Multi-Reader / Multi-Writer Type-Safe Message Queue for Windows Inter-Thread Communications Using C++

Abstract: In most multi-threaded systems, threads coexist and collaborate with each other by sharing information and synchronizing their behavior. Managing such a collaboration can be tricky and error prone since it must occur in such a way that there is no chance of corruption or race conditions. To assist with this task, operating systems provide different kernel objects that can be employed efficiently and safely by client applications. Specifically, some operating systems offer generic messaging data structures with built-in synchronization capabilities; these are known as Message Queues. The Windows operating system does not offer a type-safe, easy to use, kernel message queue object for console applications, leaving application developers with the task of developing their own. This paper discusses type-safe message queues and the approach employed to implement them using the Win32 API.


Introduction

A Message Queue is a buffer-like object through which threads send and receive messages to communicate and synchronize with data (Li & Yao 97). They provide an easy and efficient method for inter-thread communication. They are so common that some operating systems, like VxWorks, use them as their primary mechanism for inter-thread communication. In VxWorks, message queues are provided as kernel-like objects that can be used by any thread (i.e. task) to send or receive messages. Their desirability comes from providing blocking operations, where a thread can block if either send or receive operation cannot be completed. Additionally, these communication data structures solve many of the problems introduced by applications containing multiple threads that read and write to the same message queue. Applications can take advantage of this functionality to easily implement efficient asynchronous messaging schemes that can be quickly employed to supply inter-thread communication capabilities. Unfortunately, the Windows platform does not provide a light, easy to use kernel object for console applications, with the same functionality as VxWorks’ message queues. Moreover, any attempt to provide message queuing capabilities to windows threads using the GetMessage () and PostThreadMessage () system calls are not type-safe, and makes it difficult to implement application with multiple readers. Luckily, the Win32 API does provide the essential building blocks to implement these types of message queues. With careful examination, developers can employ the kernel objects already provided by the Win32 API to encapsulate the behavior
of message queues, thus acquiring the same benefits that are present in other operating systems when programming multithreaded applications.

Devising the Blocking Behavior

The Win32 API is rich in functionality. It exists to provide user applications access to the operating system kernel, which is the part of the operating system that manages all the essential system services. The windows kernel is responsible for services such as memory, process, I/O, thread creation, thread scheduling, thread preemption, and so forth. All of these tasks are performed via kernel objects, which are objects that are solely created, managed, and destroyed by the kernel. Examples of windows kernel objects include: semaphores, pipes, and mutexes. These objects have one important feature in common, and that is their ability to allow threads to efficiently wait on them. When a thread waits on an unavailable kernel object, the kernel’s scheduler preempts the thread to allow other threads to execute, allowing the system to be more efficient. Since Windows does not provide a message queue kernel object, we’re left with the task of creating our own. We need to devise a generic type-safe queue that shows kernel-object-like behavior including blocking reads and blocking writes; and we must accomplish this using one of the kernel objects provided by Windows.

The approach employed in this paper to implement the message queue uses the Win32 semaphore kernel object. A semaphore is an object that one or more threads of execution can acquire or release for the purpose of synchronization or mutual exclusion (Li & Yao
Semaphores can be in two states: available, or unavailable. When a thread attempts to acquire a semaphore in the unavailable state, the operating system will take away its execution time until the semaphore becomes available. When this happens, the thread is said to be in a blocked state, which is a state that requires no CPU time. Semaphores can also be classified with two types: binary semaphores and counting semaphores. They differ only in the number of tokens each possesses. Binary semaphores can have only one token available to clients. On the other hand, counting semaphores can have many tokens available. What this means is that clients can acquire this type of semaphore several times before blocking. This behavior exposed by semaphores is exactly what we need for our message queue to provide blocking reads and writes. So, let’s start by encapsulating a Win32 semaphore with a Semaphore class.

