3.7 Semaphores and Locks in Win32

Win32 provides four types of objects that can be used for thread synchronization: Mutex, CRITICAL_SECTION, Semaphore, and Event:

- Mutex and *CRITICAL_SECTION* objects are lock objects
- Win32 Semaphores are counting semaphores
- Semaphore, Event, and Mutex objects can be used to synchronize threads in different processes or threads in the same process, but CRITICAL_SECTION objects can only be used to synchronize threads in the same process.

3.7.1 CRITICAL_SECTION

A CRITICAL_SECTION is essentially a Win32 version of the recursive *mutexLock* object described in Section 3.3:

- A thread that calls *EnterCriticalSection*() is granted access if no other thread owns the CRITICAL_SECTION; otherwise, the thread is blocked.
- A thread releases its ownership by calling *LeaveCriticalSection*(). A thread calling *LeaveCriticalSection*() must be the owner of the CRITICAL_SECTION. If a thread calls *LeaveCriticalSection*() when it does not have ownership of the CRITICAL_SECTION, an error occurs that may cause another thread using *EnterCriticalSection()* to wait indefinitely.
- A thread that owns a CRITICAL_SECTION and requests access again is immediately granted access. An owning thread must release a CRITICAL_SECTION the same number of times that it requested ownership, before another thread can become the owner.

Listing 3.18 shows how to use a CRITICAL_SECTION. A CRITICAL_SECTION object must be initialized before it is used and deleted when it is no longer needed.

```
CRITICAL_SECTION cs;                         // global CRITICAL_SECTION

unsigned WINAPI Thread1(LPVOID lpvThreadParm) {
  EnterCriticalSection(&cs);
  // access critical section
  LeaveCriticalSection(&cs);
  return 0;
}

unsigned WINAPI Thread2(LPVOID lpvThreadParm) {
  EnterCriticalSection(&cs);
  // access critical section
  LeaveCriticalSection(&cs);
  return 0;
}

int main() {
  HANDLE threadArray[2];
  unsigned winThreadID;
  InitializeCriticalSection(&cs);
  threadArray[0]= (HANDLE)_beginthreadex(NULL,0, Thread1, NULL,0,
     &winThreadID );
  threadArray[1]= (HANDLE)_beginthreadex(NULL,0, Thread2, NULL,0,
     &winThreadID );
  WaitForMultipleObjects(2,threadArray,TRUE,INFINITE);
  CloseHandle(threadArray[0]);
  CloseHandle(threadArray[1]);
  DeleteCriticalSection(&cs);
  return 0;
}
```

Listing 3.18 Using a Win32 CRITICAL_SECTION.

Listing 3.19 shows class *win32Critical_Section*, which is a wrapper for CRITICAL_SECTION objects.

The CRITICAL_SECTION member *cs* is initialized when a *win32Critical_Section* object is constructed and deleted when the object is destructed.

```
class win32Critical_Section {
// simple class to wrap a CRITICAL_SECTION object with lock/unlock operations
private:
    CRITICAL_SECTION cs;
public:
    win32Critical_Section () { InitializeCriticalSection(&cs); }
    ~ win32Critical_Section () { DeleteCriticalSection(&cs);}
                                void lock() { EnterCriticalSection(&cs); }
    void unlock() { LeaveCriticalSection(&cs);}
};
```

Listing 3.19 C++ class *win32Critical_Section.*

Class *win32Critical_Section* can be used to create lockable objects:

```
    class lockableObject {
    public:
       void F() {
           mutex.lock();
           ...;
           mutex.unlock();
       }
       void G() {
          mutex.lock();
          ...; F(); ...;            // method G() calls method F()
          mutex.unlock();
       }
    private:
       ...
       win32Critical_Section mutex;
    };
```

A better approach to creating lockable objects is to take advantage of C++ semantics for constructing and destructing the local (automatic) variables of a method.

Listing 3.20 shows class template *mutexLocker*<> whose type parameter *lockable* specifies the type of lock (e.g., *win32Critical_Section)* that will be used to create a lockable object:

The constructor and destructor for *mutexLocker* are responsible for locking and unlocking the *lockable* object that is stored as data member *aLockable*.

```
template<class lockable> class mutexLocker {
public:
    mutexLocker(lockable& aLockable_) : aLockable(aLockable_) {lock();}
    ~mutexLocker() { unlock();}
    void lock() {aLockable.lock();}
    void unlock() {aLockable.unlock();}
private:
    lockable& aLockable;
};
```

Listing 3.20 C++ class template *mutexLocker*.

