feasible SYN-sequences of CP.

For example, some paths in the graph might not be allowed if the predicates in selective wait and if-else statements were taken into consideration.

Building accurate graph models of programs is hard to do.

Also, reachability graph models are limited by the state explosion problem, which refers to the rapid increase in the number of states as the number of threads increases.

Structural coverage criteria for synchronous message-passing programs can be defined based on the concurrency graph model:

- *all-concurrency-paths* requires all paths through the concurrency graph (i.e., all SYN-sequences) to be exercised at least once. This criterion is impossible to satisfy if cycles exist in the concurrency graph.

- *all-proper-concurrency-paths* requires all proper paths through the concurrency graph to be exercised at least once. A proper path is a path that does not contain any duplicate states, except that the last state of the path may be duplicated once. Thus, proper paths have a finite length.

- *all-edges-between-concurrency-states* requires that for each edge E in the concurrency graph there is at least one path along which E occurs.

- *all-concurrency-states* requires that for each state S in the concurrency graph there is at least one path along which S occurs.

- *all-possible-rendezvous* requires that for each state S in the concurrency graph that involves a rendezvous between threads there is at least one path along which S occurs.

The subsumes hierarchy for these criteria is shown in Fig. 7.6.

$$\begin{tikzcd}
\text{all-concurrency-paths} \arrow[d] \\
\text{all-proper-concurrency-paths} \arrow[d] \\
\text{all-edges-between-concurrency-states} \arrow[d] \\
\text{all-concurrency-states} \arrow[d] \\
\text{all-possible-rendezvous}
\end{tikzcd}$$

Figure 7.6 Subsumes hierarchy of structural coverage criteria for concurrent programs.

Once a coverage criterion is chosen, a set of SYN-sequences can be selected from the concurrency graph to satisfy the selected criterion.

The deterministic testing process, which was illustrated in Section 5.5 for message passing programs and is described in this chapter in Section 7.4, can be applied with the selected test sequences.

7.3 Definitions of Correctness and Faults for Concurrent Programs

The purpose of testing is to find failures, i.e., show that a program is incorrect.

How to define the correctness of a concurrent program?

What types of failures and faults can be found when testing concurrent programs?

Outline:

- definitions of correctness, failure, and fault for concurrent programs
- formally define three common types of failures, which are known as deadlock, livelock, and starvation.

7.3.1 Defining Correctness for Concurrent Programs

Let CP be a concurrent program.

A SYN-sequence is said to be feasible for CP with input X if this SYN-sequence can be exercised during an execution of CP with input X.

Feasible(CP,X) = the set of feasible SYN-sequences of CP with input X.

A SYN-sequence is said to be valid for CP with input X if, according to the specification of CP, this SYN-sequence is expected to be exercised during an execution of CP with input X.

Valid(CP,X) = the set of valid SYN-sequences of CP with input X.

Sets *Feasible* and *Valid* are, in general, impossible to determine.

But they are still useful for defining the correctness of concurrent programs, classifying the types of failures in concurrent programs, and comparing various validation techniques for concurrent programs.

CP is said to be correct for input X (with respect to the specification of CP) if:

(a) Feasible(CP,X) = Valid(CP,X), and

(b) every possible execution of CP with input X produces the correct (or expected) result.

The result of an execution includes the output and termination condition of the execution. The possible types of abnormal termination include divide-by-zero errors, deadlock, expiration of allocated CPU-time, etc.

CP is said to be correct (with respect to the specification of CP) if and only if CP is correct for every possible input.

For some concurrent programs, it is possible that the valid/feasible SYN-sequences are independent of the values of the program inputs. This was true for all of the programs in the previous chapters.

Several possible modifications of condition (a) are given below:

(a1) Feasible(CP,X) is a proper subset of Valid(CP,X). This condition is used when the specification of CP uses non-determinism to model design decisions that are to be made later. That is, a choice that is to be made during the design process is modeled in the specification as a non-deterministic selection between design alternatives.

- Making design decisions is a *reduction* of the non-determinism in the specification.
- In this context, specifications and implementations are relative notions in a series of system descriptions, where one description is viewed as an implementation of another description, the specification.

(a2) Valid(CP,X) is a proper subset of Feasible(CP,X). This condition is used when the specification of CP is incomplete and thus is *extended* by the implementation. In this case, the implementation adds information that is consistent with the specification.

(a3) Valid(CP,X) = Feasible(CP,X)\S, where S is a set of implementation events that are *not* mentioned in the specification of CP. Feasible(CP,X)\S is obtained from Feasible(CP,X) by deleting from each sequence in Feasible(CP,X) events in S.

- This condition is used when the events in CP's specification are a proper subset of those in CP's implementation.
- Example: In Section 4.10.3 specification-based communication events and sequences were defined for monitor-based programs. A specification may contain communication events such as “deposit” and “withdraw”, while abstracting away implementation events such as entering or executing a monitor or executing a wait or signal operation.

These alternative equivalence relations can be used by other verification techniques and may be more appropriate at certain phases of the life-cycle.

When checking the correctness of an execution it may be convenient to annotate each event in a SYN-sequence with a label that provides an abstract representation of the event, e.g., “receive_acknowledgement”. Labels are useful for

- mapping between (abstract) specifications and their implementations
- determining whether or not a feasible sequence of the implementation is also a valid sequence of the specification

7.3.2 Failures and Faults in Concurrent Programs

Based on our earlier definition of correctness, CP is incorrect for input X if and only if one or more of the following conditions hold:

(a) Feasible(CP,X) is not equal to Valid(CP,X). Thus, one or both of the following conditions hold:
    (a1) there exists at least one SYN-sequence that is feasible but invalid for CP with input X
    (a2) there exists at least one SYN-sequence that is valid but infeasible for CP with input X

(b) There exists an execution of CP with input X that exercises a valid (and feasible) SYN-sequence, but computes an incorrect result.

The existence of condition (a) is referred to as a synchronization failure (or "timing error" or "data race")

The existence of condition (b) is referred to as a computation failure.

Note that a sequential program may have computation failures but not synchronization failures.

Assume that an execution exercises a SYN-sequence S, and produces a result R. Then one of the following conditions holds: (i) S is valid and R is correct

(ii) S is valid and R is incorrect

(iii) S is invalid and R is incorrect

(iv) S is invalid and R is correct.

Condition (ii) implies that a computation failure has occurred,

Condition (iii) and condition (iv) imply that a synchronization failure has occurred.

Note that in condition (iv) a correct result R is produced from an incorrect SYN-sequence S, a condition also known as "coincidental correctness".

If the programmer checks only the correctness of R, the invalidity of S will go undetected and might cause condition (iii) to occur in the future for the same input or a different input.

⇒ when CP is executed, collect the SYN-sequences that are exercised and determine the validity of each collected SYN-sequence. (The collected SYN-sequences are also needed for regression testing and debugging.)

C onsider the faulty bounded buffer solution in Listing 7.7.

- Assume that the capacity of the buffer is two
- Assume that a single producer and a single consumer execute *deposit.call()* and *withdraw.call()* three times, respectively.

Note that the set of feasible SR-sequences and the set of valid SR-sequences of this program are independent of the program's inputs.

Thread *boundedBuffer* contains a fault. The guard for the *deposit* alternative:

```
deposit.guard (fullSlots <= capacity);
```

should be

```
deposit.guard (fullSlots < capacity);
```

This fault can cause a synchronization failure since it allows an item to be deposited when the buffer is full.

Suppose the producer deposits items 'A', 'B', and 'C' and the following invalid SR-sequence is exercised:

```
(producer,    boundedBuffer,  deposit,    rendezvous),
(producer,    boundedBuffer,  deposit,    rendezvous),
(producer,    boundedBuffer,  deposit,    rendezvous),   // deposit into a full buffer
(consumer,    boundedBuffer,  withdraw,   rendezvous),
(consumer,    boundedBuffer,  withdraw,   rendezvous),
(consumer,    boundedBuffer,  withdraw,   rendezvous).
```

The above SR-sequence starts with three consecutive rendezvous at *deposit*, followed by three consecutive rendezvous at *withdraw*:

- The output of this execution is ('C','B','C'), not the expected output ('A','B','C').
- This is an example of failure condition (iii), since this SR-sequence is invalid and the output ('C','B','C') is incorrect.

If an execution of *boundedBuffer* with input ('C','B','C') exercises the above invalid SR-sequence, then the output of this execution is ('C','B','C')"

- This is an example of failure condition (iv) above, since this SR-sequence is invalid but the output ('C','B','C') is correct.
- An execution of *boundedBuffer* that does not exercise the above SR-sequence will not produce an invalid SR-sequence nor will it produce an incorrect result.

Finally, assume that the incorrect guard for *deposit* is modified to:

```
deposit.guard (fullSlots+1 < capacity);
```

Now thread *boundedBuffer* allows at most one character in the buffer.

In this case, the set of feasible SR-sequences of *boundedBuffer* is a proper subset of the set of valid SR-sequences of *boundedBuffer*, i.e., *boundedBuffer* has a missing path:

- *boundedBuffer* still has a possible synchronization failure.
- This failure cannot be detected by a non-deterministic execution of *boundedBuffer* since such an execution will always exercise an SR-sequence that is feasible and valid, and will always produce a correct result.

```
final class boundedBuffer extends TDThread {
   private selectableEntry deposit, withdraw;
   private int fullSlots=0; private int capacity = 0;
   private Object[] buffer = null; private int in = 0, out = 0;
   public boundedBuffer(selectableEntry deposit, selectableEntry withdraw,
         int capacity) {
      this.deposit = deposit; this.withdraw = withdraw; this.capacity = capacity;
      buffer = new Object[capacity];
   }
   public void run() {
      try {
         selectiveWait select = new selectiveWait();
         select.add(deposit);            // alternative 1
         select.add(withdraw);           // alternative 2
         while(true) {
            deposit.guard (fullSlots <= capacity); // *** (fullSlots < capacity)
            withdraw.guard(fullSlots > 0);
            switch (select.choose()) {
            case 1:  Object o = deposit.acceptAndReply();
                     buffer[in] = o; in = (in + 1) % capacity; ++fullSlots;
                     break;
            case 2:  withdraw.accept();
                     Object value = buffer[out]; withdraw.reply(value);
                     out = (out + 1) % capacity; --fullSlots;
                     break;
            }
         }
      } catch (InterruptedException e) {}
        catch (SelectException e) {
           System.out.println("deadlock detected"); System.exit(1);
        }
   }
}
```

Listing 7.7 A faulty bounded buffer.

7.3.3 Deadlock, Livelock, and Starvation

The absence of deadlock and livelock is usually an implicit requirement of all programs.

Deadlock, livelock, and starvation can be formally defined in terms of the reachability graph (Section 7.2.2.) of a concurrent program.

The reachability graph of program CP, denoted by RG_{CP}, contains all the reachable states of CP.

- a state of RG_{CP} contains the number of the next statement to be executed by each of the threads in CP, and the values of the variables in CP.
- a path of RG_{CP} corresponds to a sequence of statements and SYN-events in CP.

We assume that the reachability graph of CP contains a finite number of states.

Let CP be a concurrent program containing threads $T_1, T_2, \ldots, T_r$, where $r > 1$.

A state of CP is denoted as ($S_1, S_2, \ldots, S_r$, other information):

- S_i, $1 \leq i \leq r$, is the atomic action or the set of atomic actions that can possibly be executed next by thread T_i
- “other information” in a state may include the values of local and global variables, the contents of message queues, etc.

Let S = (S_1, S_2, …, S_r, other information) be a state of CP:

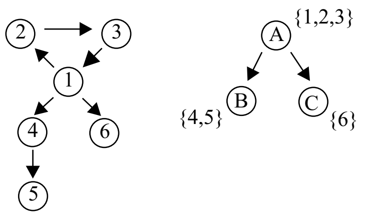
- If action S_i, $1 \le i \le r$, is a blocking synchronization statement, and executing S_i blocks thread T_i in state S, then thread T_i is said to be blocked in state S.
- If thread T_i, $1 \le i \le r$, is neither terminated nor blocked in state S, then S has one or more outgoing transitions for thread T_i resulting from the execution of S_i.
- If S_i is an if, while, or assignment statement, then S has exactly one outgoing transition for T_i.
- If S_i is a non-deterministic statement, such as a selective wait statement, then S may have multiple outgoing transitions for T_i.
- If S_i is an input statement for one or more variables, then S may have one outgoing transition for each possible combination of values of the input variables.

A strong component G' of a directed graph G is a maximal sub-graph of G in which there is a path from each node of G' to any other node of G' [Baase 1988].

Let the condensation of graph G, denoted as Condensed(G), be G modified by considering each strong component as a node.

- Condensed(G) is cycle-free and has a tree structure.
- A leaf node in Condensed(G) is a node without child nodes.

Fig. 7.8 shows a directed graph and the condensation of the graph. Nodes B and C are leaf nodes.



7.3.3.1 Deadlock.

Assume that at the end of some execution of program CP there exists a thread T that satisfies these conditions:

- T is blocked due to the execution of a synchronization statement (e.g., waiting for a message to be received)
- T will remain blocked forever, regardless of what the other threads do

Thread T is said to be deadlocked and CP is said to have a deadlock.

Example:

- Thread1 is blocked waiting to receive a message from Thread2
- Thread2 is blocked waiting to receive a message from Thread1

Both Thread1 and Thread2 will remain blocked forever since neither thread is able to send the message for which the other thread is waiting.

Let CP be a concurrent program and S be a state in RG_{CP}:

- If a thread T in CP is blocked in S and all states reachable from S, then T is deadlocked in S and S is a deadlock state of T.
- S is a deadlock state if at least one thread of CP is deadlocked in S. A deadlock state S is a global deadlock state if every thread is either blocked or terminated in S; otherwise, S is a local deadlock state.
- CP has a deadlock if RG_{CP} contains at least one deadlock state.

7.3.3.2 An algorithm for detecting deadlock. Let CP be a concurrent program containing threads T_1, T_2, …, T_r.

For each node N in the condensed reachability graph, algorithm *DeadlockTest* computes two sets of threads:

- Blocked(N) is the set of threads that are blocked in every state in node N. (Remember that all the states in N are in the same strong component of the reachability graph.)
- Deadlock(N) contains i, $1 \leq i \leq r$, if and only if thread T_i is deadlocked in every state in node N.

Program CP contains a deadlock if Deadlock(N) is not empty for some node N.
Algorithm *DeadlockTest* is as follows:

- Construct the condensation of RG_{CP}, denoted as Condensed(RG_{CP}). A node in Condensed(RG_{CP}) is a set of states in RG_{CP}.
- Perform a depth-first traversal of the nodes in Condensed(RG_{CP}). For each node N in Condensed(RG_{CP}), after having visited the child nodes of N:
  - Blocked(N) = {i | thread T_i is blocked in every state in N}
  - if N is a leaf node of Condensed(RG_{CP}) then Deadlock(N) = Blocked(N), else Deadlock(N) = the intersection of Blocked(N) and the Deadlock sets of N's child nodes.

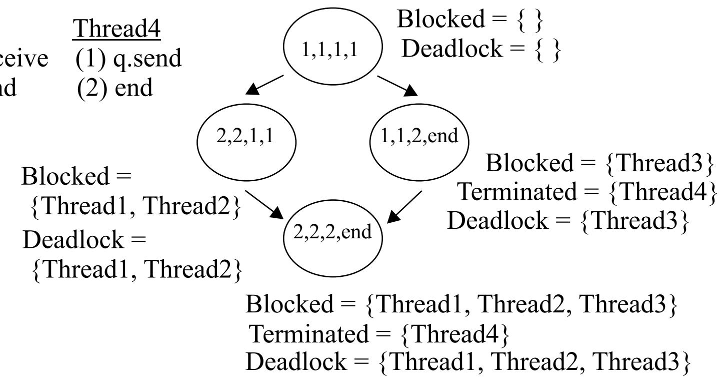
Let n be the number of transitions in RG_{CP}. Since RG_{CP} has only one initial state, the number of states in RG_{CP} is less than or equal to n+1.

- step (a) is at most O(n)
- step (b) at most O(n*r).

So the time complexity of algorithm *DeadlockTest* is at most O(n*r).

Fig. 7.9 shows four threads and the reachability graph for these threads. (The reachability graph and the condensed graph are the same.) The state labels show the next statement to be executed by each thread.

Thread1	Thread2	Thread3	Thread4
(1) p.send	(1) p.receive	(1) q.receive	(1) q.send
(2) p.send	(2) r.send	(2) s.send	(2) end
(3) r.receive	(3) p.receive	(3) end	
(4) end	(4) end		



Algorithm *DeadlockTest* proceeds as follows:

- Node (2,2,2,end) is a leaf node. Since Thread1, Thread2, and Thread3 are blocked, Deadlock(2,2,2,end) = {Thread1, Thread2, Thread3}.
- In node (2,2,1,1), Thread1 and Thread2 are blocked. The only child node of (2,2,1,1) is node (2,2,2,end), where Deadlock(2,2,2,end) was just computed to be {Thread1, Thread2, Thread3}. Thus, Deadlock(2,2,1,1) = {Thread1,Thread2} $\cap$ {Thread1, Thread2, Thread3} = {Thread1, Thread2}.
- In node (1,1,2,end), thread Thread3 is blocked. The only child node of (1,1,2,end) is node (2,2,2,end). Thus, Deadlock(1,1,2,end) = {Thread3} $\cap$ { Thread1, Thread2, Thread3} = {Thread3}.
- In node (1,1,1,1), there are no blocked threads. Thus, Deadlock(1,1,1,1) = { }.

