Multithread in Win32 Note

Yinghua Zhang

September 16, 2013

Contents

Chapter 1

Why you need multithreading

Processes: Consist of memory and resources. A process by itself cannot run. it provides a place for memory and threads to live. In UNIX, a process and its main thread of execution are one and the same.

Memory: includes *code*, *data* and *stack*. code is the executable part of the program; data includes memory for global and static variables and allocated by *malloc()* and *new*; stack include call stack and local variables. A new stack is allocated for each thread created.

Thread: Very little information is needed to define a thread: a scratch pad in the processor (internal registers), and call stack in memory.

Comparing with process, thread is cheap, fast to start up and shut down, and have minimal impact on system resources. Threads also share ownership of most kernel objects such as file handles, window handles.

Context Switching: The operating system takes care of ensuring that every thread gets to run. It does this with assistance from the hardware and lots of bookkeeping. When a hardware timer decides that a thread has run long enough, an interrupt occurs. The processor takes a snapshot of the thread by copying all registers onto the stack. Then the operating system saves the current state of the thread by copying the thread's registers from the stack into a **CONTEXT** structure and saving the structure for later use. To switch to a different thread, the operating system points the processor at the memory of the threads's process, then restores the registers that were saved in the **CONTEXT** structure of a new thread. The operating system may make a context switch several hundred times per second.

threads in different process do not share any of their memory.

Race Condition: The order of execution between multiple threads becomes unpredictable. This unpredictability is the heart of what is called a race condition.

Atomic Operations: An operation that will complete without ever being interrupted is called an *atomic operation.*

Chapter 2

Getting a feel for thread

2.1 Creating a Thread

The arguments for *HANDLE CreateThread*(...):

- provides the address of the function where the new thread will start. It should return *DWOR* and take an *LPVOID* as an argument.
- parameter to the thread function.
- create a suspended thread, or a immediately starting thread.

Thread function does not necessarily complete before *main* continues on.

- Multithreaded Programs Are Unpredictable
- Order of execution is not guaranteed.
- Task switch can happen anywhere, anytime.
- Threads are sensitive to small changes.
- Threads do not always start immediately.

CreateThread(...) returns two values to refer to each new thread created:

- *HANDLE*: return value, local to the process, refers to a kernel object
- *Thread ID*: returned with argument *lpThreadId*, uniquely refers to the thread from any process. Thread ID could be used by *AttachThreadInput()* or *PostThreadMessage()* ot affect each other's message queues.

A handle is a pointer to something in the operating system's memory space that your program is not allowed to use directly. Kernel objects are referred to with *HANDLE*:

- may have multiple owners, use a reference count to count how many handles refer to it.
- The reference ount is incremented by *CreateThread()* and other functions that return a handle, and decremented when *CloseHandle* is called.

- If a process exits without calling *CloseHandle()* on all the kernel objects it has open, then the operating system will automatically drop the reference counts of those objects.
- When reference count drops to zero, the object is automatically destroyed.
- Kernel objects may include: *Processes*, *Threads*, *Files*, *Events*, *Semaphores*, *Mutexes*, *Pipes.*
- Kernel objects are used for coordinating the actions of multiple threads or processes so they can safely work in concert.

GDI objects may be referred y many kinds of Handles, such as *HBRUSH*, *HPEN*, *HDC* etc., have a single owner.

2.2 Terminate a Thread

You cannot rely on thread termination to clean up all the kernel objects created by that thread. Many objects, such as files, are owned by the process and will not be cleaned up until the process exits.

Call *CloseHandle()* with a handle to thread: disassociate the thread from the object. All *CloseHandle* does is decrement the reference count. If the count drops to zero, the object is destroyed. Call *CloseHandle()* and terminate the thread also, destroy a thread object. Thread handle is owned by process, so it is possible for the newly created thread to call *CloseHandle* instead of the original thread.

