Topic: Windows Mechanisms for Synchronization and their Performance

1. Write programs to answer the following questions:
   a) What happens when a thread releases a mutex which it doesn’t hold?
   b) What happens when a thread requests a mutex more than once? If this is possible: what consequence does this have for releasing the mutex?
   c) What happens when a thread releases a semaphore which currently has its maximum value?
   d) What happens when a thread releases a semaphore which has never been requested by it?

2. Write a program to observe the “abandoned mutex” mechanism in Windows.

3. Write a piece of code to query the current value of a semaphore, without having changed this value in the end. There are (at least) two ways for doing so. Do not choose the “dangerous” one. Test your solution in all relevant cases for the value of the semaphore.

4. a) The purpose of this exercise is to measure the respective overhead caused by different mechanisms for achieving mutual exclusion.

   As code for the critical section to be protected, choose the operation of one of the interlocked functions. Then measure the time for the execution of this critical section, including the time used for the synchronization
   - without any synchronization (for the purpose of comparison),
   - when using the interlocked function,
   - when using a mutex,
   - when using a CriticalSection.
   Explain and comment your results. (Words like “faster” and “slower” are insufficient.)

   Perform your time measurement for a sufficiently large number of executions of the operation+synchronization (at least tens of thousands), so that the times for the loop and for the measurement only marginally affect the time to be measured. For the time measurement you may, for example, use `QueryPerformanceFrequency` and `QueryPerformanceCounter`.

   b) Which other mechanisms for achieving mutual exclusion does Windows provide? How do you expect those to behave time-wise, and why? (Of course you may also measure the time for those.)

5. Optional:
   Repeat exercises 1a) and 1b) for other synchronization mechanisms in Windows which can be used for achieving mutual exclusion.
6. Study the code example on the following pages (taken from the MSDN), and answer the subsequent questions.

Using Event Objects

Applications use event objects in a number of situations to notify a waiting thread of the occurrence of an event. For example, overlapped I/O operations on files, named pipes, and communications devices use an event object to signal their completion. For more information about the use of event objects in overlapped I/O operations, see Synchronization and Overlapped Input and Output.

In the following example, an application uses event objects to prevent several threads from reading from a shared memory buffer while a master thread is writing to that buffer. First, the master thread uses the CreateEvent function to create a manual-reset event object. The master thread sets the event object to nonsignaled when it is writing to the buffer and then resets the object to signaled when it has finished writing. Then it creates several reader threads and an auto-reset event object for each thread. Each reader thread sets its event object to signaled when it is not reading from the buffer.

```c
#define NUMTHREADS 4
HANDLE hGlobalWriteEvent;

void CreateEventsAndThreads(void)
{
    HANDLE hReadEvents[NUMTHREADS], hThread;
    DWORD i, IDThread;

    // Create a manual-reset event object. The master thread sets
    // this to nonsignaled when it writes to the shared buffer.

    hGlobalWriteEvent = CreateEvent(
        NULL,         // no security attributes
        TRUE,         // manual-reset event
        TRUE,         // initial state is signaled
        "WriteEvent"  // object name
    );

    if (hGlobalWriteEvent == NULL) {
        // error exit
    }

    // Create multiple threads and an auto-reset event object
    // for each thread. Each thread sets its event object to
    // signaled when it is not reading from the shared buffer.

    for(i = 1; i <= NUMTHREADS; i++)
    {
        // Create the auto-reset event.
        hReadEvents[i] = CreateEvent(
            NULL,         // no security attributes
            FALSE,        // auto-reset event
            TRUE,         // initial state is signaled
            NULL);        // object not named

        if (hReadEvents[i] == NULL)
        {
            // Error exit.
        }
    }
}
```
hThread = CreateThread(NULL, 0,
    (LPTHREAD_START_ROUTINE) ThreadFunction,
    &hReadEvents[i], // pass event handle
    0, &IDThread);
if (hThread == NULL)
{
    // Error exit.
}

Before the master thread writes to the shared buffer, it uses the ResetEvent function to set the state of hGlobalWriteEvent (an application-defined global variable) to nonsignaled. This blocks the reader threads from starting a read operation. The master then uses the WaitForMultipleObjects function to wait for all reader threads to finish any current read operations. When WaitForMultipleObjects returns, the master thread can safely write to the buffer. After it has finished, it sets hGlobalWriteEvent and all the reader-thread events to signaled, enabling the reader threads to resume their read operations.

VOID WriteToBuffer(VOID)
{
    DWORD dwWaitResult, i;

    // Reset hGlobalWriteEvent to nonsignaled, to block readers.
    if (! ResetEvent(hGlobalWriteEvent) )
    {
        // Error exit.
    }

    // Wait for all reading threads to finish reading.
    dwWaitResult = WaitForMultipleObjects(
        NUMTHREADS,   // number of handles in array
        hReadEvents,  // array of read-event handles
        TRUE,         // wait until all are signaled
        INFINITE);    // indefinite wait

    switch (dwWaitResult)
    {
        // All read-event objects were signaled.
        case WAIT_OBJECT_0:
            // Write to the shared buffer.
            break;

        // An error occurred.
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitProcess(0);
    }

    // Set hGlobalWriteEvent to signaled.
    if (! SetEvent(hGlobalWriteEvent) )
    {
        // Error exit.
    }

    // Set all read events to signaled.
    for(i = 1; i <= NUMTHREADS; i++)
    {
        if (! SetEvent(hReadEvents[i]) )
        {
            // Error exit.
        }
    }
Before starting a read operation, each reader thread uses `WaitForMultipleObjects` to wait for the application-defined global variable `hGlobalWriteEvent` and its own read event to be signaled. When `WaitForMultipleObjects` returns, the reader thread's auto-reset event has been reset to nonsignaled. This blocks the master thread from writing to the buffer until the reader thread uses the `SetEvent` function to set the event's state back to signaled.

```c
VOID ThreadFunction(LPVOID lpParam)
{
    DWORD dwWaitResult;
    HANDLE hEvents[2];

    hEvents[0] = *(HANDLE*)lpParam;  // thread's read event
    hEvents[1] = hGlobalWriteEvent;

    dwWaitResult = WaitForMultipleObjects(
        2,            // number of handles in array
        hEvents,      // array of event handles
        TRUE,         // wait till all are signaled
        INFINITE);    // indefinite wait

    switch (dwWaitResult)
    {
        // Both event objects were signaled.
        case WAIT_OBJECT_0:
            // Read from the shared buffer.
            break;

        // An error occurred.
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitThread(0);
            break;
    }

    // Set the read event to signaled.
    if (! SetEvent(hEvents[0]) )
    {
        // Error exit.
    }
}
```

Study the above code example demonstrating the use of event objects for implementing a Reader/Writer synchronization. In particular, think about why the types of the event objects were chosen the way they are. Would the program work if they were chosen otherwise?

Answer the following questions:

a) The readers' events are of type auto reset. How would you have to change the programs if this type of event did not exist? Hint: You don't have to use event objects (but do not use the SRW locks modeled exactly for the purpose of the above program).

b) The writer's event is of type manual reset. How would you have to change the programs if this type of event did not exist? (Again: no SRW locks allowed.) It is not possible to achieve the exact same behavior, but you can get very close. Give a detailed description and evaluation of the differences between the original program and your solution.

c) As soon as the writer "announces" its wish to write, no more read requests will get granted. How would you have to change the program to achieve a "read precedence", i.e. the writer will only be granted access when there happen to be no active readers?