Hence*, DeadlockTest* has detected a deadlock in the program. States (2,2,1,1) and (1,1,2,2) are both local deadlock states, while state (2,2,2,end) is a global deadlock state.

3.3.3.3 Livelock.

Assume that some statements in CP are labeled as "progress statements”, indicating that the threads are expected to eventually execute these statements. Examples:

- the last statement of a thread,
- the first statement of a critical section,
- the statement immediately following a loop or a synchronization statement.

If a thread executes a progress statement, it is considered to be “making progress”.

Assume there is an execution of CP that exercises an execution sequence S, and at the end of S there exists a thread T that satisfies the following conditions, regardless of what the other threads will do:

- T will not terminate or deadlock
- T will never make progress

Thread T is said to be livelocked at the end of S, and CP is said to have a livelock.

Livelock is the busy-waiting analog of deadlock. A livelocked thread is running (or ready to run), not blocked, but it can never make any progress.

Example: Incorrect solution 2 in Section 2.1.2 (and reproduced below) has an execution sequence that results in a violation of the progress requirement for solutions to the critical section problem (not to be confused with the more general requirement to “make progress” that we introduced in this section.)

The first statement of the critical section is designated as a progress statement.

```
int turn = 0;
Thread0                       Thread1
while (true) {               while (true) {
  while (turn != 0) { ; }      (1)              while (turn != 1) { ; }   (1)
  critical section    (2)      critical section      (2)
  turn = 1;           (3)      turn = 0;             (3)
  non-critical section         (4)              non-critical section   (4)
}                             }
```

Below is a prefix of the execution sequence that violates the progress requirement of the critical section problem.

- Thread0 executes (1), (2), and (3). Now *turn* is 1.
- Thread1 executes (1), (2), and (3) and then terminates in its non-critical section. Now *turn* is 0.
- Thread0 executes (4), (1), (2), (3), (4), and (1). Now *turn* is 1.
- Thread0 is stuck in its busy-waiting loop at (1) waiting for *turn* to become 0. Thread0 will never exit this busy-waiting loop and enter its critical section, i.e., make any progress. Thus, Thread0 is livelocked.

Let CP be a concurrent program and S a state of RG_{CP}:

- A thread in CP is said to make progress in S if S contains a progress statement for this thread.
- If a thread T in CP is not deadlocked, terminated, or making progress in S or any state reachable from S, then T is livelocked in S and S is a livelock state for T.
- S is a livelock state of RG_{CP} if at least one thread is livelocked in S. A livelock state S is a global livelock state if every thread in S is either livelocked or terminated; otherwise, S is a local livelock state.
- CP has a livelock if RG_{CP} contains at least one livelock state.

7.3.3.4 An algorithm for detecting livelock.

Let CP be a concurrent program containing threads T_1, T_2, …, T_r. For each node N in the condensed reachability graph, algorithm *LivelockTest* computes two sets of threads:

- NoProgress(N) is the set of threads that are not deadlocked, terminated, or executing a progress statement in any state in N.
- Livelock(N) contains i, $1 \le i \le r$, if and only if thread T_i is livelocked in every state in N.

Program CP contains a livelock if Livelock(N) is not empty for some node N. Algorithm *LivelockTest* is as follows:

(a) Construct Condensed(RG_{CP}).

(b) Perform a depth-first traversal of the nodes in Condensed(RG_{CP}). For each node N in Condensed(RG_{CP}), after having visited the child nodes of N:

- NoProgress(N) = {i | thread T_i is not deadlocked, terminated, or executing a progress statement in any state in N},
- if N is a leaf node of Condensed(RG_{CP}) then Livelock(N) = NoProgress(N), else Livelock (N) = the intersection of NoProgress(N) and the NoProgress sets of N's child nodes.

Algorithm *LivelockTest* has the same time complexity as algorithm *DeadlockTest*.

7.3.3.5 Starvation.

Assume that the scheduling policy used in executing a concurrent program CP is fair, i.e., a thread ready for execution will eventually be scheduled to run.

A cycle in RG_{CP} is said to be a fair cycle if for every thread T in CP, either this cycle contains at least one transition for T, or T is blocked or terminated in every state on this cycle.

Informally, CP is said to have a starvation if CP can reach a state on a fair cycle such that some thread in CP is not deadlocked, livelocked, or terminated in this state, but this thread may not make any progress in any state on this cycle.

Incorrect solution 3 in Section 2.1.3 (and reproduced below) has a starvation.

```
boolean intendToEnter0=false, intendToEnter1 = false;

Thread0                                  Thread1
while (true) {                          while (true) {
  intendToEnter0 = true;          (1)     intendToEnter1 = true;          (1)
  while (intendToEnter1) {        (2)     while (intendToEnter0) {        (2)
    intendToEnter0 = false;       (3)       intendToEnter1 = false;       (3)
    while(intendToEnter1) {;}     (4)       while(intendToEnter0) {;}     (4)
    intendToEnter0 = true;        (5)       intendToEnter1 = true;        (5)
  }                                       }
  critical section                (6)     critical section                (6)
  intendToEnter0 = false;         (7)     intendToEnter1 = false;         (7)
  non-critical section            (8)     non-critical section            (8)
}                                       }
```

State (4,2) is not a livelock state for Thread0 since the following sequence allows Thread0 to enter its critical section (4,2) are the next statements to be executed by Thread0 and Thread1, respectively:

(a) Thread1 executes (2) and (6), enters and exits it critical section, and executes (7)

(b) Thread0 executes (4), (5), and (2), and then enters its critical section at (6)

State (4,2) has a cycle to itself that contains one transition for Thread0 (representing an iteration of the busy-waiting loop in (4)) and no other transitions. This cycle is not a fair cycle since it does not contain a transition for Thread1.

State (4,2) has another cycle to itself that represents the following execution sequence:

(c) Thread1 executes (2) and (6), enters and exits it critical section, and then executes (7), (8), and (1)

(d) Thread0 executes (4) and stays in its busy-waiting loop

This cycle is fair. After state (4,2) is entered, if this cycle is repeated forever, Thread0 never enters its critical section. State (4,2) is called a starvation state for Thread0.

Let CP be a concurrent program and S a state of RG_{CP}:

- A cycle in RG_{CP} is said to be a no-progress cycle for a thread T in CP if T does not make progress in any state on this cycle. (Assume some statements are labeled as "progress statements").
- A cycle in RG_{CP} is said to be a starvation cycle for a thread T in CP if (1) this cycle is fair, (2) this cycle is a no-progress cycle for T, and (3) each state on this cycle is not a deadlock, livelock, or termination state for T.
- A starvation cycle for thread T is said to be a busy-starvation cycle for T if this cycle contains at least one transition for T, and is said to be a blocking-starvation cycle for T otherwise (i.e., T is blocked in each state on this cycle).
- If state S is on a starvation cycle for thread T then S is a starvation state for T. A starvation state is a global starvation state if every thread in S is either starved or terminated; otherwise, it is a local starvation state.
- CP is said to have a starvation if RG_{CP} contains at least one starvation state.

7.3.3.6 An algorithm for detecting starvation. Let CP be a concurrent program containing threads $T_1, T_2, \ldots, T_r$.

For each node N in the condensed reachability graph, algorithm *StarvationTest* computes two sets of threads:

- NoProgress(N) is the set of threads that do not terminate in N, and for which N contains a fair, no-progress cycle.
- Starvation(N) contains i, $1 \leq i \leq r$ if and only if a starvation cycle for thread T_i exists in N.

Program CP contains a starvation if Starvation(N) is not empty for some node N. Algorithm *StarvationTest* is as follows:

(a) Construct Condensed(RG_{CP}).

(b) Perform a depth-first traversal of the nodes in Condensed(RG_{CP}). For each node N in Condensed(RG_{CP}), after having visited the child nodes of N:
  - if N does not contain any fair cycles, then Starvation(N) = empty,
  - else NoProgress(N) = {i | thread T_i does not terminate in N, and N contains a fair, no-progress cycle for T_i} and Starvation(N) = NoProgress(N) – Deadlock(N) – Livelock(N).

To compute NoProgress(N), we need to search for fair cycles in N.

- We must consider cycles of length at most (1 + #Transitions) where #Transitions is the number of transitions in N.
- The number of cycles with length less than or equal to (1 + #Transitions) is at most $O(2^{\#Transitions})$.

Let n be the number of transitions in RG_{CP}. The time complexity of algorithm *StarvationTest* is at most $O(r*2^n)$.

7.3.3.7 Other Definitions

Local deadlock has been referred to as a deadness error and permanent blocking. Global deadlock has been referred to as infinite wait, global blocking, deadlock, and system-wide deadlock.

Circular deadlock: a circular list of two or more threads such that each thread is waiting to synchronize with the next thread in the list:

- Similar to a circular wait condition that arises during resource allocation (see Section 3.10.4).
- A circular wait condition is a necessary condition for deadlock during resource allocation.
- According to our definition, a deadlock in a concurrent program is different from a deadlock during resource allocation, since the former is not necessarily a circular deadlock.

Alternate definitions of livelock:

- A thread that is spinning (i.e., executing a loop) while waiting for a condition that will never become true.
- the existence of an execution sequence that can be repeated infinitely often without ever making effective progress.

Alternate definitions of starvation:

- a process, even though not deadlocked, waits for an event that may never occur
- a situation in which processes wait indefinitely,
- a situation in which processes continue to run indefinitely, but fail to make any progress.

Definitions of deadlock, livelock, and starvation based on reachability graphs:

- are independent from the programming language and constructs used to write the program
- are formally defined in terms of the reachability graph of a program
- cover all undesirable situations involving blocking or not making progress
- define deadlock, livelock, and starvation as distinct properties of concurrent programs
- provide a basis for developing detection algorithms.

The mutual exclusion, progress, and bounded waiting requirements for solutions to the critical section problem can be defined in terms of deadlock, livelock, and starvation and the correctness of a solution to the critical section problem can be verified automatically.

7.4 Approaches to Testing Concurrent Programs

Two types of testing:

- *black-box testing*: Access to CP's implementation is not allowed during black-box testing. Thus, only the specification of CP can be used for test generation, and only the result (including the output and termination condition) of each execution of CP can be collected.
- *white-box testing*: Access to CP's implementation is allowed during white-box testing. In this case, both the specification and implementation of CP can be used for test generation. Also, any desired information about each execution of CP can be collected.

White-box testing may not be practical during system or acceptance testing, due to the size and complexity of the code or the inability to access the code.

Limited white-box testing is a thrd type of testing that lies somewhere between the first two approaches: During an execution of CP, only the result and SYN-sequence can be collected:

- only the specification and the SYN-sequences of CP can be used for test generation
- an input and a SYN-sequence can be used to deterministically control (see below) the execution of CP.

7.4.1 Non-Deterministic Testing

Non-deterministic testing of a concurrent program CP involves the following steps:

1. Select a set of inputs for CP
2. For each selected input X, execute CP with X many times and examine the result of each execution

Multiple, non-deterministic executions of CP with input X may exercise different SYN-sequences of CP and thus may detect more failures than a single execution.

This approach can be used during both (limited) white-box and black-box testing.

Non-deterministic testing tries to exercise as many distinct SYN-sequences as possible:

- repeated executions do not always execute different SYN-sequences.
- the "probe effect", which occurs when programs are instrumented with testing and debugging code, may make it impossible for some failures to be observed

Techniques for exercising different SYN-sequences during non-deterministic testing:

- change the scheduling algorithm used by the operating system, e.g., change the value of the time quantum
- insert Sleep statements into the program with the sleep time randomly chosen to ensure a non-zero probability for exercising an arbitrary SYN-sequence,

Still:

- some sequences are likely to be exercised many times, which is inefficient, and some may never be exercised at all.
- the result of the execution must be checked, which is difficult and tedious if done manually.

7.4.2 Deterministic Testing

Deterministic testing of a concurrent program CP involves the following steps:

1. Select a set of tests, each of the form (X, S), where X and S are an input and a complete SYN-sequence of CP, respectively.

2. For each selected test (X, S), force a deterministic execution of CP with input X according to S. This forced execution determines whether S is feasible for CP with input X. (Since S is a complete SYN-sequence of CP, the result of such an execution is deterministic.)

3. Compare the expected and actual results of the forced execution (including the output, the feasibility of S, and the termination condition). If the expected and actual results are different, a failure is detected in the program (or an error was made when the test sequence was generated). A replay tool can be used to locate the fault that caused the failure. After the fault is located and CP is corrected, CP can be executed with each test (X,S) to verify that the fault has been removed and that in doing so, no new faults were introduced.

Note that for deterministic testing, a test for CP is not just an input of CP. A test consists of an input and a SYN-sequence, and is referred to as an IN-SYN test.

Deterministic testing provides several advantages over non-deterministic testing:

- Non-deterministic testing may leave certain paths of CP uncovered. Several path-based test coverage criteria were described in Section 7.2.2. Deterministic testing allows carefully selected SYN-sequences to be used to test specific paths of CP.

- Non-deterministic testing exercises feasible SYN-sequences only; thus, it can detect the existence of invalid, feasible SYN-sequences of CP, but not the existence of valid, infeasible SYN-sequences of CP. Deterministic testing can detect both types of failures.

- After CP has been modified to correct an error or add some functionality, deterministic regression testing with the inputs and SYN-sequences of previous executions of CP provides more confidence about the correctness of CP than non-deterministic testing of CP with the inputs of previous executions.

The selection of IN-SYN tests for CP can be done in different ways:

- Select inputs and then select a set of SYN-sequences for each input
- Select SYN-sequences and then select a set of inputs for each SYN-sequence
- Select inputs and SYN-sequences separately and then combine them
- Select pairs of inputs and SYN-sequences together

Chapters 1 through 6 dealt with various issues that arise during deterministic testing and debugging. These issues are summarized below:

Program Replay: Repeating an execution of a concurrent program is called "program replay".

The SYN-sequence of an execution must be traced so that the execution can be replayed.

- Program replay uses simple SYN-sequences, which have a simpler format than the complete sequences used for testing.
- Definitions of simple SYN-sequences for semaphores, monitors, and message passing were given in Chapters 3 – 6.

The synchronization library developed in the text supports replay, but it does not have the benefit of being closely integrated with a source-level debugger.

Program Tracing:

- Chapters 2 – 6 showed how to trace simple and complete SYN-sequences for shared variables, semaphores, monitors and various types of message channels.

- Observability problem: When tracing a distributed program it is difficult to accurately determine the order in which actions occur during an execution. Vector timestamps (Chapter 6) can be used to ensure that an execution trace of a distributed program is consistent with the actual execution.

- For long-running programs, storing all the SYN-events requires too much space. "Adaptive tracing" techniques minimize the number of SYN-events required to exactly replay an execution.

Sequence Feasibility: A sequence of events that is allowed by a program is said to be a *feasible* sequence.

- Program replay always involves repeating a feasible sequence of events.

- Testing, on the other hand, involves determining whether or not a given sequence is feasible or infeasible. Valid sequences are expected to be feasible while invalid sequences are expected to be infeasible.

- The information and the technique used to determine the feasibility of a SYN-sequence are different from those used to replay a SYN-sequence. The techniques illustrated in Chapters 4 – 6 check the feasibility of complete SYN-sequences of monitors and message channels.

Approaches for selecting valid and invalid SYN-sequences for program testing:

- Collect the feasible SYN-sequences that are randomly exercised during non-deterministic testing. These SYN-sequences can be used for regression testing when changes are made to the program.

- Generate sequences that satisfy a coverage criteria (Section 7.2.2) or that are adequate for the mutation-based testing (Section 7.4.3.2). Mutation testing has the advantage that it requires both valid and invalid sequences to be generated.

Sequence Validity: A sequence of actions captured in a trace is definitely feasible, but the sequence may or may not be valid.

The goal of testing is to find valid sequences that are infeasible and invalid sequences that are feasible; such sequences are evidence of a program failure.

A major issue then is how to check the validity of a sequence.

- If a formal specification of valid program behavior is available, then checking the validity of a SYN-sequence can be partially automated.
- Without such a “test oracle”, manually checking validity becomes time-consuming, error prone, and tedious.

The Probe Effect: Modifying a concurrent program to capture a trace of its execution may interfere with the normal execution of the program:

- Working programs may fail when instrumentation is removed
- failures may disappear when debugging code is added.