To make a method a critical section, begin the method by creating a *mutexLocker* object.

```
class lockableObject {
public:
   void F() {
      mutexLocker< win32Critical_Section > locker(mutex);
      …
   }
   void G() {
      mutexLocker< win32Critical_Section > locker(mutex);
      …; F(); …;          // this call to F() is inside a critical section
   }
private:
   ...
   win32Critical_Section mutex;
};
```

The locking and unlocking of *mutex* will occur automatically as part of the normal allocation and deallocation of local variable *locker*:

- When *locker* is constructed, *mute.lock()* is called.
- When *locker* is destructed, *mutex.unlock()* is called.

It is now impossible to forget to unlock the critical section when leaving *F()* or *G()*.

Furthermore, the destructor for *locker* will be called even if an exception is raised.

3.7.2 Mutex

A Mutex is a recursive lock with behavior similar to that of a CRITICAL_SECTION. Operations W*aitForSingleObject()* and *ReleaseMutex()* are analogous to *EnterCriticalSection()* and *LeaveCriticalSection()*, respectively:

- A thread that calls W*aitForSingleObject*() on a Mutex is granted access to the Mutex if no other thread owns the Mutex; otherwise, the thread is blocked.
- A thread that calls W*aitForSingleObject*() on a Mutex and is granted access to the Mutex becomes the owner of the Mutex.
- A thread releases its ownership by calling *ReleaseMutex*(). A thread calling *ReleaseMutex*() must be the owner of the Mutex.
- A thread that owns a Mutex and requests access again is immediately granted access. An owning thread must release a Mutex the same number of times that it requested ownership, before another thread can become the owner.

Mutex objects have the following additional features:

- A timeout can be specified on the request to access a Mutex.
- When the Mutex is created, there is an argument that specifies whether the thread that creates the Mutex object is to be considered as the initial owner of the object.

Listing 3.21 shows how to use a Mutex object. You create a Mutex by calling the *CreateMutex()* function. The second parameter indicates whether the thread creating the Mutex is to be considered the initial owner of the Mutex. The last parameter is a name that is assigned to the Mutex.

```
HANDLE hMutex = NULL;                                    // Global
Mutex
unsigned WINAPI Thread1(LPVOID lpvThreadParm) {
  // Request ownership of mutex.
  DWORD rc =::WaitForSingleObject(
    hMutex,                    // handle to the Mutex
    INFINITE);                 // wait forever (no timeout)
  switch(rc) {
    case  WAIT_OBJECT_0:       // wait completed successfully
          break;               // received ownership
    case  WAIT_FAILED:         // wait failed
          // received ownership but the program's state is unknown
    case  WAIT_ABANDONED:
    case  WAIT_TIMEOUT: // timeouts impossible since INFINITE used
          PrintError("WaitForSingleObject failed at ",__FILE__,__LINE__);
          // see Listing 1.3 for PrintError().
          break;
  }
  // Release ownership of mutex
  rc = ::ReleaseMutex(hMutex);
  if (!rc) PrintError("ReleaseMutex failed at ",__FILE__,__LINE__);
  return 0;
}
unsigned WINAPI Thread2(LPVOID lpvThreadParm) { /* same as Thread1 */ }
int main() {
HANDLE threadArray[2];
  unsigned threadID;
  hMutex = CreateMutex(
    NULL, // no security attributes
    FALSE,     // this mutex is not initially owned by the creating thread
    NULL);     // unnamed mutex that will not be shared across processes
  threadArray[0]= (HANDLE)_beginthreadex(NULL,0,Thread1,NULL,0,
     &threadID );
  threadArray[1]= (HANDLE)_beginthreadex(NULL,0, Thread2, NULL,0,
     &threadID );
  WaitForMultipleObjects(2,threadArray,TRUE,INFINITE);
  CloseHandle(threadArray[0]); // release references when finished with them
  CloseHandle(threadArray[1]);     CloseHandle(hMutex);
  return 0;
}
```

Listing 3.21 Using Win32 Mutex objects.

Unlike a CRITICAL_SECTION, a Mutex object is a kernel object that can be shared across processes.

The fact that Mutex objects are kernel objects means that CRITICAL_SECTIONS may be faster than Mutexes:

- If a thread executes *EnterCriticalSection()* on a CRITICAL_SECTION when the CRITICAL_SECTION is not owned, an atomic *interlocked* test is performed (see Sections 2.2.1 and 2.3) and the thread continues without entering the kernel.
- If the CRITICAL_SECTION is already owned by a different thread, then the thread enters the kernel and blocks. A call to W*aitForSingleObject*() on a Mutex always enters the kernel.