Semaphore Constructor

The Semaphore constructor creates a Win32 semaphore and allows clients to configure it to a binary or a counting semaphore. The member attribute `m_hSemaphore` will be used to save the handle to the newly created object. Example 1 shows the implementation of the constructor.

```
1. Semaphore::Semaphore(int initialCount, int maxCount) :
2.                          m_maxCount(maxCount)
3. {                         
4.    m_hSemaphore = CreateSemaphore(NULL, initialCount, m_maxCount, 
5.                                NULL);
6. }
```

Example 1 - Semaphore Constructor

Clients wanting a binary semaphore can pass in the value of 1 for the parameter `maxCount`, and a 0 or 1 for the `initialCount` parameter to indicate the state of the semaphore. When
creating a binary semaphore, an initial count of 1 will indicate the available state, while an initial count of 0 will indicate the unavailable state. For counting semaphores, clients will pass in a value greater than 1 to indicate the number of tokens available to clients.

**Semaphore’s Take Operation**

The take operation is used to acquire a semaphore’s token. If one is available, the method will decrease the semaphore’s token count and return the value of true. If no tokens are available, the take operation will wait the amount of time specified by the `timeout` value before returning. If the `timeout` value expires and no token could be acquired, the operation will return a value of false. Example 2 shows the implementation of the take method.

```c
1. bool Semaphore::take (int timeout)  
2. {  
3.    bool retVal = false;  
4.    DWORD dwReturn = WaitForSingleObject(m_hSemaphore, timeout);  
5.    if( dwReturn == WAIT_OBJECT_0 )  
6.        retVal = true;  
7.    return retVal;  
8. }
```

Example 2 – Semaphore::take method

Internally, the take operation uses the `WaitForSingleObject` system call to request the acquisition of the `m_hSemaphore` object. Since this is a kernel object, the kernel will determine if it can be acquired. If so, the kernel will grant possession of the semaphore to the calling thread, and will allow it to continue execution; otherwise, it will block the thread. The way to determine that the semaphore was acquired is by checking the return status of the `WaitForSingleObject` call. The return value of `WAIT_OBJECT_0` will
indicate that the semaphore was acquired. It is this value that will determine if the take method call was successful or not

**Semaphore’s Release Operation**

The release operation does the opposite of the take operation. It is used to release a semaphore’s token. If successful, the release method will increase the semaphore’s token count and return the value of true; otherwise the value of false will be returned. Example 3 shows the implementation of the release method.

```cpp
1. bool Semaphore::release (void)
2. {
3.   bool retVal = false;
4.   if ( ReleaseSemaphore(m_hSemaphore, 1, NULL) != 0 )
5.     retVal = true;
6.   return retVal;
7. }
```

Example 3 – Semaphore::release method

Internally, the release operation uses the ReleaseSemaphore system call to increase the value of the semaphore, hence changing the state of the semaphore to the available state. The second parameter of the ReleaseSemaphore method indicates the amount used to increment the semaphore’s token count. In this case, the count will be incremented by one.

**Designing the Message Queue**

First, let’s deal with the type-safety issue. Microsoft’s approach to message queuing via the GetMessage () and PostThreadMessage () system calls do not expose good type-safe interfaces. They rely on type-unsafe parameters that can be interpreted as integers, packed
bit flags, pointers to structures containing additional data, and so on. This approach is error-prone, since it takes away the compiler’s ability to detect all misuses of variables that result in type errors (Sebesta 171). To improve this, we make use of the C++ template mechanism. The idea is to generalize the message queue class by using a name that can stand for any type. The compiler will only generate code for the template message queue class once the class is actually invoked by a statement in the program (Lafore 618). At this point, the compiler is fully aware of the types that the message queue can support, and will perform validity checks on the operations executed against it. Now that this is out of the way, we can move on to the message queue interfaces.