On the other hand, executions can be purposely disturbed during non-deterministic testing in order to capture as many different SYN-sequences as possible - instrumentation at least offers the prospect of being able to capture and replay the failures that are observed.

One approach to circumventing the probe effect is to systematically generate all the possible SYN-sequences. This approach can be realized through reachability testing if the number of sequences is not too large (Sections 3.10.5, 4.11.4, 5.5.5, and 7.5).

Three different problems:

- The observability problem is concerned with the difficulty of accurately tracing an execution of a distributed program. In Section 6.3.6, we saw how to use vector timestamps to address the observability problem.

- The probe effect is concerned with the ability to perform a given execution at all:
  - Deterministic testing partially addresses the probe effect by allowing us to choose a particular SYN-sequence that we want to exercise.
  - Reachability testing goes one step further and attempts to exercise all possible SYN-sequences.

- The observability problem and the probe effect are different from the replay problem, which deals with repeating an execution that has already been observed.

Real-Time: The probe effect is a major issue for real-time concurrent programs.

The correctness of a real-time program depends not only on its logical behavior, but also on the time at which its results are produced. [Tsai et al. 1996].

A real-time program may have execution deadlines that will be missed if the program is modified for tracing.

- tracing is performed by using special hardware to remove the probe effect, or by trying to account for or minimize the probe effect.
- Real-time programs may also receive sensor inputs that must be captured for replay.

The text does not considered the special issues associated with timing correctness.

Tools: The synchronization library presented in Chapters 1 – 6 is a simple but useful programming tool; however, it is no substitute for an integrated development environment that supports traditional source level debugging as well as the special needs of concurrent programmers.

Life-Cycle Issues: Deterministic testing is better suited for the types of testing that occur early in the software life-cycle.
Feasibility checking and program replay require information about the internal behavior of a system. Thus, deterministic testing is a form of white-box or limited white-box testing.

Deterministic testing can be applied during early stages of development allowing concurrency bugs to be found as early as possible, when powerful debugging tools are available and bugs are less costly to fix.

7.4.3 Combinations of Deterministic and Non-Deterministic Testing

Deterministic testing has advantages over non-deterministic testing but it requires considerable effort for selecting SYN-sequences and determining their feasibility.

This effort can be reduced by combining deterministic and non-deterministic testing. Below are four possible strategies for combining these approaches:

(a) Apply non-deterministic testing first with random delays to collect random SYN-sequences and detect failures. Then apply deterministic regression testing with the collected sequences. No extra effort is required for generating SYN-sequences since they are all randomly selected during non-deterministic executions.

(b) Apply non-deterministic testing until test coverage reaches a certain level. Then apply deterministic testing to achieve a higher level of coverage. This strategy is similar to the combination of random and special value testing for sequential programs.

Six Pascal programs were randomly tested against the same specification. Random testing rapidly reached steady-state values for several test coverage criteria: 60% for decision (or branch) coverage, 65% for block (or statement) coverage, and 75% for definition-use coverage, showing that special values (including boundary values) are needed to improve coverage.

(c) SYN-sequences collected during non-deterministic testing can be modified to produce new SYN-sequences for deterministic testing (easier than starting from scratch.

(d) Apply deterministic testing during module and integration testing and non-deterministic testing during system and acceptance testing.

7.4.3.1 Prefix-Based Testing.

The purpose of prefix-based testing is to allow non-deterministic testing to start from a specific program state other than the initial one.

Prefix-based testing uses a “prefix sequence”, which contains events from the beginning part of an execution, not a complete execution.

Prefix-based testing of CP with input X and prefix sequence S proceeds as follows:

(1) Force a <u>deterministic execution</u> of CP with input X according to S. If this forced execution succeeds, (i.e., it reaches the end of S), then go to step (2); otherwise S is infeasible.

(2) Continue the execution of CP with input X by performing <u>non-deterministic testing</u> of CP.

If S is feasible for CP with input X, then prefix-based testing replays S in step (1).

The purpose of step (1) is to force CP to enter a particular state, e.g., a state in which the system is under a heavy load, so that we can see what happens after that in step (2).

Prefix-based testing is an important part of reachability testing (Section 7.5).

7.4.3.2 Mutation-Based Testing.

Mutation-based testing helps the tester create test cases and then interacts with the tester to improve the quality of the tests.

Mutation-based testing subsumes the coverage criteria in Fig. 7.3. That is, if mutation coverage is satisfied, then the criteria in Fig. 7.3 are also satisfied.

$$\begin{tikzcd}
& \text{multiple condition coverage} \arrow[d] & \\
& \text{decision/condition coverage} \arrow[dl] \arrow[dr] & \\
\text{decision coverage} \arrow[dr] & & \text{condition coverage} \arrow[dl] \\
& \text{statement coverage} &
\end{tikzcd}$$

Figure 7.3 Hierarchy of sequential, structural coverage criteria based on the subsumes relation.

Mutation-based testing also provides some guidance for the generation of invalid SYN-sequences, unlike the criteria in Fig. 7.3.

Mutation-based testing constructs of a set of *mutants* of the program under test:

- Each mutant differs from the program under test by one *mutation*.
- A mutation is a single syntactic change made to a program statement, generally inducing a typical programming fault, e.g., changing <= to <.

If a test case causes a mutant program to produce output different from the output of the program under test:

- that test case is strong enough to detect the faults represented by that mutant,
- the mutant is considered to be *distinguished* from the program under test.

Each set of test cases is used to compute a *mutation score*.

- A score of 100% indicates that the test cases distinguish all mutants of the program under test and are adequate with respect to the mutation criterion.
- Some mutants are functionally equivalent to the program under test and can never be distinguished. This is factored into the mutation score.

Fig. 7.10 shows a mutation-based testing procedure for a *sequential* program P.

Non-deterministic execution behavior creates the following problem:

> In line (10), the condition $Actual_p$ <> $Actual_{mi}$ is not sufficient to mark mutant m_i as distinguished. Different actual results may be a product of non-determinism and not the mutation.

This problem can be solved by using a combination of deterministic testing and non-deterministic mutation-based testing.

```
(1)  Generate mutants ($m_1$,$m_2$,...,$m_n$) from P;
(2)  repeat {
(3)     Execute P with test input X producing actual result $Actual_p$;
(4)     Compare the actual result $Actual_p$ with the expected result $Expected_p$;
(5)     if ($Expected_p$ != $Actual_p$)
(6)        Locate and correct the error in P and restart at (1);
(7)     else
(8)        for (mutant $m_i$, i<=i<=n) {
(9)           Execute $m_i$ with test input X producing actual result $Actual_{mi}$;
(10)          if ($Actual_p$ <> $Actual_{mi}$)
(11)             mark mutant $m_i$ as distinguished;
(12)       }
(13) }
(14) until (the mutation score is adequate);
```

Figure 7.10 A mutation-based testing procedure for a sequential program P.

A two-phase procedure for deterministic mutation testing (DMT).

- phase one: SYN-sequences are randomly generated using non-deterministic testing, until the mutation score has reached a steady value.
- phase two: select IN_SYN test cases and apply deterministic testing until an adequate mutation score is achieved.

Fig. 7.11 shows a phase one procedure using non-deterministic testing to randomly select SYN-sequences for mutation-based testing:

- line (4): if SYN-sequence S_{CP} and actual result $Actual_{CP}$ were produced by an earlier execution of CP with input X, then we should execute CP again until a new SYN-sequence or actual result is produced.
- line (16), deterministic testing is used to distinguish mutant programs by differentiating the output *and the feasible SYN-sequences* of the mutants from those of the program under test.
    - If the SYN-sequence randomly exercised by CP during non-deterministic testing is infeasible for the mutant program, or this sequence is feasible but the mutant program produces results that are different from CP's, then the mutant is marked as distinguished.

```
(1)   repeat {
(2)      Generate mutants ($m_1$,$m_2$,...$m_n$) from CP;
(3)      Apply non-determ. testing to randomly execute CP with test input X;
(4)      Assume execution exercises new SYN-sequence $S_{CP}$, or produces a new
            actual result $Actual_{CP}$.
(5)      Check which of the following conditions holds:
(6)      (a) $S_{CP}$ is valid and $Actual_{CP}$ is correct
(7)      (b) $S_{CP}$ is valid and $Actual_{CP}$ is incorrect
(8)      (c) $S_{CP}$ is invalid and $Actual_{CP}$ is correct
(9)      (d) $S_{CP}$ is invalid and $Actual_{CP}$ is incorrect;
(10)     if (condition (b), (c), or (d) holds) {
(11)        Locate and correct the error in CP using program replay; apply
(12)        Apply det. testing to validate the correction by forcing an execution
               of CP with IN_SYN test case (X,$S_{CP}$); and restart at (1);
(13)     } else
(14)        for (mutant $m_i$, i<=i<=n) {
(15)           Apply deterministic testing to $m_i$ with IN_SYN test case (X,$S_{CP}$)
                  producing actual result $Actual_{mi}$;
(16)           if (($S_{CP}$ is infeasible for $m_i$) or
                     ($S_{CP}$ is feasible and $Actual_{CP}$ <> $Actual_{mi}$))
(17)              mark mutant $m_i$ as distinguished;
(18)        }
(19) }
(20) until (the mutation score reaches a steady value);
```

Figure 7-11 Deterministic Mutation Testing (DMT) using non-deterministic testing to generate SYN-sequences.

It may not be possible to distinguish some of the mutants if non-deterministic testing alone is applied to CP in line (3):

- To distinguish a mutant m_i, we may need to exercise SYN-sequences that are feasible for mutant m_i but *infeasible* for CP;
- however, in line (3) only *feasible* SYN-sequences of CP can be exercised using non-deterministic testing.

Example 1. Assume that the program under test is an incorrect version of the bounded buffer that allows at most *one* (instead of two) consecutive *deposits* into the buffer. (In other words, the program under test has a fault.) Call this program boundedBuffer1.

A possible mutant of this program is the correct version in Listing 5-10. Call this correct version boundedBuffer2.

Mutant *boundedBuffer2* is distinguished by an SR-sequence that exercises *two* consecutive *deposits*, as this sequence differentiates the behaviors of these two versions. But this SR-sequence is a *valid, infeasible* SR-sequence of *boundedBuffer1* that cannot be exercised when non-deterministic testing is applied to *boundedBuffer1* in line (3).

Example 2. Assume that the program under test is *boundedBuffer2*, which correctly allows at most *two* consecutive *deposit* operations.

A possible mutant of this program is *boundedBuffer3* (the mutation shown in Listing 7.7).

Mutant *boundedBuffer3* is distinguished by an SR-sequence that exercises *three* consecutive *deposits*. But this SR-sequence is an *invalid, infeasible* SYN-sequence of *boundedBuffer2* that cannot be exercised when non-deterministic testing is applied to *boundedBuffer2* in line (3).

⇒ Upon reaching a steady mutation score, select IN_SYN test cases and apply deterministic testing (DT) to CP in line (3) in order to distinguish more mutants.

- The SYN-sequences selected for deterministic testing may need to be infeasible for CP.
- both valid and invalid SYN-sequences should be selected.

A phase two test procedure using selected IN_SYN test cases in line (3) is shown in Fig. 7.12.

```
(1)   repeat {
(2)     Generate mutants (m1,m2,...mn) from CP;
(3)     Apply DT to deterministically execute CP with a selected IN_SYN test
           case (X,S);
(4)     Compare the actual and expected results of this forced execution:
(5)     (a) The results are identical. Then no error is detected by the test (X,S).
(6)     (b) The results differ in the feasibility of S.
(7)     (c) The results agree on the feasibility of S, but not on the termination
            condition of CP.
(8)     (d) The results agree on the feasibility of S and the termination
            condition, but not on the output of CP.
(9)     if (condition (b), (c), or (d) holds) {
(10)        Locate and correct the error in CP using program replay;
(11)        Apply DT to validate the correction by forcing an execution of CP
                with IN_SYN test case (X,S); and restart at (1);
(12)    } else
(13)        for (mutant mi, i<=i<=n) {
(14)          Apply DT to mi with IN_SYN test case (X,S);
(15)          Compare the actual results of the forced executions of CP and
                  mutant mi;
(16)          if (the results differ in the feasibility of S, the termination
                  condition, or the output)
(17)             mark mutant mi as distinguished;
(18)        }
(19) }
(20) until (the mutation score is adequate);
```

Figure 7.12 Deterministic mutation testing (DMT) using deterministic testing (DT) with selected IN_SYN test cases.

Example: Deterministic mutation testing was applied to the correct version of the bounded buffer program, denoted as *boundedBuffer2*.

The result was a set of 95 mutants. Since 14 of the mutations resulted in mutants that were equivalent to *boundedBuffer2*, this left 81 live mutants.

In phase one, we used non-deterministic testing to generate SR-sequences of *boundedBuffer2*.

- Random delays were inserted into *boundedBuffer2* to increase the chances of exercising different SR-sequences during non-deterministic testing.
- The mutation score leveled off at 71%.
- All four valid and feasible sequences of Deposit (D) and Withdraw (W) events had been exercised
  (D,D,W,W,D,W), (D,W,D,D,W,W), (D,W,D,W,D,W), (D,D,W,D,W,W).
- It was not possible to distinguish any more mutants using non-deterministic testing to select SR-sequences of *boundedBuffer2*.

Two of the SR-sequences exercised using non-deterministic testing were modified to produce two new invalid SR-sequences for phase 2:

- (D,D,D,W,W,W) // invalid: three consecutive deposits into a 2-slot buffer
- (W,D,D,W,D,W) // invalid: the first withdrawal is from an empty buffer

Both of these invalid SR-sequences were shown to be infeasible for *boundedBuffer2*, but feasible for the remaining mutants. Thus, all of the remaining mutants were distinguished.

7.5 Reachability Testing

Non-deterministic testing is easy to carry out, but it can be very inefficient. It is possible that some behaviors of a program are exercised many times while others are never exercised at all.

Deterministic testing allows a program to be tested with carefully selected valid and invalid test sequences.

- Test sequences are usually selected from a static model of the program or of the program's design.
- Several coverage criteria for reachability graph models were defined in Section 7.2.2.
- However, accurate static models are difficult to build for dynamic program behaviors.

Reachability testing is an approach that combines non-deterministic and deterministic testing.

Reachability Testing is based on prefix-based testing, which was described in Section 7.4.3.1:

- prefix-based testing controls a test run up to a certain point, and then lets the run continue non-deterministically.
- The controlled portion of the test run is used to force the execution of a prefix SYN-sequence, which is the beginning part of one or more feasible SYN-sequences of the program.
- The non-deterministic portion of the execution randomly exercises one of these feasible sequences.

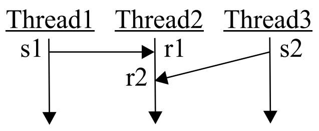
Reachability testing uses prefix-based testing to generate test sequences automatically and on-the-fly as the testing process progresses.

- the SYN-sequence traced during a test run is analyzed to derive prefix SYN-sequences that are “race variants” of the trace.
- A race variant represents the beginning part of a SYN-sequence that definitely could have happened but didn’t, due to the way race conditions were arbitrarily resolved during execution.
- The race variants are used to conduct more test runs, which are traced and then analyzed to derive more race variants, and so on.

If every execution of a program with a given input terminates, and the total number of possible SYN-sequences is finite, then reachability testing will terminate and every partially-ordered SYN-sequence of the program with the given input will be exercised.

7.5.1 The Reachability Testing Process

Assume that an execution of some program CP with input X exercises SYN-sequence Q represented by the space-time diagram in Fig. 7.13.



Send events *s1* and *s2* in Q have a race to see which message will be received first by Thread2.

We can see that there exists at least one execution of CP with input X in which the message sent at *s2* is received by *r1*.

⇒ message sent by *s2* is in the *race set* for r1.

An analysis of sequence Q in Fig. 7.13 allows us to guarantee that *s2* can be received at r1. It does not, however, allow us to guarantee that *s1* can be received at *r2* since we cannot guarantee that Thread2 will always execute two receive statements.

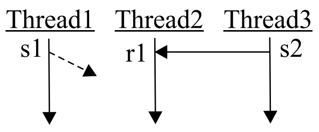
```
Thread2
x = port.receive();    // generates event r1 in Q
if (x>0)
   y = port.receive();// generates event r2 in Q
```

If *r1* receives the message sent by *s2* instead of *s1*, the condition (*x*>*0*) may be false, depending on the value of *s2's* message.

But if the condition (*x*>*0*) is false, the second *receive* statement will not be executed, and since we do not examine CP's code during race analysis, it is not safe to put *s1* in the race set of *r2*.

A race variant represents the beginning part of one or more alternative program paths, i.e., paths that could have been executed if the message races had been resolved differently.

Fig. 7.14 shows the race variant produced for sequence Q in Fig. 7.13.



When this variant is used for prefix-based testing, Thread2 will be forced to receive its first message from Thread3, not Thread1.