If a thread terminates while owning a Mutex, the Mutex is considered to be "abandoned".

- When this happens, the system will grant ownership of the Mutex to a waiting thread.
- The thread that becomes the new owner receives a return code of WAIT_ABANDONED.

3.7.3 Semaphore

Win32 Semaphores are counting semaphores. Operations W*aitForSingleObject*() and *ReleaseSemaphore()* are analogous to *P()* and *V()*, respectively.

When a Semaphore is created, its initial and maximum values are specified:

- initial value must be greater than or equal to zero and less than or equal to the maximum value.
- maximum value must be greater than zero.
- the value of the Semaphore can never be less than zero or greater than the specified maximum value.

```
HANDLE hSemaphore;
hSemaphore = CreateSemaphore(
   NULL,          // no security attributes
   1L,            // initial count
   LONG_MAX,      // maximum count (defined in C++ as at least 2147483647)
   NULL);         // unnamed semaphore
if (!hSemaphore) PrintError("CreateSemaphore",__FILE__,__LINE__);

DWORD rc = WaitForSingleObject(
   hSemaphore,    // handle to semaphore
   INFINITE);     // no time-out
switch(rc) {
   case  WAIT_OBJECT_0:
           break;                    // wait completed successfully
   case  WAIT_FAILED:
   case  WAIT_TIMEOUT:     // no timeouts possible since INFINITE was used
           PrintError("WaitForSingleObject failed at ",__FILE__,__LINE__);
           break;
}
rc = ReleaseSemaphore(
   hSemaphore,// handle to semaphore
   1,             // increase count by one
   NULL);         // not interested in previous count
if (!rc) PrintError("Release Semaphore failed at ",__FILE__,__LINE__);
```

- The second argument for *ReleaseSemaphore()* specifies how much to increment the value of the Semaphore.
- The last argument for *ReleaseSemaphore()* is the address of a long value that will receive the value of the Semaphore's count *before* incrementing the count.

When you are finished with the semaphore call *CloseHandle()* to release your reference to it:

```
CloseHandle(hSemaphore);
```

A simple C++ wrapper class called *win32Semaphore* is shown in Listing 3.22.

```
#include <windows.h>
#include <limits.h>
const int maxDefault = LONG_MAX; //defined in C++ as at least 2147483647
class win32Semaphore {
private:
   HANDLE hSemaphore;
   int initialValue; int maxValue;
public:
   void P();
   DWORD P(long timeout);
   void V();
   win32Semaphore(int initial);
   win32Semaphore(int initial, int max);
   ~win32Semaphore();
};

win32Semaphore :: win32Semaphore(int initial) : initialValue(initial),
maxValue(maxDefault) {
   hSemaphore = CreateSemaphore(
      NULL,        // no security attributes
      initial,     // initial count
      maxValue,    // maximum count
      NULL);       // unnamed semaphore
   if (!hSemaphore) PrintError("CreateSemaphore",__FILE__,__LINE__);
}
```

```
win32Semaphore :: win32Semaphore(int initial, int max) : initialValue(initial),
         maxValue(max) {
   hSemaphore = CreateSemaphore(
      NULL, initial,maxValue, NULL);
   if (!hSemaphore)        PrintError("CreateSemaphore",__FILE__,__LINE__);
}

win32Semaphore :: ~win32Semaphore() {
   DWORD rc = CloseHandle(hSemaphore);
   if (!rc)      PrintError("CloseHandle",__FILE__,__LINE__);
}

void win32Semaphore :: P() {
   DWORD rc = WaitForSingleObject(
      hSemaphore,// handle to semaphore
      INFINITE);  // no time-out
   if (!rc)      PrintError("WaitForSingleObject",__FILE__,__LINE__);
}

DWORD win32Semaphore :: P(long timeout) {
   DWORD rc = WaitForSingleObject(hSemaphore,        timeout);      // no time-out
   if (!(rc==WAIT_OBJECT_0)||(rc==WAIT_TIMEOUT))
      PrintError("WaitForSingleObject failed at ",__FILE__,__LINE__);
        return rc;
}

void win32Semaphore :: V() {
   DWORD rc = ReleaseSemaphore(
   hSemaphore,     // handle to semaphore
   1,   // increase count by one
   NULL);   // not interested in previous count
   if (!rc)      PrintError("ReleaseSemaphore failed at ",__FILE__,__LINE__);
}
```

Listing 3.22 Class *win32Semaphore.*

3.7.3.1 Class mutexLock.

Class *mutexLock* in Listing 3.23 guarantees FCFS notifications.