The message queue has two essential operations: the send operation, and the receive operation. These are the main interfaces exposed to clients for inter-thread communication. The general concept is to allow threads to communicate efficiently with each other. Specifically, we want to decouple a sending and receiving thread to keep them from having to send and receive messages simultaneously (Li & Yao 97). One approach could have one thread polling another thread’s message queue for a message. This is an expensive technique because it will drive the polling thread into a busy waiting state. Busy waiting states are undesirable because they waste valuable CPU cycles that some other thread may use productively (Silberschatz & Galvin 168). A better approach would be to inject blocking behavior to the send and receive operations, which we can accomplish by means of the semaphore class. But, first things first; the message queue is intended to execute in a multi-threaded environment, which means we must protect it from known problems like race conditions. Particularly, we need to pay attention to read and write operations. The
MsgQ class will employ the use of a binary semaphore to control client access to the queue. That is, in order for a client to add/retrieve a message, they need to successfully acquire the binary semaphore. This takes care of serializing access to the message queue. Now we move on to the blocking behavior. The general concept is to allow a client thread, or many client threads, to block whenever a send or receive cannot be serviced. For this particular feature, we will employ the use of a counting semaphore. A counting semaphore will be used to represent the number of messages in the queue at any given time. Once a client message is sent to the queue, one of the semaphore’s tokens will be released. On the receive side, every time one or more threads attempt to retrieve a message from the queue, one of the semaphore’s tokens will be requested, causing threads to block if none are available. This technique manages contention when multiple threads attempt to read from the same message queue and relies on the operating system to guarantee data integrity.

**MsgQ’s Constructor**

The MsgQ class uses an overloaded constructor to provide a parameter that specifies the size of the message queue. Example 4 shows the constructor’s implementation.

```cpp
1. template <class MimeType>
2. MsgQ<MimeType>::MsgQ(int maxMsg) : m_queueMaxSize(maxMsg),
3.                          m_countingSem(0, maxMsg),
4.                          NUM_OF_HANDLES(2)
5. {
6.    m_semHandles[0] = m_binarySem.getHandle();
7.    m_semHandles[1] = m_countingSem.getHandle();
8. }
```

Example 4 – MsgQ Constructor

On line 2, we initialize the m_queueMaxSize attribute to the size specified by the client. Line 3 initializes the member attribute “m_countingSem” to the unavailable state by
setting its initial token count to 0 and its maximum token count equal to the size of the message queue. Line 6 and 7 simply saves the handles of both semaphores used by the message queue. These handles will be used later on when attempting to retrieve messages from the queue.

**MsgQ Send Operation**

At a minimum, our send operation needs the following information: message to send, the timeout value, and the priority type. The timeout value specifies the number of milliseconds to wait in case the client cannot immediately acquire control of the queue for message insertion. The priority type specifies the type of priority of the message; that is, normal priority or high priority. For simplicity’s sake, we’ll only support two priorities. Normal priority will allow the queue to behave in a first-in-first-out (FIFO) manner. High priority messages will deviate from the FIFO scheme and store the messages in the front of the queue. This will cause the next receive operation to retrieve this message as the next message to process. Example 5 shows the implementation of the send operation.

```
1. template <class MsgType>
2. bool MsgQ<MsgType>::msgQSend (MsgType item,
3.                                   int waitTime = MSG_Q_WAIT_FOREVER,
4.                                   MsgQPriority priority = MSG_Q_NORMAL)
5. { 
6.   bool retVal = false;
7.   if( m_binarySem.take(waitTime) )
8.   { 
9.     if( m_msgQ.getListCount() < m_queueMaxSize )
10.    {
11.       retVal = (priority == MSG_Q_PRIORITY) ?
12.          m_msgQ.insertFront(item) : m_msgQ.insertBack(item);
13.     if( retVal )
14.       {   
15.         m_countingSem.release ();
16.       }
17.   }
```
On line 7 we attempt to take control of the queue. Once control is acquired, we can start making modifications. On line 11 we check the priority type of the message, placing a high priority item in front of the queue; otherwise, we send the item to the back of the queue. On line 15 we increase the value of the counting semaphore by calling its release method. This will cause any thread waiting on this semaphore to unblock and continue execution. Finally, on line 18 we give up control of the queue.