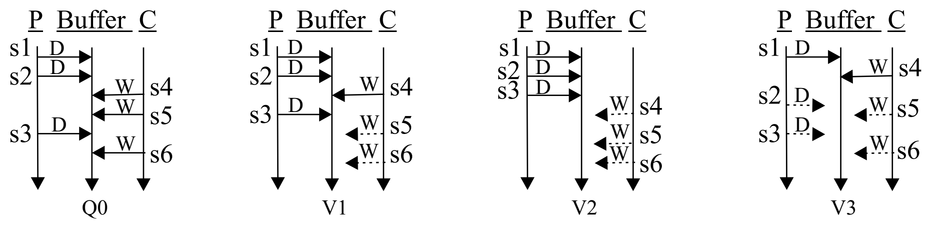
What Thread2 will do after that is unknown:

- Perhaps Thread2 will receive the message sent at *s1*, or perhaps Thread2 will send a message to Thread1 or Thread3.
- The dashed arrow from *s1* indicates that *s1* is not received as part of the variant, though it may be received later.
- In any event, whatever happens after the variant is exercised will be traced, so that new variants can be generated from the trace and new paths can be explored.

Next, we illustrate the reachability testing process by applying it to a solution for the bounded buffer program.

```
    Producer                  Consumer                           Buffer
 (s1) deposit.call(x1);  (s4) item = withdraw.call();   loop
 (s2) deposit.call(x2);  (s5) item = withdraw.call();      select
 (s3) deposit.call(x2);  (s6) item = withdraw.call();         when (buffer is not full) =>
                                                                 item = deposit.acceptAndReply();
                                                                 /* insert item into buffer */
                                                           or
                                                              when (buffer is not empty) =>
                                                                 withdraw.accept();
                                                                 /* remove item from buffer */
                                                                 withdraw.reply(item);
                                                           end select;
                                                        end loop;
```

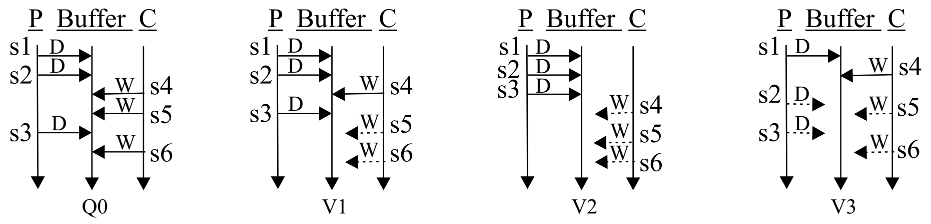
Assume sequence Q0 is recorded during a non-deterministic execution. Sequence Q0 and the three variants derived from Q0 are shown in Fig 7.15.



The variants are derived by changing the order of *deposit* (D) and *withdraw* (W) events whenever there is a message race.

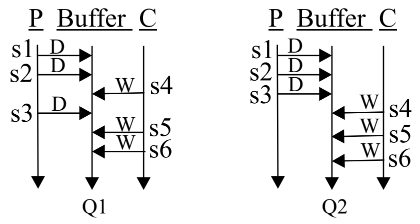
If the message for a receive event *r* is changed, then all the events that happened after *r* are removed from the variant (since we cannot guarantee these events can still occur).

Notice that there is no variant in which the first *receiving event* is for a *withdraw*. Runtime information collected about the guards will show that the guard for *withdraw* was false when the first *deposit* was accepted in Q0. Thus, we do not generate a variant to cover this case.



To create variant *V1* in Fig. 7.15, the outcome of the race between s3 and *s5* in Q0 is reversed. During the next execution of CP, variant *V1* is used for prefix-based testing.

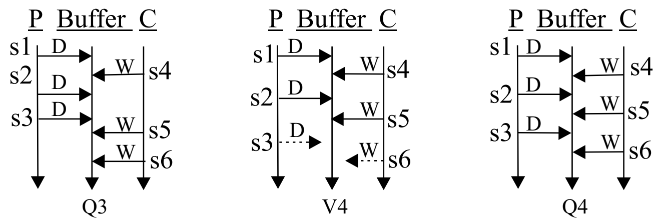
Sequence *Q1* in Fig. 7.16 is the only sequence that can be exercised when V1 is used as a prefix. No new variants can be derived from *Q1*.



To create variant *V2* in Fig. 7.15, the outcome of the race between *s3* and *s4* in *Q0* is reversed. When variant *V2* is used for prefix-based testing, sequence *Q2* in Fig. 7.16 is the only sequence that can be exercised. No new variants can be derived from *Q2*.

To create variant *V3* in Fig. 7.15, the outcome of the race between *s2* and *s4* in *Q0* is reversed. During the next execution of CP, variant *V3* is used for prefix-based testing.

Assume that sequence *Q3* in Fig. 7.17 is exercised. Variant *V4* can be derived from *Q3* by changing the outcome of the race between *s3* and *s5*. Notice that there is no need to change the outcome of the race between *s2* and *s5* in Q3 since the information collected about the guard conditions will show that a withdraw for *s5* cannot be accepted in place of the deposit for *s2*.



During the next execution of CP, variant *V4* is used for prefix-based testing and sequence *Q4* in Fig. 7.17 is the only sequence that can be exercised. Reachability testing stops at this point since *Q0*, *Q1*, *Q2*, *Q3*, and *Q4* are all the possible SYN-sequences that can be exercised by this program.

7.5.2 SYN-sequences for Reachability Testing

In order to perform reachability testing, we need to find the race conditions in a SYN-sequence. The SYN-sequences defined for replay and testing were defined without any concern with identifying races.

For reachability testing, an execution is characterized as a sequence of event pairs:

- For asynchronous and synchronous message-passing programs, an execution is characterized as a sequence of *send* and *receive* events. (For the execution of a synchronous *send* statement, the *send* event represents the start of the *send*, which happens before the message is received.)
- For programs that use semaphores or locks, an execution is characterized as a sequence of *call* and *completion* events for *P*, *V*, *lock*, and *unlock* operations.
- For programs that use monitors, an execution is characterized as a sequence of monitor *call* and monitor *entry* events.

We refer to a *send* or *call* event as a *sending event*, and a *receive*, *completion*, or *entry* event as a *receiving event*.

We refer to a pair <*s,r*> of sending and receiving events as a *synchronization pair*. In the pair <*s*,*r*>, *s* is said to be the sending partner of *r*, and *r* is said to be the receiving partner of *s*.

An arrow in a space-time diagram connects a sending event to a receiving event if the two events form a synchronization pair.

An event descriptor is used to encode certain information about each event:

A descriptor for a sending event *s* is denoted by *(SendingThread, Destination, op, i)*, where

- S*endingThread* is the thread executing the sending event
- D*estination* is the destination thread or object (semaphore, monitor, etc)
- *op* is the operation performed (P, V, send, receive, etc)
- *i* is the event index indicating that *s* is the *i*th event of the *SendingThread*.

A descriptor for a receiving event *r* is denoted by *(Destination, OpenList, i)*, where

- *Destination* is the destination thread or object and *i* is the event index indicating that *r* is the *i*th event of the *Destination* thread or object.
- The *OpenList* contains program information that is used to compute the events that could have occurred besides *r*. Several *OpenList* examples are given below.

The individual fields of an event descriptor are referenced using dot notation. For example, operation *op* of sending event *s* is referred to as *s.op*.

Tables 7.1 and 7.2 summarize the specific information that is contained in the event descriptors for the various synchronization constructs

Synchronization construct	**SendingThread**	**Destination**	**Operation**	**i**
asynchronous message passing	sending thread	port ID	send	event index
synchronous message passing	sending thread	port ID	send	event index
semaphores	calling thread	semaphore ID	P or V	event index
locks	calling thread	lock ID	lock or unlock	event index
monitors	calling thread	monitor ID	method name	event index

Table 7.1 Event descriptors for a sending event *s*.

Synchronization construct	**Destination**	**OpenList**	**i**
asynchronous message passing	receiving thread	the port of r	event index
synchronous message passing	receiving thread	list of open ports (including the port of r)	event index
semaphores	semaphore ID	list of open operations (P and/or V)	event index
locks	lock ID	list of open operations (lock and/or unlock)	event index
monitors	monitor ID	list of the monitor's methods	event index

Table 7.2 Event descriptors for a receiving event *r*.

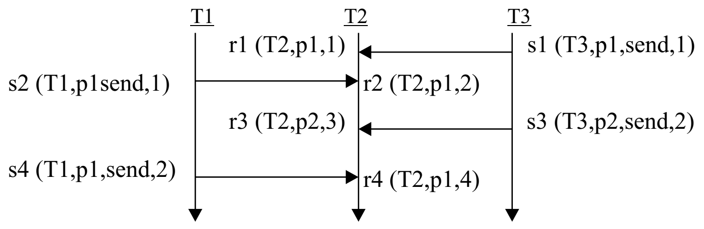
7.5.2.1 Descriptors for asynchronous message passing events.

For asynchronous message-passing, the *OpenList* of a receive event r contains a single port, which is the source port of *r*.

A send event *s* is said to be open at a receive event *r* if port *s.Destination* is in the *OpenList* of *r*, which means that the ports of *s* and *r* match.

In order for a sending event *s* to be in the race set of receive event *r* it is necessary (but not sufficient) for *s* to be open at *r*.

Fig. 7.18 shows a space-time diagram representing an execution with three threads.



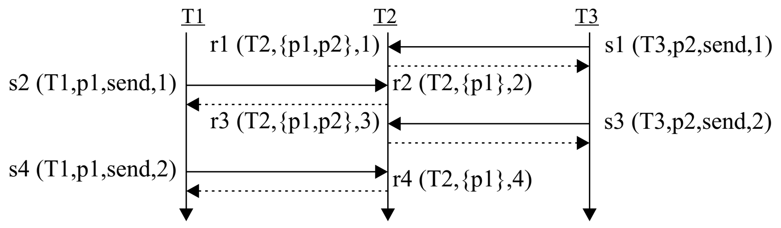
7.5.2.2 Descriptors for synchronous message passing events.

Synchronous message passing may involve the use of selective waits.

The *OpenList* of a receive event *r* is a list of ports that had open receive-alternatives when *r* was selected. Note that this list always includes the source port of *r*.
For a simple receive statement that is not in a selective wait, the *OpenList* contains a single port, which is the source port of the receive statement.

Event *s* is said to be open at *r* if port *s.Destination* is in the *OpenList* of *r*.

Fig. 7.19 shows a space-time diagram representing an execution with three threads.

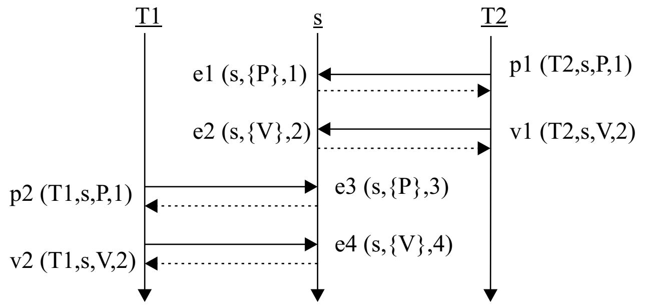


Assume that whenever *p2* is selected, the alternative for *p1* is open, and whenever *p1* is selected, the alternative for *p2* is closed. This is reflected in the *OpenLists* for the receive events, which are shown between braces {…} in the event descriptors.

Note that each solid arrow is followed by a dashed arrow in the opposite direction. The dashed arrows represent the updating of timestamps when the synchronous communication completes. Timestamp schemes are described in Section 7.5.4.

7.5.2.3 Descriptors for semaphore events.

Fig. 7.20 shows an execution involving threads *T1* and *T2* and semaphore *s*, where s is a binary semaphore initialized to 1.



There is one timeline for each thread and each semaphore.

A solid arrow represents the completion of a *P()* or *V()* operation.

The open-lists for the completion events model the fact that P and V operations on a binary semaphore must alternate. This means that the *OpenList* of a completion event for a binary semaphore always contains one of *P* or *V* but not both.

A call event *c* for a P or V operation is open at a completion event *e* if *c* and *e* are operations on the same semaphore, i.e., *c.Destination* = *e.Destination*, and operation *c.op* of *c* is in the *OpenList* of *e*.

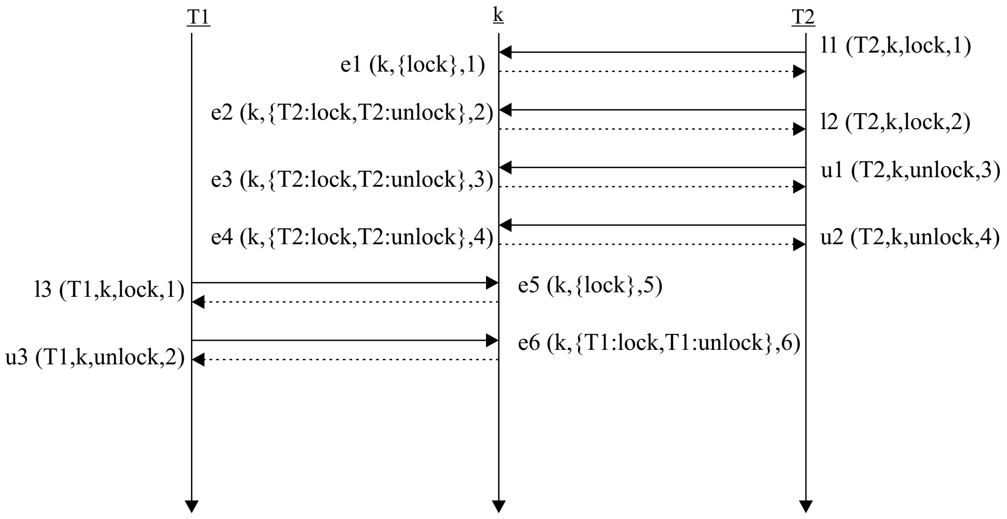
7.5.2.4 Descriptors for lock events.

If a lock is owned by some thread *T* when a completion event *e* occurs, then each operation in the *OpenList* of *e* is prefixed with *T* to indicate that only *T* can perform the operation. (Recall that if a thread *T* owns lock *L*, then only *T* can complete a *lock()* or *unlock()* operation on *L*.)

For example, if the *OpenList* of a completion event *e* on a lock *L* contains two operations *lock()* and *unlock()*, and if *L* is owned by thread *T* when *e* occurs, then the *OpenList* of *e* is *{T:lock, T:unlock}*.

A call event *c* on lock *L* that is executed by thread *T* is open at a completion event *e* if (i) *c.Destination* = *e.Destination*; (ii) operation *c.op* is in the *OpenList* of *e*, and (iii) if *L* is already owned when *e* occurs then *T* is the owner.

Fig. 7.21 shows a space-time diagram representing an execution with two threads and a mutex lock *k*.



The *OpenList* for *e2* reflects the fact that only thread *T2* can complete a *lock()* or *unlock()* operation on *k* since *T2* owns *k* when *e* occurs.

7.5.2.5 Descriptors for monitor events.

The invocation of a monitor method is modeled as a pair of monitor-call and monitor-entry events:

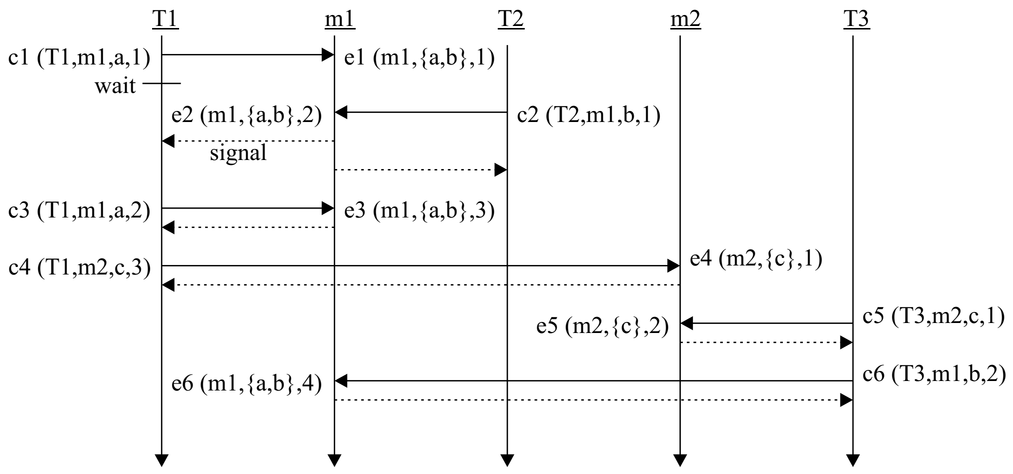
- *SU monitors*: When a thread *T* calls a method of monitor *M*, a monitor-call event *c* occurs on *T*. When *T* eventually enters *M*, a monitor-entry event *e* occurs on *M*, and then *T* starts to execute inside *M*.
- *SC monitors*: When a thread *T* calls a method of monitor *M*, a monitor-call event *c* occurs on *T*. A call event also occurs when *T* tries to reenter a monitor *M* after being signaled. When *T* eventually (re)enters *M*, a monitor-entry event *e* occurs on *M*, and *T* starts to execute inside *M*.