- When a thread needs to wait in method *lock()*, it acquires a *win32Semaphore* from a pool of available semaphores, inserts the semaphore into a FCFS queue, and executes a *P()* operation on the semaphore.
- In method *unlock()*, ownership of the *mutexLock* is passed to a waiting thread by performing a *V()* operation on the semaphore at the front of the queue.
- The unblocked thread returns its *win32Semaphore* to the pool by calling *release()*.

The FCFS queue of semaphores maintained by class *mutexLock* guarantees FCFS notifications for *mutexLock* objects.

- Each thread that is blocked in method *lock()* is blocked on a *win32Semaphore* in the queue.
- The *unlock()* operation unblocks the thread that has been blocked the longest.

Acquiring semaphores from a semaphore pool makes it possible to reuse *win32Semaphores* instead of creating a new one each time a thread blocks:

- The semaphore pool initially contains a single semaphore.
- If an attempt is made to acquire a semaphore when the pool is empty, the *acquire()* method creates another semaphore and adds it to the pool.

This type of “resource pooling” is commonly used for resources that are expensive to create.

3.7.3.2 Class countingSemaphore.

Class *countingSemaphore* in Listing 3.24 uses a FCFS queue of *win32Semaphores* to implement FCFS notifications just as *mutexLock* did.

Class *countingSemaphore* also provides an implementation of the *VP()* operation. An execution of *t.VP(s)* performs a *V()* operation on *s* followed by a *P()* operation on *t*.

The *VP()* operation must lock both semaphores before either operation is begun. This locking must be done carefully.

Thread 1	Thread 2
t.VP(s);	s.VP(t);

If Thread 1 succeeds in locking *s* while Thread 2 succeeds in locking *t*, then neither thread will be able to lock its other semaphore, resulting in a deadlock.

To prevent this circular waiting condition from occurring, the locks for the two semaphores are always acquired in the same order. This is accomplished by giving each semaphore a unique ID and forcing the *VP()* operation to lock the semaphore with the lowest ID first.

The *V()* part of a *VP()* operation must satisfy the requirement that it will not block. This is checked by *VP()*.

- If a *V()* is attempted on a binary semaphore whose *permits* value is 1, the *VP()* operation fails.
- If the *P()* part of the operation is required to block, *VP()* releases both semaphore locks and blocks the calling thread.

```
class countingSemaphore : public semaphore {
// a countingSemaphore with FCFS notifications
private:
   std::queue<win32Semaphore*> waitingP;// queue of threads blocked on P
   bool doP();
   semaphorePool pool;              // pool of semaphores
public:
   countingSemaphore(int initialPermits);
   void P();
   void V();
   void VP(semaphore* vSem);
};

countingSemaphore::countingSemaphore(int initialPermits):semaphore(initialPermits){}

void countingSemaphore::P() {
   lock();                    // lock is inherited from class semaphore
   --permits;
   if (permits>=0) {unlock(); return; }
   // each thread blocks on its own semaphore
   win32Semaphore* s =  pool.acquire();
   waitingP.push(s);                // append the semaphore
   unlock();
   s->P();                          // block on the semaphore
   pool.release(s);
}

void countingSemaphore::V() {
// each thread blocks on its own semaphore
   lock(); // lock semaphore
   ++permits;
   if (permits>0) { unlock(); return; }
   win32Semaphore* oldest = waitingP.front();
   waitingP.pop();
   oldest->V();
   unlock(); // end synchronized(this) to avoid doing s.P() while
}          // holding the lock on this semaphore
```

```
bool countingSemaphore::doP() {
// Called by VP() operation; checks permits and returns true if P() should block;
// false otherwise.
   --permits;
   if (permits>=0)  return false; // P() does not block
   else return true; // P() does block
}

void countingSemaphore::VP(semaphore* vSem) {
// Execute {vSem->V(); this->P();} without any intervening  P() or V() operations
// on this or vSem.   Return 0 if this operation fails.
// Lock the semaphores in ascending order of IDs to prevent circular deadlock (i.e.
// T1 holds the lock of this and waits for vSem's lock while T2 holds vSem's lock
// and waits for the lock of this.)
   semaphore* first = this; semaphore* second = vSem;
   if (this->getSemaphoreID() > vSem->getSemaphoreID()) {
      first = vSem; second = this;
   }
   first->lock(); second->lock();
   //vSem.V() must not block
   if (vSem->permits==1 && vSem->isBinarySemaphore()) {
     // method isBinarySemaphore() is inherited from class semaphore
      std::cout << "VP failed at line " << __LINE__ << " in " << __FILE__
         << " with error: V operation will block." << std::endl;
      exit(1);
   }
   // perform vSem.V()
   vSem->V(); //  okay to already hold vSem's lock (which is first or second)
              //       since it is a recursive lock
   // perform this->P()
   bool blockingP = doP();
   if (!blockingP) {second->unlock(); first->unlock();}
   // each thread blocks on own semaphore
   win32Semaphore* s =  pool.acquire();
   waitingP.push(s);            // otherwise append blocked thread
   second->unlock();            // unlock semaphores before blocking
   first->unlock();
   s->P();              // s is already in waitingP so FCFS is enforced
   pool.release(s);
}
```
Listing 3.24 C++/Win32 class *countingSemaphore.*