**MsgQ Receive Operation**

The most important (and risky) implementation of the message queue is the receive operation. This is because if done carelessly, it can lead to inefficient code or drive the system to a deadlock state. In order for clients to retrieve a message from the queue, two things must happen: 1) clients must acquire access to the queue, which they do by acquiring the binary semaphore, and 2) at least one message must be in the queue, which they know by successfully acquiring a token from the counting semaphore. This is where the implementation of the receive operation becomes risky. One approach could have the receive operation attempting to acquire the semaphores independently. This is inefficient and will most likely drive your system to a deadlock state. Anytime you make a system call, the operating system traps the call, suspends the calling thread, invokes the kernel to perform the requested service, places the result on the calling thread’s stack, and then resumes the thread’s execution (Phang & Garg 135). This causes two context switches; one
from user mode to kernel mode, and another from kernel mode to user mode. Two independent system calls in the receive method will double your execution time, since it will require four context switches for every receive operation. A more compelling reason to keep away from this approach is the possibility of introducing deadlocks to the system. The signature of the receive operation provides clients to specify a timeout value that could possibly indicate an infinite wait. If you successfully acquire the binary semaphore, but no counting semaphore tokens are available, you could find the code waiting for a counting semaphore infinitely while owning the binary semaphore. There are ways around this, but all things considered, it is not a good implementation to pursue. To achieve an efficient “deadlock-free” solution we use the WaitForMultipleObjects system call. The WaitForMultipleObjects method is very useful because it performs all of its operations atomically (Richter 291). The WaitForMultipleObjects operation, when configured properly, will successfully return only when it can acquire both semaphores. In the case that only one of the semaphores is available; its state will not be changed by the WaitForMultipleObjects method unless the other semaphore becomes available. It is then and only then, that the method will change both semaphores’ states. Example 6 shows the implementation of the receive operation.

```cpp
1. template <class MsgType>
2. bool MsgQ<MsgType>::msgQReceive(MsgType& item, int timeout)
3. { 
4.   bool retVal = false;
5.   if( ::WaitForMultipleObjects (NUM_OF_HANDLES, 
6.       m_semHandles, TRUE, timeout) == WAIT_OBJECT_0 )
7.     { 
8.       item = m_msgQ.getFirst()->nodeItem;
9.       retVal = m_msgQ.removeFront();
10.      m_binarySem.release();
11.     }
12. }
```
On lines 6 and 7 we attempt to take both semaphores. We do this by passing in an array containing the handles of both semaphores. The third parameter of TRUE configures the WaitForMultipleObjects method to return successfully only when both semaphores can be acquired. The timeout parameter, like in the other implementations, configures the amount of time to wait before the objects can be acquired. Lines 9 and 10 remove the item from the queue and make it available to the client. Finally, we release control of the queue on line 11.

**MsgQ Sample Usage**

Using the message queue is fairly easy. First, you create a message queue and specify the type and size of the queue. Once the queue is instantiated, you simply call either the msgQSend or msgQReceive methods. Example 7 shows a multi-reader scenario where two threads read from a global message queue.

```cpp
1. MsgQ<string> gMsgQ(1000);
2. DWORD WINAPI msgQthread(LPVOID lpParam)
3. {
4.    string msgQValue;
5.    while(1)
6.        {
7.            if( gMsgQ.msgQReceive (msgQValue, 5000) )
8.                {
9.                    // Messages received, handle it.
10.                }
11.            else
12.                // I’m still alive, no messages received in the last five seconds.
13.            }
14.        }
```
On line 17 and 18 we create two threads, both using the global message queue. Each thread makes use of the queue without paying attention to special synchronization techniques or data corruption. They simply block waiting on a message on the queue. Once a message is received, they can process it, otherwise they wake up after the configured amount of time to indicate to the main application that they are still alive; they just have nothing else to do.
References