In these scenarios, we say that *T* is the calling thread of *c* and *e*, and *M* is the destination monitor of *c* as well as the owning monitor of *e*.

A call event *c* is open at an entry event *e* if the destination monitor of *c* is the owning monitor of *e*, i.e., *c.Destination* = *e.Destination*.

The *OpenList* of an entry event always contains all the methods of the monitor since threads are never prevented from entering any monitor method (though they must enter sequentially and they may be blocked after they enter).

Fig. 7.22 shows a space-time diagram representing an execution involving three threads *T1*, *T2*, and *T3*, an *SC* monitor *m1* with methods *a()* and *b(),* and an SC monitor *m2* with a single method *c()*.



Note that if *m1* were an *SU* monitor, there would be no *c3* event representing reentry.

7.5.3 Race Analysis of SYN-sequences

To illustrate race analysis, we will first consider a program CP that uses asynchronous ports.

- We assume that the messages sent from one thread to another may be received out of order.
- To simplify our discussion, we also assume that each thread has a single port from which it receives messages.

Let Q be an SR-sequence recorded during an execution of CP with input X.

Assume that *a*□*b* is a synchronization pair in Q, *c* is a send event in Q that is not *a*, and *c*'s message is sent to the same thread that executed b. We need to determine whether sending events *a* and *c* have a race, i.e., whether *c*□*b* can happen instead of *a*□*b* during an execution of CP with input X.

Furthermore, we need to identify races by analyzing Q, not CP.

In order to accurately determine all the races in an execution, the program's semantics must be analyzed. Fortunately, for the purpose of reachability testing, we need only consider a special type of race, called a lead race.

Lead races can be identified by analyzing the SYN-sequence of an execution, i.e., without analyzing the source code.

Definition 6.1: Let *Q* be the *SYN*-sequence exercised by an execution of a concurrent program CP with input X. Let $a \underset{\cdot}{\rightarrow} b$ be a synchronization pair in *Q* and let *c* be another sending event in *Q*. There exists a *lead race* between *c* and <*a, b*> if $c \rightarrow b$ can form a synchronization pair during some other execution of CP with input X, provided that all the events that happened before *c* or *b* in *Q* are replayed in this other execution.

Note that Definition 6.1 requires all events that can potentially affect *c* or *b* in *Q* to be replayed in the other execution.

If the events that happened before *b* are replayed, and the events that happened before *c* are replayed, then we can be sure that *b* and *c* will also occur, without analyzing the code.

Definition 6.2: The *race set* of $a \rightarrow b$ in Q is defined as the set of sending events *c* such that *c* has a (lead) race with $a \underset{\cdot}{\rightarrow} b$ in Q.

We will refer to the receive event in Q that receives the message from *c* as receive event *d*, denoted by $c \rightarrow d$. (If the message from *c* was not received in Q, then *d* does not exist.)

To determine whether $a \rightarrow b$ and *c* in Q have a message race, consider the eleven possible relationships that can hold between *a*, *b*, *c*, and *d* in Q:

(1) $c \Rightarrow d$ and $d \Rightarrow b$
(2) $c \Rightarrow d$, $b \Rightarrow d$, and $b \Rightarrow c$
(3) *c* is send event that is never received and $b \Rightarrow c$
(4) $c \Rightarrow d$, $b \Rightarrow d$, $c \parallel b$, and *a* and *c* are send events of the same thread
(5) $c \Rightarrow d$, $b \Rightarrow d$, $c \parallel b$, and *a* and *c* are send events of different threads
(6) $c \Rightarrow d$, $b \Rightarrow d$, $c \Rightarrow b$, and *a* and *c* are send events of the same threads
(7) $c \Rightarrow d$, $b \Rightarrow d$, $c \Rightarrow b$, and *a* and *c* are send events of different threads
(8) *c* is a send event that is not received, $c \parallel b$, and *a* and *c* are send events of the same thread

(9) *c* is a send event that is not received, *c* || *b*, and *a* and *c* are send events of different threads
(10) *c* is a send event that is not received, *c*⇒*b*, and *a* and *c* are send events of the same thread
(11) *c* is a send event that is not received, *c*⇒*b*, and *a* and *c* are send events of different threads

The happened before relation e⇒*f* was defined in Section 6.3.4.

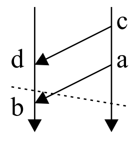
Recall that it is easy to visually examine a space-time diagram and determine the causal relations. For two events *e* and *f* in a space-time diagram, *e*⇒*f* if and only if there is no message *e* ↔ *f* or *f* ↔ *e* and there exists a path from *e* to *f* that follows the vertical lines and arrows in the diagram.

Fig. 7.23 shows eleven space-time diagrams that illustrate these eleven relations.

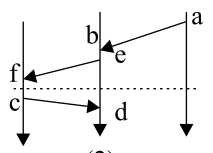
Each of the diagrams contains a curve, called the frontier. Only the events happening before *b* or *c* are above the frontier. (A send event before the frontier may have its corresponding receive event below the frontier, but not vice versa.)

For each of diagrams (4) through (11), if the send and receive events above the frontier are repeated, then events *b* and *c* will also be repeated and the message sent by *c* could be received by *b*. This is not true for diagrams (1), (2), and (3).

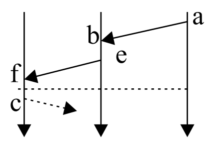
(1) $c \Rightarrow d$ and $d \Rightarrow b$
(2) $c \Rightarrow d$, $b \Rightarrow d$, and $b \Rightarrow c$
(3) c is send event that is never received and $b \Rightarrow c$
(4) $c \Rightarrow d$, $b \Rightarrow d$, $c \parallel b$, and a and c are send events of the same thread
(5) $c \Rightarrow d$, $b \Rightarrow d$, $c \parallel b$, and a and c are send events of different threads
(6) $c \Rightarrow d$, $b \Rightarrow d$, $c \Rightarrow b$, and a and c are send events of the same threads
(7) $c \Rightarrow d$, $b \Rightarrow d$, $c \Rightarrow b$, and a and c are send events of different threads
(8) c is a send event that is not received, $c \parallel b$, and a and c are send events of the same thread
(9) c is a send event that is not received, $c \parallel b$, and a and c are send events of different threads
(10) c is a send event that is not received, $c \Rightarrow b$, and a and c are send events of the same thread
(11) c is a send event that is not received, $c \Rightarrow b$, and a and c are send events of different threads



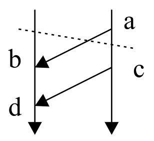
(1)



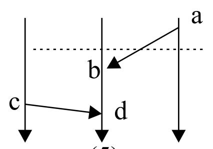
(2)



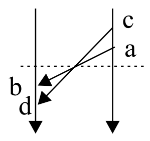
(3)



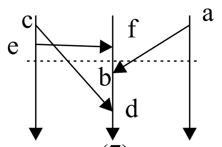
(4)



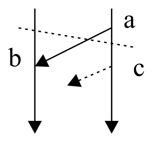
(5)



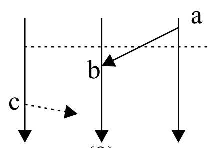
(6)



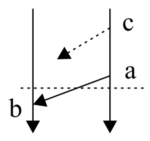
(7)



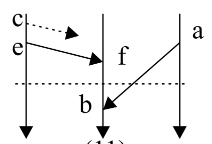
(8)



(9)



(10)

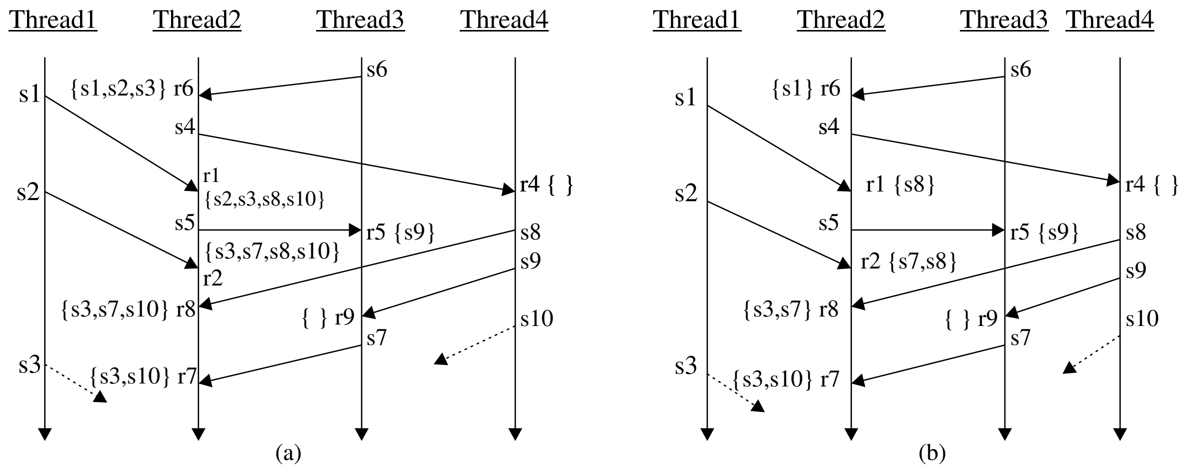


(11)

Based on these diagrams, we can define the race set of $a \rightarrow b$ in Q as follows:

Definition 6.3: Let Q be an SR-sequence of a program using asynchronous communication and let $a \rightarrow b$ be a synchronization pair in Q. The race set of $a \underset{\cdot}{\rightarrow} b$ in Q is $\{c \mid c$ is a send event in Q; c has b*'s* thread as the receiver; not $b \Rightarrow c$; and if $c \underset{\cdot}{\rightarrow} d$ then $b \Rightarrow d\}$.

Fig. 7.24a shows an SR-sequence and the race set for each receive event in this SR-sequence.



Consider send event *s8* in Fig. 7.24a.

- Send event *s8* is received by Thread2 and is in the race sets for receive events *r1* and *r2* of Thread2.

- Send event *s8* is not in the race set for receive event *r6* since *r6* happens before *s8*.

- Send event *s8* is not in the race set for receive event *r7* since *s8*□*r8* but r8⇒r7.

Thus, *s8* is in the race sets for receive events of Thread2 that happen before *r8* but do not happen before *s8*.

The asynchronous ports and mailboxes used in Chapters 5 and 6 are FIFO ports, which means that messages sent from one thread to another thread are received in the order that they are sent.

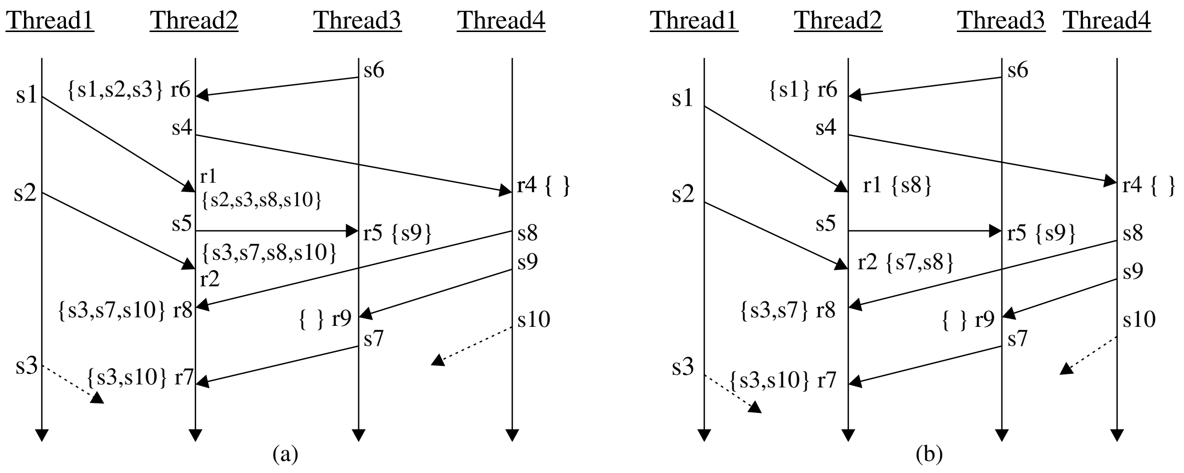
With FIFO ordering, some of relations (1) through (11) above must be modified:

- Relations (4) and (8) no longer have a race between message $a \rightarrow b$ and c
- Relations (6) and (10) are not possible
- Relations (5), (7), (9), and (11) have a race between $a \rightarrow b$ and c if and only if all the messages that are sent from c's thread to b's thread before c is sent are received before b occurs

Thus, the definition of race set must also be modified for FIFO asynchronous SR-sequences.

Definition 6.4: Let Q be an SR-sequence of a program using FIFO asynchronous communication, and let $a \rightarrow b$ be a message in Q. The race set of $a \rightarrow b$ in Q is $\{c \mid c$ is a send event in Q; c has b's thread as the receiver; not $b \Rightarrow c$; if $c \rightarrow d$ then $b \Rightarrow d$; and all the messages that are sent from c's thread to b's thread before c is sent are received before b occurs$\}$.

Fig. 7.24b shows a FIFO asynchronous SR-sequence and the race set for each receive event in this SR-sequence. (Since the asynchronous SR-sequence in Fig. 7.24a satisfies FIFO ordering, it is also used in Fig. 7.24b.)



Consider the non-received send event *s3* in Fig. 7.24b.

- Send event *s3* has Thread2 as the receiver and is in the race sets for receive events *r7* and *r8* in Thread2.
- Thread2 executes *r2* immediately before executing *r8*.
- Since *r2* has the same sender as *s3* and *s2* is sent to Thread2 before *s3* is sent, *s2* has to be received by Thread2 before *s3* is received.

⇒ *s3* is not in the race set for receive event *r2*.

In general, sending and receiving events may involve constructs such as semaphores, locks, and monitors, not just message passing.

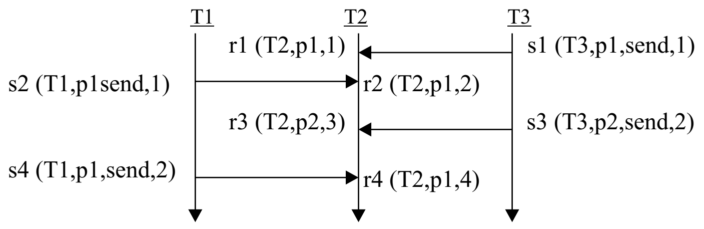
The following definition describes how to compute the race set of a receiving event assuming all the constructs use FIFO semantics.

Definition 6.5: Let *Q* be a SYN-sequence exercised by program CP. A sending event *s* is in the race set of a receiving event *r* if (1) *s* is open at *r*; (2) *r* does not happen before *s*; (3) if <*s, r'*> is a synchronization pair, then *r* happens before *r'*; and (4) *s* and *r* are consistent with FIFO semantics (i.e., all the messages that were sent to the same destination as *s*, and were sent before *s*, are received before *r*).

Below are some examples of race sets:

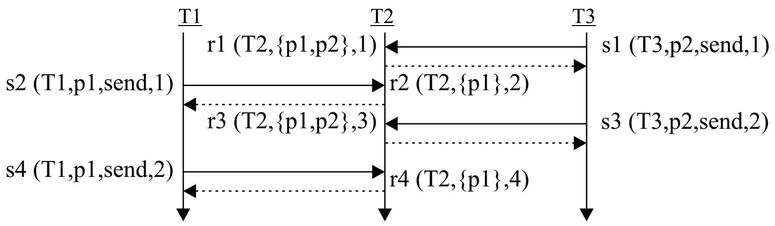
Asynchronous message passing. The race sets for the receive events in Fig. 7.18 are as follows: *race(r1) = {s2}* and *race(r2) = race(r3) = race(r4) = {}*.

- Note that *s3* is not in the race set of *r1* because *s3* is sent to a different port and thus *s3* is not open at *r1*.
- For the same reason, *s4* is not in the race set of *r3*.
- Note also that *s4* is not in the race set of *r1*, because FIFO semantics ensures that *s2* is received before *s4*.



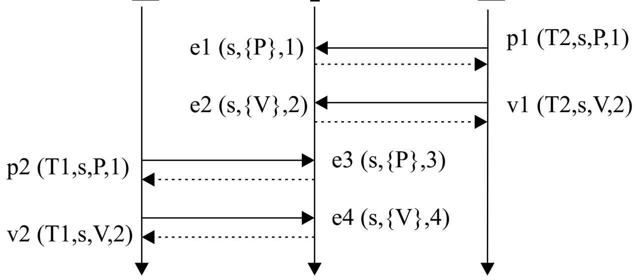
Synchronous message passing. The race sets of the receive events in Fig. 7.19 are as follows: *race(r1) = {s2}, race(r2) = { }, race(r3) = {s4}, and race(r4) = { }.*

- Since the receive-alternative for port *p2* was open whenever thread *T2* selected the receive-alternative for port *p1*, the race set for *r1* contains *s2* and the race set for *r3* contains *s4*.
- Since the receive-alternative for *p1* was closed whenever thread *T2* selected the receive-alternative for *p2*, the race set for *r2* does not contain *s3*.