Use *GetExitCodeThread(HANDLE, DWORD exitCode)* to check if the thread is *STILL_ACTIVE.*

We can exit thread by returning from the thread function. We can also use *void ExitThread(DWORD dwExitCode)*, any code after this call will never execute.

Exiting the primary thread by returning from *main()* or calling *ExitProcess()* will cause all threads to be forcibly terminated and therefore the application itself will terminate. Always wait for your threads to exit before returning from *main().*

Anything that goes wrong in a API can be described by calling *GetLastError().* Proper error handling will create a more reliable application.

Chapter 3

Hurry and Wait

3.1 Wait

wait for something to happen:

- sleep
- busy loop

```
DWORD WaitForSingleObject(
    HANDLE hHandle, //handle to a kernel object
    DWORD dwMilliseconds //zero to INFINITE
);
```

WaitForSingleObject will success if

- the object becomes signaled, WAIT_OBJECT_0 is returned.
- time out, WAIT_TIMEOUT is returned
- a thread that owns a mutex exits without releasing the mutex, WAIT_ABANDONED is returned.

```
DWORD WaitForSMultipleObjects(
    DWORD nCount, //maximum number of objects
    CONST HANDLE * lpHandles, //handle to a kernel object
    BOOL bWaitAll, //whether wait for all
    DWORD dwMilliseconds //zero to INFINITE
);
```

WaitForMultipleObjects will success if

- *bWaitAll* = *true*, and all the objects signaled, WAIT_OBJECT_0 is returned.
- *bWaitAll* = *false*, return value minus WAIT_OBJECT_0 tells which object becomes signaled

- time out, WAIT_TIMEOUT is returned
- waiting on any mutexes return value could be from WAIT_ABANDONED_0 up to WAIT_ABANDONED_0+nCount-1.

This function wakes up if an object is signaled or if a message arrives in the queue.

```
DWORD WaitForSMultipleObjects(
    DWORD nCount, //maximum number of objects
    LPHANDLEpHandles, //handle to a kernel object
    BOOL fWaitAll, //whether wait for all
    DWORD dwMilliseconds //zero to INFINITE
    DWORDdwWakeMask //types of user input to look for
);
```

To indicate that a message has arrived in the queue,
the return value is WAIT_OBJECT_0+nCount. If another thread updates the list of objects that you are waiting on, you need a way to force *MsgWaitForMultipleObjects()* to return and be restarted to include the new handle.

3.2 Signal

Kernel objects can state something interesting to the running thread.

- Semaphore and Mutex can report to the *next* waiting thread
- Files can tell when an I/O operation is complete
- Thread can tell when they have exited.

WaitforsingleObject() wait up when an object becomes signaled.
meaning of signaled kernel objects

- Thread*: signaled when terminates
- Process*: Singaled when terminates
- Event*: The state of an event object is directly controlled by the application using *SetEvent()*, *PulseEvent()*, and *ResetEvent()*
- Mutex*: singaled when it is not owned by any thread. When a wait on a mutex returned, it is owned by the thread of the wait function, and the mutex becomes unsignaled.
- Semaphore*: Signaled when its count is greater than zero and nonsignaled when the count is zero.

There is a limit to the number of objects you can wait for at one time with $WaitForMultipleObjects()$.

Chapter 4

Synchronization

A synchronization mechanism is the equivalent of a traffic light for threads.

4.1 Critical Section

There could be multiple places in your application that access the shared resource. Together, all of these pieces of code make up a single critical section.

```
void InitializeCriticalSection(
    LPCRITICAL_SECTION lpcriticalSection,
    ...//pointer to variable of type CRITICAL_SECTION
);
```

Clean it up by calling

```
void DeleteCriticalSection(
    LPCRITICAL_SECTION lpcriticalSection,
    ...//pointer to variable of type CRITICAL_SECTION
    that is no longer needed
);
```

Once the critical section has been initialized, each thread that wants to enter it must ask permission by calling:

```
void EnterCriticalSection(
    LPCRITICAL_SECTION lpcriticalSection,
    ...//pointer to variable of type CRITICAL_SECTION
    should be locked
);
```

When the thread is ready to leave the critical section of code, it must call

```
void LeaveCriticalSection(
    LPCRITICAL_SECTION lpcriticalSection,
    ...//pointer to variable of type CRITICAL_SECTION
    should be unlocked
);
```

By putting the code into the critical section, we have forced the operation to be atomic. Once a thread enters a critical section, it is allowed to reenter the critical section over and over again, so that functions with critical section code could call each other.