3.7.4 Events

One thread can signal the occurrence of an activity or event to one or more other threads using a Win32 Event object.

- An Event can be either a manual-reset or auto-reset Event.
- The state of an Event is either “signaled” or “nonsignaled”.
- When an Event is created, the initial state (signaled or non-signaled) and the type (manual-reset or auto-reset) is specified.

Signaling an event:

- When the state of a manual-reset Event object is set to signaled, it remains signaled until it is explicitly reset to non-signaled by the *ResetEvent*() function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified Event object, can be released while the object's state is signaled.
- When the state of an auto-reset Event object is set to signaled, it remains signaled until a single waiting thread is released; the system then automatically resets the state to non-signaled.

The state of an Event is changed using *SetEvent()*, *ResetEvent()*, or *PulseEvent()*:

- *SetEvent()*:
  - For an auto-reset event, *SetEvent()* sets the state to signaled until one waiting thread is released. That is, if one or more threads are waiting one will be released and the state will be reset to nonsignaled. If no threads are waiting, the state will stay signaled until one thread waits, at which time the waiting thread will be released and the state will be returned to nonsignaled.
  - For a manual-reset event, all waiting threads are released and the state remains signaled until it is reset by *ResetEvent()*. Setting an event that is already in the signaled state has no effect.

- *ResetEvent()* sets the state to nonsignaled (for both manual-reset and auto-reset Events). Resetting an event that is already in the nonsignaled state has no effect
- *PulseEvent()*
    - For a manual-reset event, *PulseEvent()* sets the state to signaled, wakes up all waiting threads, then returns the state to nonsignaled.
    - For an auto-reset event, *PulseEvent()* sets the state to signaled, wakes up a single waiting thread (if one is waiting), then returns the state to nonsignaled. If no threads are waiting, *PulseEvent()* simply sets the state to nonsignaled and returns.

We used a manual-reset Event in the implementation of the C++/Win32 *Thread* class from Chapter 1. Listing 3.25 shows several methods of class *Thread*.

```
Thread::Thread(std::auto_ptr<Runnable> runnable_) : runnable(runnable_) {
    if (runnable.get() == NULL)
        PrintError("Thread(std::auto_ptr<Runnable> runnable_) failed at ",
            __FILE__,__LINE__);
    completionEvent = CreateEvent(
        NULL,       // no security attributes
        1,          // manual reset Event
        0,          // initially nonsignaled
        NULL);      // unnamed event

    hThread = (HANDLE)_beginthreadex(NULL,0, Thread::startThreadRunnable,
        (LPVOID)this, CREATE_SUSPENDED, &winThreadID );
    if (!hThread) PrintError("_beginthreadex failed at ",__FILE__,__LINE__);
    }
}

unsigned WINAPI Thread::startThread(LPVOID pVoid) {
    Thread* aThread = static_cast<Thread*> (pVoid);
    assert(aThread);
    aThread->result = aThread->run();
    aThread->setCompleted();
    return reinterpret_cast<unsigned>(aThread->result);
}

void* Thread::join() {
    DWORD rc = WaitForSingleObject(
      completionEvent,    // handle to event
      INFINITE);      // no timeout
    if (!(rc==WAIT_OBJECT_0))
        PrintError("WaitForSingleObject failed at ",__FILE__,__LINE__);
    return result;
}

void Thread::setCompleted() {
    DWORD rc = SetEvent(completionEvent);
    if (!rc) PrintError("SetEvent failed at ",__FILE__,__LINE__);
}
```

Listing 3.25 The Event object in the Win32 *Thread* class.