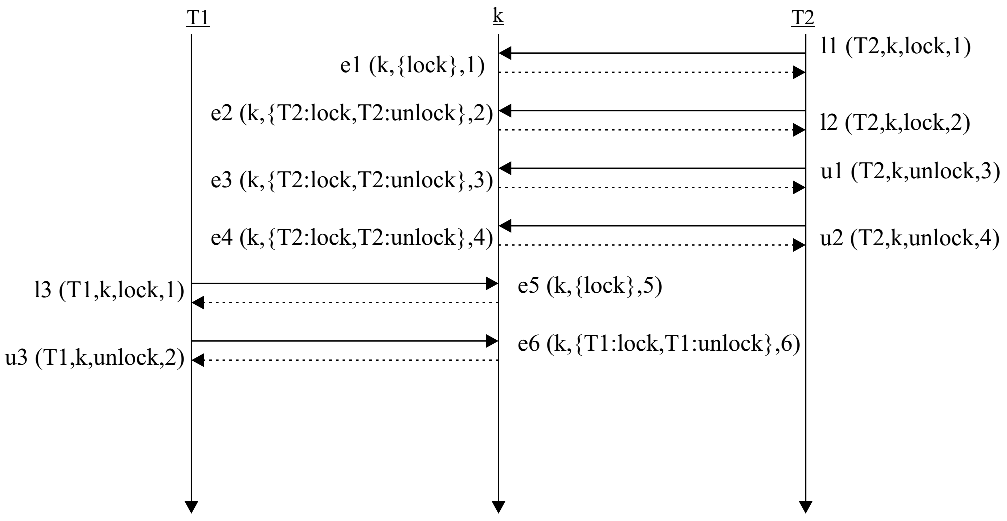
Semaphores. The race sets of the completion events in Fig. 7.20 are as follows: *race(e1) = {p2}* and *race(e2) = race(e3) = race(e4) = { }.*

- Note that since *P()* is not in the *OpenList* of *e2*, the race set for *e2* does not contain *p2*. This captures the fact that the *P()* operation by *T1* could start but not complete before the *V()* operation by *T2* and hence that these operations do not race.



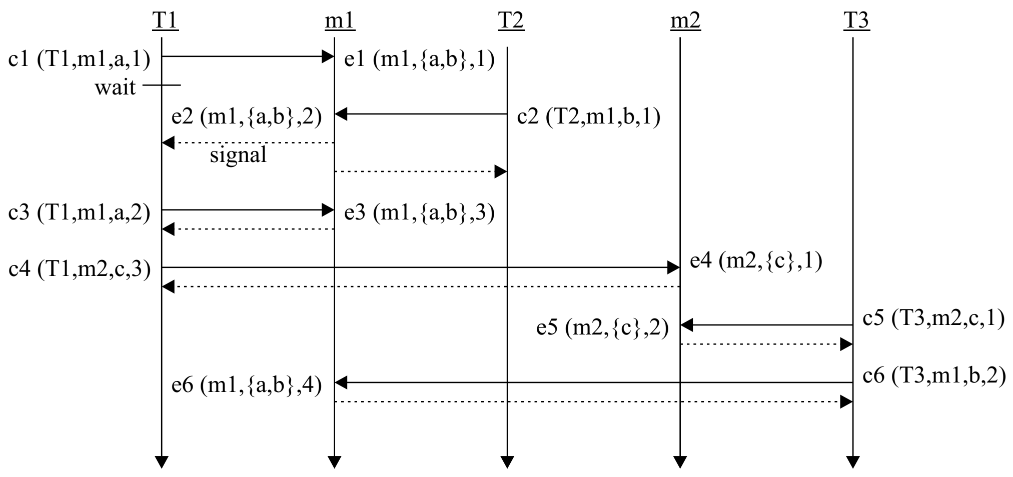
Locks. The race sets of the completion events in Fig. 7.21 are as follows: *race(e1) = {l3}* and *race(e2) = race(e3) = race(e4) = race(e5) = race(e6) = {}*.

- Note that since *T2* owned lock *k* when the operations for events *e2*, *e3*, and *e4* were started, the race sets for *e2*, *e3*, and *e4* are empty. This represents the fact that no other thread can complete a *lock()* operation on *k* while it is owned by *T2*.



Monitors. The race sets of the entry events in Fig. 7.22 are as follows: *race(e1) = {c2}*, *race(e2) = race(e3) = { } race(e4) = {c5}*, and *race(e5) = race(e6) = { }*.

- Sending event *c3* is not in the race set of *e2* since *c3* happened after *e2*. (Thread T2 entered monitor *m* at *e2* and executed a signal operation that caused *T1* to issue the call at *c3*.)



7.5.4 Timestamp Assignment

As we just saw, the definition of a race between sending events is based on the happened-before relation, which was defined in Section 6.3.3. and can be computed using vector timestamps.

A *thread-centric* timestamp has a dimension equal to the number of threads involved in an execution.

An *object-centric* timestamp has a dimension equal to the number of synchronization objects involved.

A thread-centric scheme is preferred when the number of threads is smaller than the number of synchronization objects, and an object-centric scheme is preferred otherwise.

7.5.4.1 A Thread-Centric Scheme.

The vector timestamp scheme described in Section 6.3.5 is thread-centric and can be used for race analysis:

- Each thread maintains a vector clock. A vector clock is a vector of integers used to keep track of the integer clock of each thread.
- The integer clock of a thread is initially zero, and is incremented each time the thread executes a send or receive event.
- Each send and receive event is also assigned a copy of the vector clock as its timestamp.

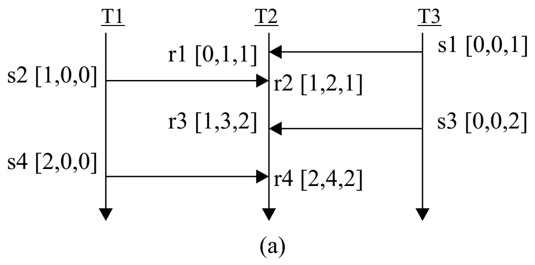
Let *T.v* be the vector clock maintained by a thread *T*. The vector clock of a thread is initially a vector of zeros.

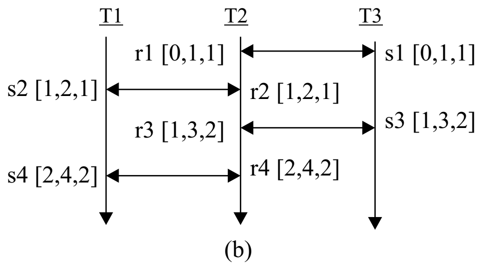
Let *f.ts* be the vector timestamp of event *f*.

The following rules are used to update vector clocks and assign timestamps to the send and receive events in asynchronous message passing programs:

1. When a thread T_i executes a non-blocking send event *s*, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$; (b) $s.ts = T_i.v$. The message sent by *s* also carries the timestamp s.ts.
2. When a thread T_j executes a receive event *r* with synchronization partner *s*, it performs the following operations: (a) $T_j.v[j] = T_j.v[j] + 1$; (b) $T_j.v = max(T_j.v, s.ts)$; (c) $r.ts = T_j.v$.

Fig. 7.25a shows the timestamps for the asynchronous message passing program in Fig. 7.18.





A timestamp scheme for synchronous message passing was also described in Section 6.3.5, but this scheme must be extended before it can be used for race analysis.

The scheme in Section 6.3.5 assigns the same timestamp to send and receive events that are synchronization partners:

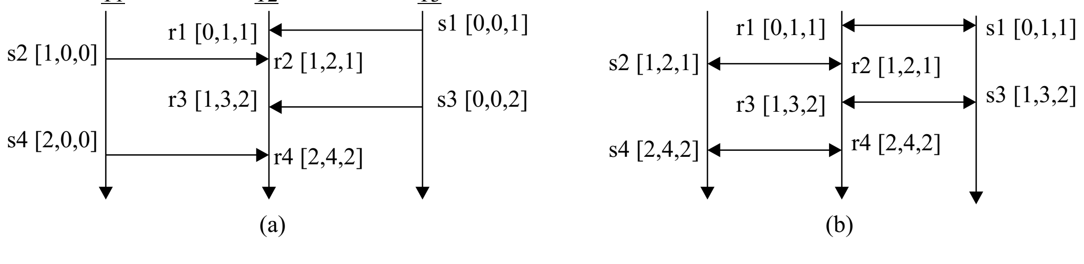
- When a thread T_i executes a blocking send event s, it performs the operation $T_i.v[i] = T_i.v[i] + 1$. The message sent by s also carries the value of vector clock $T_i.v$.

- When a thread T_j executes a receiving event r that receives the message sent by s, it performs the following operations: (a) $T_j.v[j] = T_j.v[j] + 1$; (b) $T_j.v = max(T_j.v, T_i.v)$; (c) $r.ts = T_j.v$. Thread T_j also sends $T_j.v$ back to thread T_i.

- Thread T_i receives $T_j.v$ and performs the following operations (a) $T_i.v = max(T_i.v, T_j.v)$; (b) $s.ts = T_i.v$.

The exchange of vector clock values between threads T_i and T_j represents the synchronization that occurs between them -- their send and receive operations are considered to be completed at the same time.

Fig. 7.25b shows the timestamps for the synchronous message passing program in Fig. 7.19.



In our execution model for synchronous message passing,

- a send event models the start of a send operation, not its completion.

- For send and receive events that are synchronization partners, the start of the send is considered to happen before the completion of the receive

Thus, when a synchronization completes:

- we use the timestamp of the receive event to update the vector clock of the sending thread, which models the synchronization that occurs between the threads.

- we do not use the timestamp of the receive event to update the timestamp of the send event, since the start of the send is considered to happen before the completion of the receive.

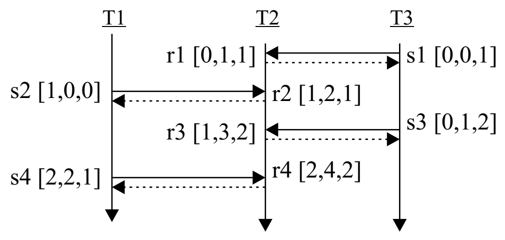
The timestamp scheme synchronous message passing is as follows:

1. When a thread T_i executes a blocking send event *s*, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$. (b) $s.ts = T_i.v$. The message sent by *s* also carries the value of vector clock $T_i.v$.

2. When a thread T_j executes a receiving event *r* that receives the message sent by *s*, it performs the following operations: (a) $T_j.v[j] = T_j.v[j] + 1$; (b) $T_j.v = max(T_j.v, T_i.v)$; (b) $r.ts = T_j.v$. Thread T_j also sends $T_j.v$ back to thread T_i.

3. Thread T_i receives $T_j.v$ and performs the operation: $T_i.v = max(T_i.v, T_j.v)$.

Fig. 7.26 shows the timestamps that are assigned so that race analysis can be performed on the synchronous message passing program in Fig. 7.19.



Note that the dashed arrows represent the application of rule (3).

The timestamps for *s1* and *s2* indicate that these send events were concurrent even though the synchronization between T1 and T2 happened after the synchronization between T3 and T2.

A thread-centric timestamp scheme for semaphores, locks, and monitors.

We refer to semaphores, locks, and monitors generally as “synchronization objects”.

Each thread and synchronization object maintains a vector clock.

- Position *i* in a vector clock refers to the integer clock of thread T_i
- Synchronization objects do not have integer clocks and thus there are no positions in a vector clock for the synchronization objects.

Let *T.v* (or *O.v*) be the vector clock maintained by a thread *T* (or a synchronization object *O*).

The following rules are used to update vector clocks and assign timestamps to events:

1. When a thread T_i executes a sending event *s*, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$; (b) $s.ts = T_i.v$;

2. When a receiving event *r* occurs on a synchronization object *O*, the following operations are performed: (a) $O.v = max(O.v, s.ts)$; (b) $r.ts = O.v$, where *s* is the sending partner of *r*

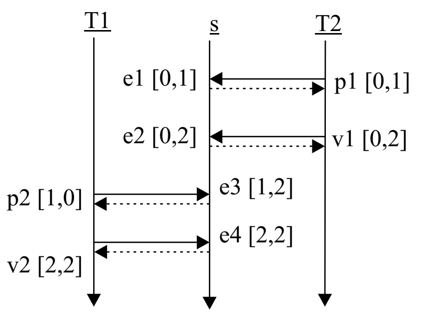
3.

Semaphore/Lock: When a thread T_i finishes executing an operation on a semaphore or lock *O*, it updates its vector clock using the component-wise maximum of $T_i.v$ and *O.v*, i.e., $T_i.v = max(T_i.v, O.v)$.

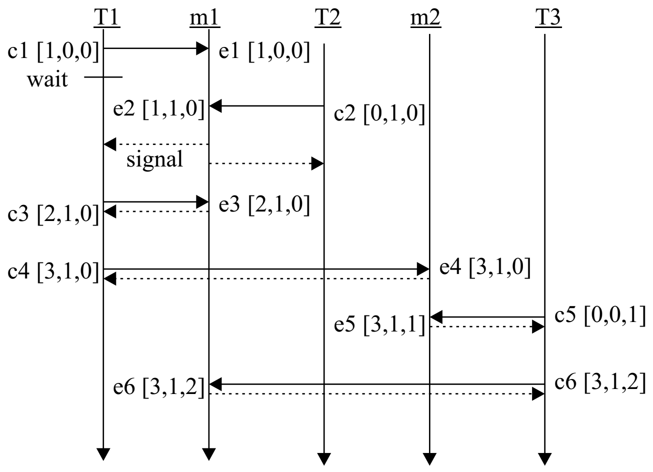
SU monitor: When a thread T_i finishes executing a method on monitor *O*, it updates its vector clock using the component-wise maximum of $T_i.v$ and *O.v*, i.e., $T_i.v = max(T_i.v, O.v)$.

SC monitor: When a thread T_i finishes executing a method on a monitor *O*, or when a thread T_i is signaled from a condition queue of *O*, it updates its vector clock using the component-wise maximum of $T_i.v$ and *O.v*, i.e., $T_i.v = max(T_i.v, O.v)$.

Figs. 7.27a and 7.27b show the thread-centric timestamps assigned for the executions in Figs. 7.20 and 7.22, respectively. Again, dashed arrows represent the application of the third rule.



(a)



(b)

Thread-centric timestamps can be used to determine the happened-before relation between two arbitrary events, as the following Proposition shows:

Proposition 6.1: Let CP be a program with threads $T_1, T_2, \ldots, T_n$ and one or more semaphores, locks, or monitors. Let Q be a *SYN*-sequence exercised by CP. Assume that every event in Q is assigned a thread-centric timestamp. Let *f.tid* be the (integer) thread ID of the thread that executed event *f*, and let f_1 and f_2 be two events in Q. Then, $f_1 \rightarrow f_2$ if and only if (1) $<f_1, f_2>$ is a synchronization pair; or (2) $f_1.ts[f_1.tid] \leq f_2.ts[f_1.tid]$ and $f_1.ts[f_2.tid] < f_2.ts[f_2.tid]$.

7.5.4.2 An object-centric scheme.

Eeach thread and synchronization object (port, semaphore, lock, or monitor) maintains a version vector.

- A version vector is a vector of integers used to keep track of the version number of each synchronization object.
- The version number of a synchronization object is initially zero, and is incremented each time a thread performs a sending or receiving event.
- Each sending and receiving event is also assigned a version vector as its timestamp.

Let *T.v* (or *O.v*) be the version vector maintained by a thread *T* (or a synchronization object *O*). Initially, the version vector of each thread or synchronization object is a vector of zeros.

The following rules are used to update version vectors and assign timestamps to events:

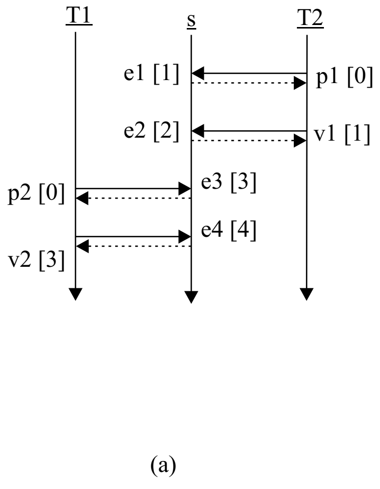
1. When a thread *T* executes a sending event *s*, *T* assigns its version vector as the timestamp of *s*, i.e., *s.ts* = *T.v*;.

2. When a receiving event *r* occurs on a synchronization object *O*, letting *s* be the sending partner of *r*, the following operations are performed: (a) *O.v* = *max(O.v, s.ts)*; (b) *r.ts* = *O.v*.