Never call *sleep()* or wait function in a critical section.

There is no way to tell if the thread currently inside a critical section is still alive: reason for us to use mutex

deadlock: each has what the other needs. Any time a section of code requires two or more resources, there is a potential for deadlock.

```
void SwapLists(List * A, List * B){
    List * tmp;
    EnterCriticalSection(A- > Critical_sec);
    EnterCriticalSection(B- > Critical_sec);
    tmp = A- > head;
    A- > head = B- > head;
    b- > head = tmp;
    LeaveCriticalSection(A- > Critical_sec);
    LeaveCriticalSection(B- > Critical_sec);
}
Thread1 : SwapLists(home_list, work_list);
Thread2 : SwapLists(work_list, howm_list);
```

"Dining philosophers"

We can not wait for multiple critical sections. Critical section is not kernel objects, it can only be used within the same process.

4.2 Mutex

mutex: MUTual EXclusion.

by a mutex name, any thread anywhere in the system can access the mutex (threads of different process).

Create a mutex by calling:

```
HANDLE CreateMutex(
    LPSECURTY_ATTRIBUTES lpMutexAttributes,
    ...//Use NULL to get default attributes
    BOOL bInitialOwner
    ...//Set TRUE creating thread own the mutex
    LPCTSTR lpName//Text name
);
```

return a handle if succeeds, otherwise it returns NULL. Set the *bInitialOwner* help to prevent race condition, if the the creating thread need the mutex immediately: if set this FALSE, an extra waiting function is needed.

When you are finished with a mutex, close your handle to it by calling: *CloseHandle(hMutex);*

If a mutex has already been created with a particular name, then any other thread or process can open it using that name. If you call *CreateMutex* with the name of an existing mutex, then it will pass back a handle to the existing mutex. *GetLastError* returns *ERROR_ALREADY_EXISTS* after calling *CreateMutex*. *OpenMutex()* can also used to do this.

To acquiring ownership of the mutex, use one of the wait functions. If a mutex waits on a nonsignaled mutex then the thread will ***block***. A wait call will cause a thread to block if the timeout is greater than zero.

the thread releases ownership of the mutex by calling

$$
\begin{aligned}
&BOOL\ ReleaseMutex(\\
&\quad HANDLE\ hMutex\\
&);
\end{aligned}
$$

Ownership: the thread did a Wait...() on the mutex and has not yet called *ReleaseMutex()*.

If a thread that owns a mutex exits without calling *ReleaseMutex()*, the mutex is not destroyed.
Instead, it is marked as *unowned* and *nonsignaled*. The next waiting thread will be notified with the special flag WAIT_ABANDONED_0. *WatiForMultipleObjects()* will return a value between
$WAIT\_ABANDONED\_0$ and $WAIT\_ABANDONED\_n-1$, n will indicate which mutex in the array was abandoned.

It is handy to know that a mutex has been abandoned, but figuring out what to do is hard. If a thread crashed, the whole data structure may be damaged.

Use the following code to solve the dining-philosopher problem:

$$WaitForMultipleObjects(2,\ myChopsticks, TRUE, INFINITE);$$

4.3 Semaphores

A ***Semaphore*** maintains a count, assuring that increments and decrements are handled in an atomic fashion.

They are key ingredient in solving various producer/consumer problems.

Unlike mutex, there is no specific state called "wait abandoned" that can be detected by other thread.