Recall from Chapter 1 that the *Thread* class constructor calls Win32 function *_beginthreadex()* to create a new Win32 thread. Several arguments are passed to *_beginthreadex()*, including:

- Thread::*startThread():* the startup method for the Win32 thread
- (LPVOID) this: a pointer to the *Thread* object that is being constructed. This pointer is forwarded to method *startThread().*

Method *startThread():*

- casts its void* pointer parameter to *Thread**
- then calls the *run()* method of the *Thread*.
- When *run()* returns, *startThread()* calls *setCompleted()* to set the thread's status to completed and to notify any threads waiting in *join()* that the thread has completed.

Methods *setCompleted()* and *join()* are implemented using an Event called *completionEvent* that is created in the *Thread* class constructor.

- A thread calling *T.join()* is blocked on *completionEvent* if T has not yet completed.
- The call to *setCompleted()* releases all threads that are blocked on *completionEvent* and leaves *completionEvent* in the signaled state.
- Since the *completionEvent* is never reset, threads that call *join()* after *setCompleted()* is called are not blocked.

If *completionEvent* were an auto-reset event, then a call to *setCompleted()* would release all the waiting threads and reset *completionEvent* to the non-signaled state.

This would cause a problem since any threads that then called T.*join()* would be blocked forever even though thread T had already completed.

3.7.5 Other Synchronization Functions

The *WaitForMultipleObjects*() function was described in Section 1.4 where it was used in the main thread to wait for child threads to finish.

- Threads are kernel objects and thus are either in the signaled or non-signaled state. When a thread is created and running, its state is non-signaled.
- When the thread terminates, it becomes signaled.

The *SignalObjectAndWait()* function allows the caller to atomically signal an object and wait on another object. When used with semaphores, it is equivalent to the *VP()* operation defined in Section 3.4.2.

```
DWORD SignalObjectAndWait(
  HANDLE      // handle to object for signal
  HANDLE      // handle to object for wait
  DWORD       // time-out interval
  BOOL        // alertable option: specifies whether the wait state can be aborted
);
```

Function *SignalObjectAndWait()* is available in Windows NT/2000 4.0 and higher. It is not supported in Windows 95/98.

3.7.6 Example: C++/Win32 Bounded Buffer

Listing 3.26 is a Win32 solution to the bounded buffer problem that is based on the Java version in Listing 3.17.

```
const int capacity = 3;
class Buffer {
private:
    int buffer[capacity];
    int count, in, out;
public:
    Buffer() : in(0), out(0), count(0) { }
    int size() { return count;}
    int withdraw () {
      int value = 0;
      value = buffer[out];          // out is shared by consumers
      out = (out + 1) % capacity;
      count--;
      return value;
    }
    void deposit (int value) {
      buffer[in] = value;           // in is shared by producers
      in = (in + 1) % capacity;
      count++;
    }
};

Buffer sharedBuffer;                // 3-slot buffer
mutexLock mutexD, mutexW;
countingSemaphore emptySlots(capacity);
countingSemaphore fullSlots(0);
int main() {
    std::auto_ptr<Producer> p1(new Producer);
    std::auto_ptr<Producer> p2(new Producer);
    std::auto_ptr<Consumer> c1(new Consumer);
    std::auto_ptr<Consumer> c2(new Consumer);
    p1->start();c1->start(); p2->start();c2->start();
    p1->join(); p2->join(); c1->join(); c2->join();
    return(0);
}
```

Listing 3.26 Win32 bounded buffer using *countingSemaphores* and *mutexLocks*.

```
class Producer : public Thread {
public:
    virtual void* run () {
      int i;
      std::cout << "producer running" << std::endl;
      for (i=0; i<2; i++) {
        emptySlots.P();
        mutexD.lock();
        sharedBuffer.deposit(i);
        std::cout << "Produced: " << i << std::endl;
        mutexD.unlock();
        fullSlots.V();
      }
      return 0;
    }
};
class Consumer : public Thread {
public:
    virtual void* run () {
      int result;
      std::cout << "consumer running" << std::endl;
      for (int i=0; i<2; i++) {
        fullSlots.P();
        mutexW.lock();
        result = sharedBuffer.withdraw();
        mutexW.unlock();
        std::cout << "Consumed: " << result << std::endl;
        emptySlots.V();
      }
      return 0;
    }
};
```

Listing 3.26 (cont.) Win32 bounded buffer using *countingSemaphores* and *mutexLocks*.