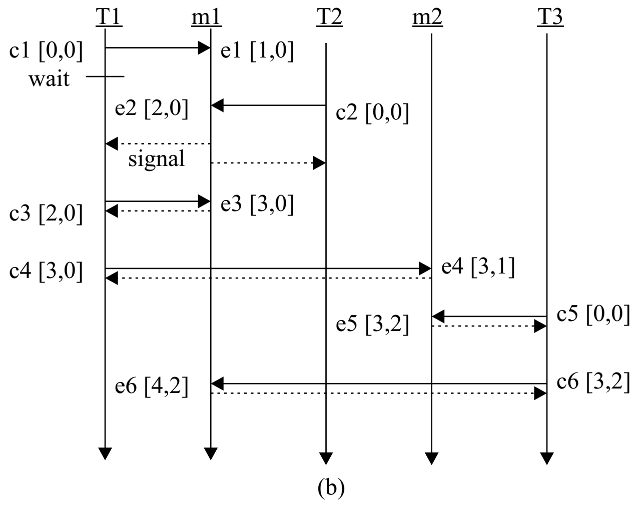
3. Semaphore/Lock: When a thread *T* finishes an operation on a semaphore or lock *O*, *T* updates its version vector using the component-wise maximum of *T.v* and *O.v*, i.e., *T.v = max(T.v, O.v).*
   - *SU* monitor: When a thread *T* finishes executing a method on a monitor *O*, *T* updates its version vector using the component-wise maximum of *T.v* and *O.v*, i.e., *T.v = max(T.v, O.v).*
   - *SC* monitor: When a thread *T* finishes executing a method on a monitor *O*, or when a thread *T* is signaled from a condition queue of *O*, *T* updates its version vector using the component-wise maximum of *T.v* and *O.v*, i.e., *T.v = max(T.v, O.v).*

Timestamps assigned using the above rules are called object-centric timestamps. Note that this scheme is preferred only if the number of synchronization objects is smaller than the number of threads.

Fig. 7.28 shows object-centric timestamps assigned for the executions in Fig 7.27.



(a)



(b)

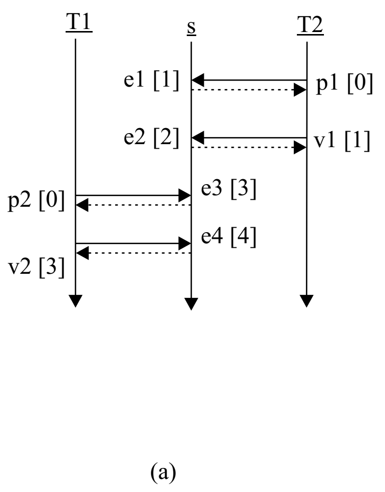
Object-centric timestamps cannot be used to determine the happened-before relation between two arbitrary events.

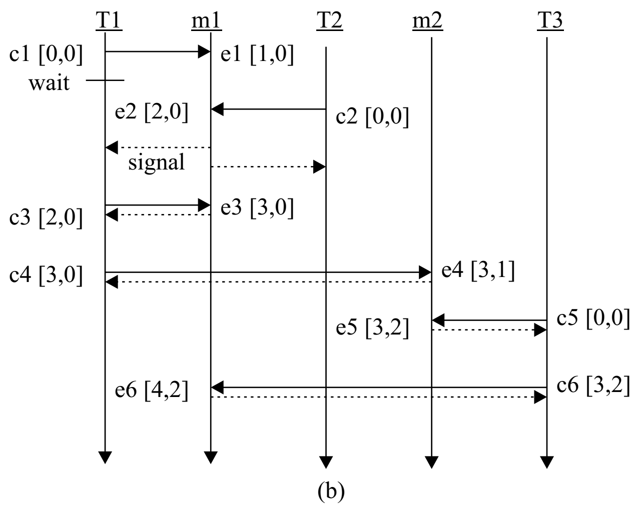
Object-centric timestamps can be used to determine the happened-before relation between two events if at least one of the events is a receiving event, which is sufficient for our purposes.

Proposition 6.2: Let CP be a program that uses synchronization objects $O_1, O_2, \ldots, O_m$, and let Q be a *SYN*-sequence exercised by CP. Assume that every event in Q is assigned an object-centric timestamp. Let e be a receiving event on O_i, and f be a receiving event on O_j, where $1 \leq i, j \leq m$. Then, $e \rightarrow f$ if and only if $e.ts[i] \leq f.ts[i]$ and $e.ts[j] < f.ts[j]$

Proposition 6.3: Let CP be a program that uses synchronization objects $O_1, O_2, \ldots, O_m$, and let Q be the *SYN*-sequence exercised by CP. Assume that every event in Q is assigned an object-centric timestamp. Let r be a receiving event on O_i, and s be a sending event on O_j, where $1 \leq i, j \leq m$. Then, $r \rightarrow s$ if and only if $r.ts[i] \leq s.ts[i]$.

In Fig. 7.28b, monitor entry e_3 happens before entry e_4 since $e_3.ts[1] \leq e_4.ts[1]$ and $e_3.ts[2] < e_4.ts[2]$. Monitor entry e_3 happens before call c_6 since $e_3.ts[1] \leq c_6.ts[1]$.





7.5.5 Computing Race Variants

The race variants of a SYN-sequence Q are computed by constructing a “race table”, where every row in the race table represents a race variant of Q.

Each race variant V of Q is required to satisfy the following three conditions:

1. if we create V by changing the sending partner of receiving event *r*, the new sending partner of *r* must be a sending event in the race set of *r*;

2. if we create V by changing the sending partner of receiving event *r*, then any event *e* that happens after *r* must be removed from V if *e’s* execution can no longer be guaranteed

3. there must be at least one difference between *Q* and *V*.

As an example, consider the race table for sequence Q0 of the bounded buffer program in Section 7.5.1.

Sequence Q0 and its variants are reproduced in Fig. 7.29. The receiving events in Q0 are numbered and shown with their race sets.

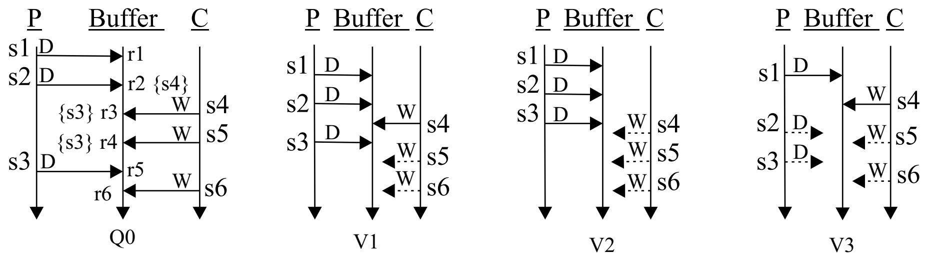


Figure 7-29.

Table 7.3 is the race table for sequence Q0.

	r_2	r_3	r_4
1	0	0	1
2	0	1	-1
3	1	-1	-1

Table 7.3 Race table for Q0.

The three columns represent the three receiving events whose race sets are non-empty. Each row represents a race variant of Q0.

Consider the second row, which is (0, 1, -1). Each value indicates how the sending partner of the corresponding receiving event in Q0 is changed to create variant V2:

1. The value 0 indicates that the sending partner of receiving event r_2 will be left unchanged.
2. The value 1 indicates that the sending partner of receiving event r_3 will be changed to s_3, which is the first (and only) send event in *race*(r_3).
3. The value -1 indicates that receiving event r_4 will be removed from V2.

In general, let *r* be the receiving event corresponding to column *j*, *V* the race variant derived from row *i*, and *v* the value in row *i* column *j*. Value *v* indicates how receiving event *r* is changed to derive variant *V*:

- *v = -1* indicates that *r* is removed from V
- *v = 0* indicates that the sending partner of *r* is left unchanged in V
- *v > 0* indicates that the sending partner of *r* in V is changed to the v^{th} (sending) event in *race(r)*, where the sending events in *race(r)* are arranged in an arbitrary order and the index of the first sending event in *race(r)* is 1.

The receiving events with non-empty race sets are arranged across the columns in left-to-right order with respect to the happened-before relation. (If receiving event *a* happens before receiving event *b* then the column for *a* appears to the left of the column for *b*.)

Conceptually, a race table is a number system, where each row is a number in the system and each column is a digit in a number. In Table 7.3:

- each receiving event has a race set of size 1. Thus, the base of the number system is 2 and each digit (i.e., column value) has the value 0 or 1. The significance of the digits increases from right to left.

	r_2	r_3	r_4
1	0	0	1
2	0	1	-1
3	1	-1	-1

Table 7.3 Race table for Q0.

The rows in the race table are computed iteratively.

Starting with the number 0, all the numbers in the number system are enumerated by adding 1 at each iteration.

Each new number (not including 0) becomes the next row in the table.

Example: For the binary (base 2) system in Table 7.3:

- the first row of the race table is 001.
- adding 1 to this row generates the second row 010
- adding one to the second row generates the third row 011, etc.

The value -1 is used to ensure that each row represents a variant that is a feasible prefix of the program being tested.

To compute the next row in the race table for a SYN-sequence Q, we increment the least significant digit whose value is less than the value of its base minus 1 and whose value is not -1.

Let *t[]* be an array representing the next row in the race table.

We use the following rules to ensure that *t[]* represents a valid race variant V of Q:

1. Whenever we change digit *t[i]* from 0 to 1, which means that the sending partner of receiving event r_i will be changed to create V, we set $t[j] = -1$, $i < j \leq n$, if r_i happens before r_j and r_j is no longer guaranteed to occur. This removes receiving event(s) r_j from V and ensures that V represents a feasible prefix of one or more executions.
2. Let b_i be the base of digit *t[i]*. Whenever we change digit *t[i]* from b_i to 0, which means that the sending partner of r_i will be changed back to r_i's original sending partner in Q, we set $t[j] = 0$, $i < j \leq n$, if the current value of *t[j]* is -1 and there no longer exists an index *k*, $1 \leq k < j$, such that $t[k] > 0$ and r_k happens before r_j.

   In other words, if r_i is the only event causing *t[j]* to be set to *-1* (due to the application of rule (1)) and we change r_i*'s* sending partner back to its original sending partner in Q, then we need to change *t[j]* to 0 so that r_j is no longer removed from V.

3. Whenever we increment *t[i]*, we need to determine whether there exists an index *j* such that $t[j] = m$, $m>0$, and $r_i \rightarrow s$, where *s* is the m^{th} send event in *race*(r_j). Array *t[]* is added to the race table as the next row if and only if such an index *j* does not exist.

   If such an index *j* does exist, then the sending partner of receiving event r_j was previously changed to *s* but since $r_i \rightarrow s$ and the sending partner of r_i has just been changed, we can no longer guarantee that send event *s* will occur.

Example: consider how to add 1 to the number represented by row one in Table 7.3:

- First, we increment the value in the second column (i.e., the column for r_3) which is the right-most column whose value is less than its base minus 1 and is not -1. (The base of the third column is 2, which is one more than the number of send events in the race set of r_4. The value 1 in the third column is not less than 2 minus 1, hence we do not increment the value in the third column.)

- We then apply rule (1), changing the value in the third column to -1 since r_3 happens before r_4 in sequence Q0.

- Rule (2) is not applicable since we did not change the second column from 1 to 0.

- For rule (3), observe that no other column has a value greater than 0, hence changing the sending partner of r_3 does not affect the sending partners of any other receiving events.

Notice that when we change $t[i]$ from 0 to 1 or from b_i to 0, we need only check the values of $t[k]$, $i < k \leq n$, which are the values in the columns to the right of $t[i]$. This is because receiving events are ordered from left-to-right based on the happened before relation.

This ordering also ensures that the value represented by $t[]$ increases at each iteration. Therefore, this iterative process of computing rows will eventually terminate.

Race analysis never produces two sequences that differ only in the order of concurrent events, i.e., two different totally-ordered sequences that have the same partial ordering.

Example: consider a race table that has columns for two concurrent receiving events r_1 and r_2. Three variants will be generated:

- one in which the sending partner of r_1 is changed
- one in which the sending partner of r_2 is changed
- one in which the sending partners of both r_1 and r_2 are changed

No variants are generated to cover the two orders in which r_1 and r_2 themselves can be executed (r_1 followed by r_2, and r_2 followed by r_1). The order in which r_1 and r_2 are executed is not specified by the variants.

7.5.6 A Reachability Testing Algorithm

The objective of reachability testing is to exercise every (partially-ordered) SYN-sequence exactly once during reachability testing.

However, if a newly derived race variant *V* is a prefix of a SYN-sequence *Q* that was exercised earlier, then prefix-based testing with *V* could exercise *Q* again.

Reachability testing algorithms must deal with the potential for duplicate sequences.

One approach to preventing duplicates:
save all the SYN-sequences that are exercised:
a newly derived variant can be used for prefix-based testing only if it is not a prefix of a SYN-sequence that has already been exercised.

For large programs, the cost of saving all of the sequences can be prohibitive, both in terms of the space to store the sequences and the time to search through them.

An alternative approach: identify variants that may cause duplicates and prevent them from being generated.

Some of these variants, however, cannot be prevented, since the best we can say before we execute these variants is that they *might* produce duplicates. In such cases, we allow the suspect variants to be executed, but we prevent duplicate sequences from being collected from them.

A graph-theoretic perspective: All the possible SYN-sequences that could be exercised by program *CP* with input *X* can be organized into a directed graph *G*, which we refer to as the *Sequence/Variant* graph of CP, or simply the *S/V*-graph.

Fig. 7.30a is the S/V-graph for the bounded buffer example in section 7.5.1.

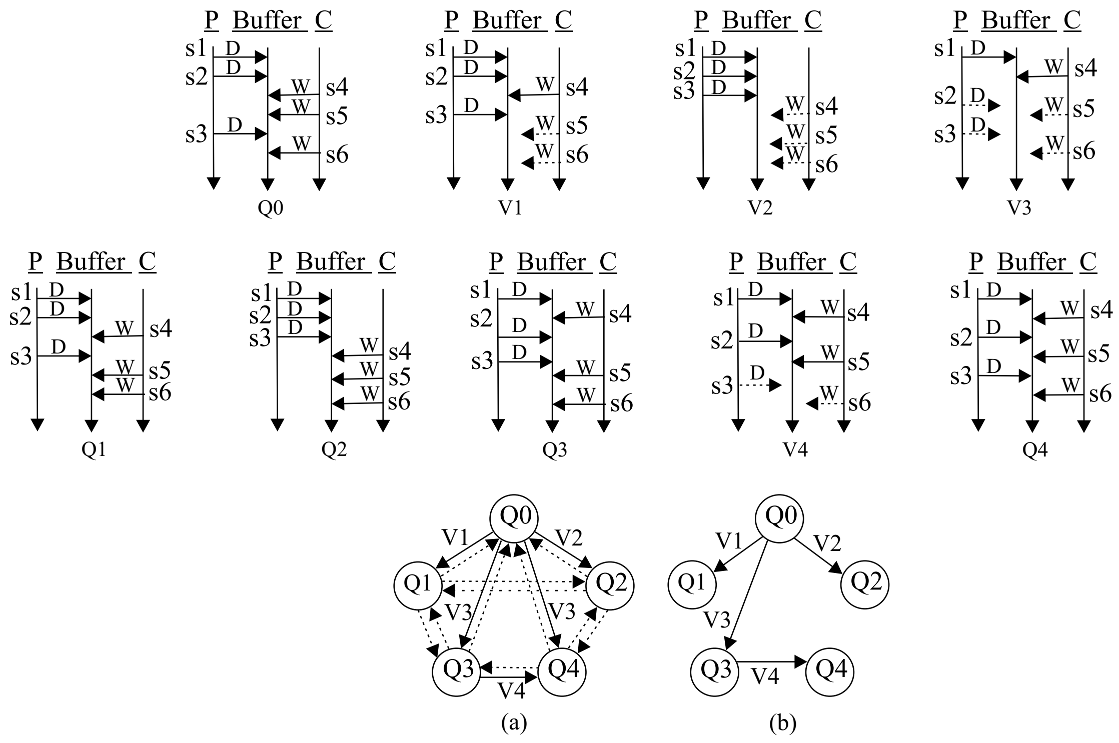


Figure 7.30

Each node *n* in *S/V*-graph *G* represents a SYN-sequence that could be exercised by *CP* with input *X*.

Each edge represents a race variant. An edge labeled *V* from node *n* to node *n'* indicates that sequence *n'* could be exercised by prefix-based testing with the variant *V* derived from sequence *n*.

For example, in Fig. 7.30a node *Q0* has two outgoing edges that are both labeled *V3* since prefix-based testing with variant *V3* may exercise *Q3* or *Q4*.

Note also that an S/V-graph is strongly connected, which means that there is a path in the graph from each node to every other node.

From a graph-theoretic perspective, the goal of reachability testing is to construct a spanning tree of the S/V-graph.

A spanning tree of S/V-graph *G* is a subgraph of *G* that is a tree (i.e., a graph with no cycles) and that connects the *n* nodes of *G* with *n-1* edges (i.e., each node, except the root, has one and only one incoming edge.).