To create a semaphore, call

$$
\begin{aligned}
&HANDLE\ CreateSemaphore(\\
&\quad LPSECURTY\_ATTRIBUTES\ lpAttributes,\\
&\quad LONG\ lInitialCount,\\
&\quad LONG\ lMaximumCount, //Maximum\ value\ of\ semaphore\\
&\quad ...//Maximum\ number\ of\ threads\ can\ lock\ the\ semaphore\\
&\quad LPCTSTR\ lpName//Text\ name\\
&);
\end{aligned}
$$

If the current count of a semaphore is non-zero, wait function will return immediately. This is similar to the mutex. But you do not receive ownership of the semaphore when the lock succeeds. Because there is no concept of ownership a thread that repeatedly calls one of the wait functions will create a new lock for each successive call. This is in contrast to the mutex, where the thread that owns the mutex will not block no matter how many times it calls a wait function. On the other hand, if the count of a semaphore reaches zero, any thread that tries to call wait will wait until one of the locks is released.

Call *ReleaseSemaphore()* to release a lock: increments the semaphore's value.

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,
    LPLONG lpPreviousCount,
    ...//the previous value of the semaphore
);
```

The third argument is useful in the multi-thread environment. By setting the initial count to zero, the thread that creates the semaphore can take care of any necessary initialization. When initialization is complete, a single call to *ReleaseSemaphore()* can increment the ount up to its maximum.

4.4 Event

Kernel Object under fully control

To create an event object, use:

```
HANDLE CreateEvent(
    LPSECURTY_ATTRIBUTES lpEventAttributes,
    BOOL bManualReset,
    BOOL bInitialState,
    LPCTSTR lpName//Text name
);
```

bManualReset determines whether the event resets itself automatically to nonsignaled after waking up a single thread. *bInitialState* set TRUE for initially signaled, FALSE for initially nonsignaled.

When Event is signaled, and there is no threads waiting, the Event will be lost. With *auto-reset*, a wait function must be called, after *SetEvent()* or *PulseEvent()*.

- SetEvent(): set the event to signaled, wake everything that is currently waiting
- ResetEvent(): set the event to nonsignaled
- *SendMessage()* will not return until the primary thread has proceed the message.
- *PostMessage()* returns immediately.

4.5 interlocked variables

```
LONG InterlockedIncrement(
    LPLONG lpTarget
);
LONG InterlockedDecrement(
    LPLONG lpTarget
);
```

lpTarget should be a address of 32-bit value. Return the resulting incremented or decremented value.

```
LONG InterlockedIncrement(
    LPLONG lpTarget
    LONG lValue
);
```

Set a new value and return the old value.

Chapter 5

Keeping your thread on a leash

Aborting a thread with *TerminateThread()*.

```
BOOL TerminateThread(
    HANDLE hThread
    DWORD dwExitCode//should be reported by hThread
);
```

hThread is given no chance to clean itself.

hThread will become signaled with the exit code specified. Any DLLs that are attached to the thread will not be notified that the thread is detaching.

If it is called in a critical section, the critical section will stay permanently locked.

Use exceptions in Windows to clean up a thread.

Use *SetEvent* to terminate another thread who is waiting for it.

5.1 Priority Class

Priority Classes apply to processes, not threads, can be adjusted and examined with *SetPriorityClass()* and *GetPriorityClass()*.

- HIGH_PRIORITY_CLASS: 13
- IDEL_PRIORITY_CLASS: 4
- NORMAL_PRIORITY_CLASS: 7 or 8
- REALTIME_PRIORITY_CLASS: 24

Most applications run with *NORMAL_PRIORITY_CLASS*.

5.2 Priority Level

Priority Level allows you to adjust the relative importance of threads within a process without having to know the importance of the process as a whole.

- THREAD_PRIORITY_HIGHEST: +2
- THREAD_PRIORITY_ABOVE_NORMAL: +1
- THREAD_PRIORITY_NORMAL: 0
- THREAD_PRIORITY_BELOW_NORMAL: -1
- THREAD_PRIORITY_LOWEST: -2
- THREAD_PRIORITY_IDLE: set to 1
- THREAD_PRIORITY_TIME_CRITICAL: set to 15

Priority levels are modified with *SetThreadPriority()*

