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, $0 \leq i \leq n-1$ 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 Ti 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.