Since S/V-graphs are strongly connected, reachability testing can start from an arbitrary node, which explains why the reachability testing process begins by collecting a sequence during a non-deterministic execution.

Also note that each variant is used to conduct a single test run. Therefore, in a spanning tree that represents the reachability testing process, no two edges should be labeled with the same variant.

Fig. 7.30b shows the spanning tree representing the reachability testing process that was illustrated in Section 7.5.1.

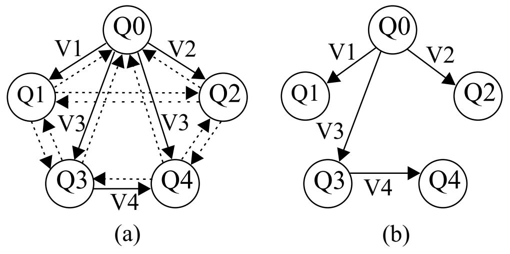


Figure 7.30

A reachability testing algorithm must constrain the way variants are generated so that every sequence is exercised exactly once, i.e., so that the reachability testing process represents a spanning tree of the SV-graph.

The SV-graph is not known when reachability testing begins!

SV-graphs and spanning trees are devices to guide the implementation of, and demonstrate the correctness of, the reachability testing algorithm.

Let *G* be the S/V graph of program *CP* with input *X*. If we can find some constraints on the paths through *G* such that given two arbitrary nodes *n* and *n'* in *G* there is exactly one acyclic path from *n* to *n'* that satisfies these constraints, then we can construct a spanning tree of *G* by enforcing these constraints.

If the reachability testing algorithm implements these constraints, then the reachability testing process will exercise every sequence once.

Node *n* and *n'* are the source node and target node, or the source sequence and target sequence, respectively.

Constraints C1 and C2 below constrain the path between *n* and *n'* such that there is exactly one path *H* that satisfies the constraints.

Constraint C1: The sending partner of a receiving event can be changed only if the receiving event exists in the target sequence and can be changed at most once along a path.

This constraint ensures that path *H* between n and n' is acyclic.

Example: Consider the S/V-graph in Fig. 7.30a.

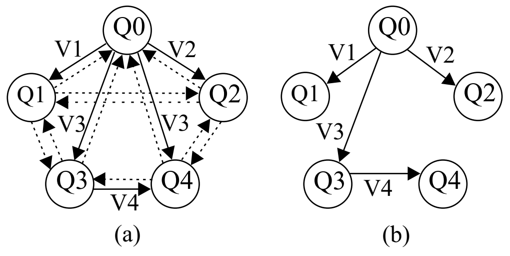


Figure 7.30

A reachability testing process involving the cyclic path *Q0Q1Q0* would not represent a spanning tree of the S/V-graph since trees cannot have cycles.

Such a path would represent a reachability testing process in which sequence Q0 was exercised twice.

Note that receiving event r4 has a different sending partner in *Q0* and *Q1*.

- Variant V1 changes *r4* so that its sending partner is *s3* instead of *s5*. Therefore, the edge from *Q1* to *Q0* must change the sending partner of *r4* back to *s5*.
- This is, however, impossible due to Constraint C1, since the sending partner of *r4* was already changed once in *V1* and is not allowed to be changed again.

Therefore, the cyclic path *Q0Q1Q0* cannot occur during reachability testing.

Constraint C1 can be implemented during reachability testing:

- Associate each receiving event *r* in variant *V* with a color that is either black or white.
- If the sending partner of *r* is changed to derive variant *V*, then *r*'s color is set to black, and this color is inherited by *r* in any sequences collected from V.
- Black receiving events are excluded from race tables (even though they may have non-empty race sets), which prevents the sending partners of black receiving events from being changed again.

Example: In Fig. 7.30a, variant *V1* was derived by changing the sending partner of *r4* (see Fig. 7.29).

- the color of *r4* in *V1* will be black, and this color will be inherited by *r4* in *Q1*.
- *r4* will be excluded from the heading of *Q1*'s race table, preventing the sending partner of *r4* from being changed again when deriving race variants from *Q1*.

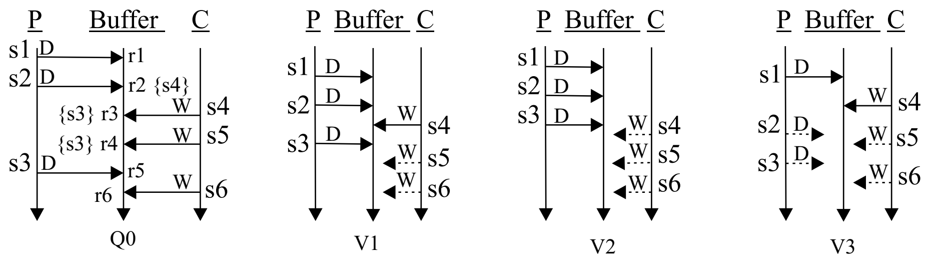


Figure 7.29

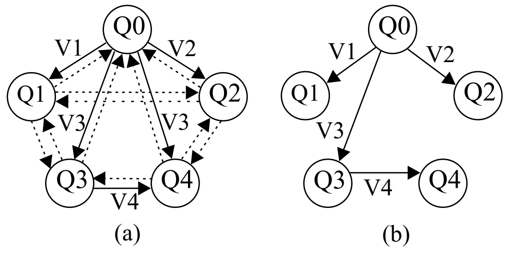


Figure 7.30

Constraint C2: Each edge along a path must reconcile as many differences as possible.

A *difference* between source sequence *n* and target sequence *n*' refers to a receiving event *r* that exists in both sequences but has different sending partners in each sequence.

In terms of these differences, reachability testing can be viewed as the process of transforming, through one or more variants, sequence *n* into sequence *n*'.

Each variant resolves one or more differences between *n* and *n*'.

Constraint C2 says that if there are differences that can be reconciled by an edge, e.g., the sending partner of *r* in *n'* is in the race set of *r* in *n*, then these differences should be reconciled by this edge.

⇒ when deriving a variant V, if there are receiving events whose sending partners can be changed, but are not changed, then these unchanged receiving events cannot be changed afterwards in any sequences derived from V.

Recall that it is common for a variant to leave some receiving events unchanged since all combinations of changed and unchanged receiving events are enumerated in the race table. Constraint C2 ensures that a particular set of changes occurs in only one variant.

Example: consider sequence Q0 and its three variants in Fig. 7.31.

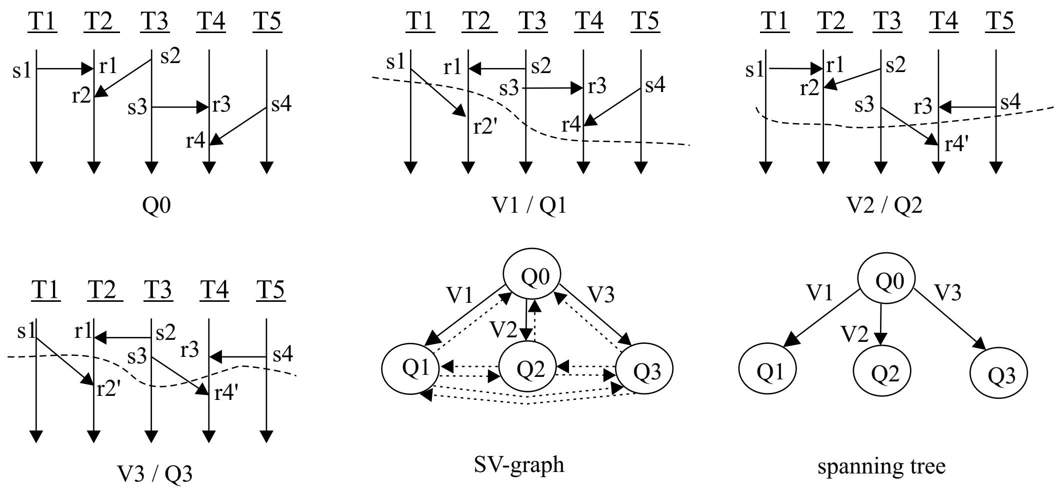


Figure 7.31

Notice that the SV-graph contains paths *Q0Q2Q3* and *Q0Q3*, both of which are paths from *Q0* to *Q3*. Constraint C2 excludes path *Q0Q2Q3* from the spanning tree:

- receiving events *r2* and *r4* exist in *Q0* and also in *Q3*, but the messages they receive in *Q0* are different from the messages they receive in *Q3*.
- edge *V2* along the path *Q0Q2Q3* only changes the sending partner of *r4*, leaving the sending partner of *r2* unchanged.
- The sending partner of *r2* is changed afterwards by the edge from Q2 to Q3.

⇒ path *Q0Q2Q3* violates Constraint C2, which prohibits *r2* from being changed in any sequences derived from V2 since *r2* could have been changed in V2 but wasn't.

Note that edge *V3* of path *Q0Q3* can be included in the spanning tree since it changes the sending partners of both *r2* and *r4*.

Constraint C2 can be implemented during reachability testing by removing *old* sending events from the race sets of *old* receiving events before variants are derived.

A sending or receiving event in a SYN-sequence *VQ* is an old event if it also appears in the variant *V* that was used to collect *VQ*.

Example: consider SYN-sequence *Q2* in Fig. 7.31.

- Events *r1* and *s2* are old events because they appear in both *V2* (the variant that was used to collect *Q2*) and *Q2*.
- Therefore, *s2* will be removed from the race set of *r1* in *Q2*, which means that the sending partner of *r1* cannot be changed to *s2* when the race variants of *Q2* are derived.

⇒ path *Q0Q2Q3* cannot be generated during reachability testing, as in order to reach *Q3* from *Q2*, the sending partner of *r1* must be changed from *s1* to *s2*.

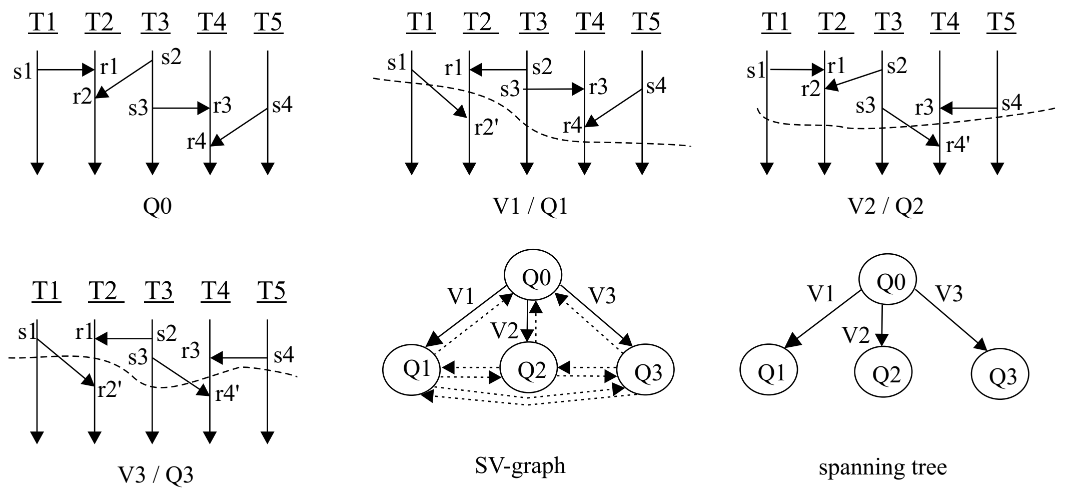


Figure 7.31

Implementing Constraints C1 and C2 is complicated by the possibility that a receiving event may be removed from a variant and then recollected when the variant is used for prefix-based testing.

Fig. 7.32 shows a variant V containing a receiving event *r1* that happens before receiving event *r2*.

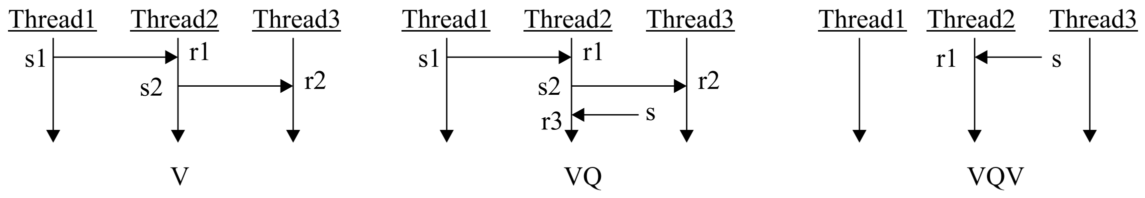


Figure 7.32

Suppose variant *V* is used to collect sequence *VQ* and some thread executes a sending event *s* that is received by Thread2 and is in the race set of *r1* in *VQ*.

When the sending partner of *r1* is changed from *s1* to *s* in order to derive variant *VQV* of *VQ*:

- *r2* will be removed from VQV since *r1* happens before *r2* in *VQ*.
- *r2* will be *recollected* when *VQV* is used for prefix-based testing since Thread3 will definitely execute *r2* again.

In this case, changing the sending partner of *r1* to *s* does not affect the flow of control in Thread3 *before* the point where Thread3 executes *r2* (though possibly after that point).

So we can guarantee that Thread3 will execute *r2* in the sequence collected from variant VQV.

Recollected events like *r2* must be handled carefully. There are two cases to consider:

(1) Event *r2* in V is a black receiving event, indicating that the sending partner of *r2* was changed earlier in the reachability testing process:

When *r2* is recollected during prefix-based testing with *VQV*, it will be recollected as a new, i.e., white, event.

The send partners of white receiving events can be changed. However, Constraint C1 would be violated if we allowed *r2*'s sending partner to be changed when deriving variants of *VQV* since it was already changed earlier.

To prevent a violation of Constraint C1, when *r2*'s color is set to black in V, receiving event *e*'s color in *V* is also set to black for any receiving event *e* that happened before *r2*, such as *r1*.

This prevents *r1*'s sending partner from being changed when deriving variants from *VQ*, which in turn prevents *r2* from being removed from any variant derived from VQ or from any sequence collected afterwards. (Recall that if event *e* is colored black in a variant then *e* inherits that color in any sequence(s) collected from that variant.)

(2) Event *r2* in V is a white receiving event, indicating that the sending partner of *r2* has not been changed yet.

When *r2* is recollected during prefix-based testing with *VQV*, it will be recollected as a white receiving event, but *r2* will also be an old receiving event.

This means that old sending events must be pruned from *r2*'s race set in the sequences collected from variant *VQV*; otherwise, Constraint C2 would be violated when we changed the sending partner of *r2* to an old sending event.

Recollected white receiving events like *r2* are handled as follows:

- When the race table of a sequence like VQ is built, *r2* should not be removed (i.e., set to -1) when variants like *VQV* are created, since *r2* will definitely be recollected.
- Furthermore, in a variant like VQV, which has recollected event *r2* in it, we allow r2's sending partner to be changed just like the other receiving events in the race table.

If, r2's sending partner is changed, then nothing special must be done when the variant is used for prefix-based testing.

If the sending partner of *r2* is not changed, then the sending partner of *r2* must be left unspecified in the variant, since the original sending partner of r2 must be removed.

In this case, *r2* must be prevented from receiving a message from any old sending events when the variant is used for prefix-based testing. This prevents Constraint C2 from being violated.

For example, the sending partner *s2* of *r2* in VQ happens after *r1* in VQ, so *s2* must be removed when the sending partner of *r1* is changed to derive *VQV*.

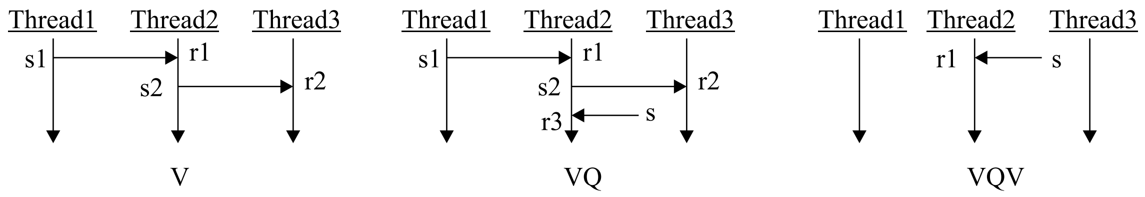


Figure 7.32

Fig. 7.33 shows the algorithm that drives the reachability testing process.

```
ALGORITHM Reachability-Testing (CP: a concurrent program; I: an input of CP) {
    let variants be an empty set;
    collect a SYN-sequence Q0 by executing CP with input X non-deterministically;
    compute the race variants of Q0, variants (Q0), by constructing the race table of Q0
    and enforcing Constraints C1 and C2;
    variants = variants(Q0);
    while (variants is not empty) {
       withdraw a variant V from variants;
       collect a SYN-sequence Q by performing prefix-based testing with V;
       compute the race variants of Q, variants (Q), by constructing the race table of Q and
         enforcing Constraints C1 and C2;
       variants = variants ∪ variants(Q);
    }
}
```

Figure 7.33 A reachability testing algorithm.