```
BOOL SetThreadPriority(
    HANDLE hThread
    int nPriority
);
```

If succeeds, return one of the thread priorities, if fails, return FALSE.

A thread's current priority level can be examined using

```
int GetThreadPriority(
    HANDLE hThread
);
```

5.3 Dynamic Boost

Dynamic Boost is an adjustment to current priority that the system applies to a thread on an as-needed basis to enhance the application's usability.

Foreground application get more CPU time than the background applications. Things may cause a priority boost include: keyboard input, mouse messages, timer messages, disk input that has completed, or a a wait condition is satisfied.

Setting priority unwisely could lead to ***Starvation***.

set the fifth argument of CreateThread() as CREATE_SUSPENDED to give the thread a chance to do initial setup.

Once the thread is setup, start the thread by calling

```
BOOL ResumeThread(
    HANDLE hThread
);
```

There is a companion function to *Resumethread()*

```
BOOL SuspendThread(
    HANDLE hThread
);
```

Suspending a thread who is currently in a critical section or locking a mutex can leads to deadlock

Chapter 6

Overlapped I/O

Nonblocking or asynchronous I/O.

Support for overlapped I/O is limited in Windows and can only be used for named pipes, mailslots, serial I/O, and sockets returned by *socket()* or *accept()*.

6.1 Windows File Calls

CreateFile() can be used to open a wide variety of resources, including but not limited to:

- Files
- Serial and parallel ports
- Named pipes
- console

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwSharedMode,
    LPSECURTY_ATTRIBUTES lpMutexAttributes,
    DWORD dwCreationDistribution,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile,
);
```

Set the sixth argument *dwFlagsAndAttributes*
as FILE_FLAG_OVERLAPPED.

Overlapped I/O can read and/or write from multiple parts of the file at the same time. The tricky part is that operations all use the same file handle at the same time. Thus there is no concept of *current file position* when you are using overlapped I/O, Every read or write call must include the file position.

There is no way to use overlapped I/O with the calls in *stdio.h* in the C run-time library. Therefore there is no easy way of doing overlapped text-based I/O.

The basic form of overlapped I/O is performed with

```
HANDLE ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped,
);
```

and

```
HANDLE WriteFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped,
);
```

OVERLAPPED structure performs two important functions.

- acts as a key that uniquely identifies each overlapped operation currently in progress
- provides a shared area between you and the system where parameters can be passed in each direction.

```
typedef struct_OVERLAPPED{
    DWORD Internal;
    DWORD InternalHigh;
    DWORD offset;
    DWORD offsetHigh;
    HANDLE hEvent;
};
```

Offset: Bytes offset within the file to begin reading or writing, measured from the beginning of the file, is ignored for devices that do not support a file position. *OffsetHigh*: High 32 its of the 64-bit file offset to begin rading and writing. Because the lifetime of the OVERLAPPED structure extends beyond the call to *ReadFile()* or *WriteFile()*, it is usually safest to allocate the OVERLAPPED structure on the heap.

If you need to wait for the result as part of a call to *WaitForMultipleObjects()*, use the handle to the file in the handle array. The file handle, as a kernel object, will be signaled when the operation is complete. When you are ready to process the request, call *GetOverlappedResult()* to find out what happened. This call exists because it is not possible to know for sure if the operation succeeded until it actually happens.

$$
\begin{aligned}
&BOOL\ GetOverlappedResult(\\
&\quad HANDLE\ hFile, \\
&\quad LPOVERLAPPED\ lpOverlapped, \\
&\quad LPDWORD\ lpNumberOfBytesTransferred, \\
&\quad BOOL\ bWait, \\
&);
\end{aligned}
$$

bWait indicates whether to wait for operation to complete. Waits if TRUE.

To initialize the overlapped structure

$$
\begin{aligned}
&memset(overlap,\ 0,\ sizeof(overlap)); \\
&overlap.Offset = 1500;
\end{aligned}
$$

To use the overlapped structure

$$
\begin{aligned}
&WaitForSingleObject(hFile,\ INFINITE); \\
&GetOverlappedResult(hFile,\ \&overlap,\ \&numread,\ FALSE);
\end{aligned}
$$

If call *GetLastError()* and it returns ERROR_IO_PENDING, it means that the overlapped I/O is queued and waiting to happen.

6.2 Signaled Event Objects

When wait for a file handle, you can not tell which operation is completed. When the event handle in an OVERLAPPED structure is set to an event object, the kernel will automatically signal the event object when the overlapped I/O operation is completed.

It is very important that the event object you use be created as manual reset. Using auto-reset leads to race condition because the kernel could signal the object *before* you have a chance to wait on it. In this case the event would be lost and your wait would never return.

6.3 Asynchronous Procedure Calls

APC: a callback routine that the system should call when an overlapped I/O operation finishes.

Your thread must be in an alertable state for a APC to be served. To be alertable, call one of the five functions with the alertable flag set to TRUE:

- SleepEx();
- WaitForSingleObjectEx();
- WaitForMultipleObjectsEx();
- SignalObjectAndWait();

use a class member function as APC: storing a pointer to the object in the user-defined data, then call a member function through this pointer. Given the existence of APCs, there sis little use for waiting on either file handle or event objects for ReadFile() and WriteFile() operations For a series of requests less then some amount, using overlapped I/O took more time than making blocking calls Use a pool of threads for large amount of small transfers.

6.4 I/O Completion Ports

Completion ports were desinged to more easily build scalable servers by allowing a pool of threads to service a pool of events.

- A thread implicitly becomes part of the pool by waiting one the I/O completion port.
- Every time a new file is opened for overlapped I/O, you associate its file handle with the I/O completion port. Once this association is established, any file operation that comppletes successfully will cause an I/O completion packet to sent to the port.
- In response to the packet, the completion port releases one of the waiting threads in the pool. The released thread is given enough information to be able to identify the context of the completed overlapped I/O operation. The thread becomes active thread and not a waiting thread. When work done, it becomes waiting thread again.

Create a I/O completion port:

```
HANDLE CreateIOCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    ...//Use NULL to Create a New port
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

FileHandle: can be set as INVALID_HANDLE_VALUE initially *CompletionKey*: User-defined value that will be passed to the thread that services a request. This key is associated with FileHandle. A file handle attached to an I/O completion port can no longer be used with ReadFileEx() or WriteFileEx(). *NumberOfConcurrentThreads* can be set to zero so that as many threads will run as there are processors in the system.

Create thread pool:

$$\begin{aligned}&Threadscurrentlyrunning+\\&Threadsblocked+\\&Threadswaitingontheport\\&=Threadsinpool\end{aligned}$$

After each thread initializes itself it should call *GetQueuedCompletionStatus()*. It acts like a combination of *WaitForSingleObject()* and *GetOverlappedResult()*.

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytesTransferred,
    LPDWORD lpCompletionKey,
    LPOVERLAPPED * lpOverlapped,
    DWORD dwMilliseconds
);
```

Chapter 7

Data Consistency

The optimizer in the compiler tries to keep frequently used data in the processors' internal registers. Data can be read from the registers much faster than from RAM. However, if another thread changes the original value in RAM, then the copy of the variable in the register will be out of date. The keyword ***volatile*** tells the compiler not to keep temporary copies of a variable. Using volatile does not negate the need for critical sections or mutexes.

```
void WaitForKey(volatile char * pch){
    while(*pch == 0)
    ;
);
```

List of Figures

List of Tables