Window Threads

Windows Thread Management
Example Code win_thread.cpp
win_condvar.cpp

Thread Management

- Creating a Thread
- The Thread Function
- Thread Termination
- Thread Exit Codes
- Thread Identities
- Suspending and Resuming Threads

Creating a Thread

- Specify the thread's start address within the process' code
- Specify the stack size, and the stack consumes space within the process' address space
  - The stack cannot be expanded

Creating a Thread

- Specify a pointer to an argument for the thread
  - Can be nearly anything
  - Interpreted by the thread itself
- **`CreateThread`** returns a thread's ID value and its handle
  - A **`NULL`** handle value indicates failure

Creating a Thread

```
HANDLE CreateThread (
     LPSECURITY_ATTRIBUTES lpsa,
     DWORD cbStack,
     LPTHREAD_START_ROUTINE lpStartAddr,
     LPVOID lpvThreadParm,
     DWORD dwCreate,
     LPDWORD lpIDThread )
```

Creating a Thread

- Parameters

  `lpsa`

  - Security attributes structure (use **`NULL`**)

  `cbStack`

  - Byte size for the new thread's stack
  - Use 0 to default to the primary thread's stack size (1 MB)

Creating a Thread

`lpStartAddr`

- Points to the function (within the calling process) to be executed
- Accepts a single pointer argument and returns a 32-bit **`DWORD`** exit code
- The thread can interpret the argument as a **`DWORD`** or a pointer

`lpThreadParm`

- The pointer passed as the thread argument

Creating a Thread

- Threads are terminated by **`ExitProcess`**
  - The process and all its threads terminate
  - The exit code returned by the thread start function same as the process exit code
  - Or a thread can simply return with its exit code

The Thread Function

```
DWORD WINAPI MyThreadFunc (
          PVOID pThParam )
{    . . .
     ExitThread (ExitCode);  /* OR */
     return ExitCode;
}
```

Thread Terminaton

- Threads are terminated by **`ExitProcess`**
  - The process and all its threads terminate
  - The exit code returned by the thread start function same as the process exit code
  - Or a thread can simply return with its exit code

Thread Terminaton

- **`ExitThread`** is the preferred technique
- The thread's stack is deallocated on termination

```
VOID ExitThread (DWORD (dwExitCode)
```

- When the last thread in a process terminates, so does the process itself

Thread Terminaton

- You can terminate a different thread with **`TerminateThread`**
  - Dangerous: The thread's stack and other resources will not be deallocated
  - Better to let the thread terminate itself
- A thread will remain in the system until the last handle to it is closed (using **`CloseHandle`**)
  - Then the thread will be deleted
- Any other thread can retrieve the exit code

Thread Exit Codes

```
BOOL GetExitCodeThread (
      HANDLE hThread,
      LPDWORD lpdwExitCode )
```

lpdwExitCode

- Contains the thread's exit code
- It could be **STILL_ACTIVE**

Thread Identities

- A thread has a permanent "**`ThreadId`**"
- A thread is usually accessed by **`HANDLE`**
- An ID can be converted to a **`HANDLE`**

Thread Identities

```
HANDLE GetCurrentThread (VOID);
DWORD GetCurrentThreadId (VOID);
HANDLE OpenThread (
      DWORD dwDesiredAccess,
      BOOL InheritableHandle,
      DWORD ThreadId );
```

Suspend and Resume Threads

- Every thread has a suspend count
  - A thread can execute only if this count is zero
- A thread can be created in the suspended state
- One thread can increment or decrement the suspend count of another:

Suspend and Resume Threads

```
DWORD ResumeThread (HANDLE hThread)
DWORD SuspendThread (HANDLE hThread)
```

- Both functions return previous suspend count
- **0xFFFFFFFF** indicates failure
- Useful in preventing “race conditions”
  - Do not allow threads to start until initialization is complete
- Unsafe for general synchronization

Waiting for Thread Termination

- Wait for a thread to terminate using general purpose wait functions
- **`WaitForSingleObject`** or **`WaitForMultipleObjects`**
  - Using thread handles
- The wait functions wait for the thread handle to become signaled
  - Thread handle is signaled when thread terminates

Waiting for Thread Termination

- **`ExitThread`** and **`TerminateThread`** set the object to the signaled state
  - Releasing all other threads waiting on the object
- **`ExitProcess`** sets the process' state and all its threads' states to signaled

Waiting for Thread Termination

```
DWORD WaitForSingleObject (
      HANDLE hObject,
      DWORD dwTimeOut )

DWORD WaitForMultipleObjects (
      DWORD cObjects,
      LPHANDLE lphObjects,
      BOOL fWaitAll,
      DWORD dwTimeOut )
```

- Return: The cause of the wait completion

Wait Options

- Specify either a single handle **`hObject`**
- Or an array of **`cObjects`** referenced by **`lphObjects`**
- **`cObjects`** should not exceed **`MAXIMUM_WAIT_OBJECTS - 64`**

Wait Options

- **`dwTimeOut`** is in milliseconds
  - 0 means the function returns immediately after testing the state of the specified objects
  - Use **`INFINITE`** for no timeout
  - Wait forever for a thread to terminate
- **`GetExitCodeThread`**
  - Returns the thread exit code

Wait Return Values

- **`fWaitAll`**
  - If **`TRUE`**, wait for all threads to terminate

Possible return values are:

- **`WAIT_OBJECT_0`**
  - The thread terminated (if calling **`WaitForMultipleObjects`**; **`fWaitAll`** set)

Wait Return Values

- **`WAIT_OBJECT_0 + n`**

  where **`0 <= n < cObjects`**

  - Subtract **`WAIT_OBJECT_0`** from the return value to determine which thread terminated when calling **`WaitForMultipleObjects`** with **`fWaitAll`** set

- **`WAIT_TIMEOUT`**
  - Timeout period elapsed

Wait Return Values

- **`WAIT_ABANDONED`**
  - Not possible with thread handles
- **`WAIT_FAILED`**
  - Call **`GetLastError`** for thread-specific error code

Mutex

A thread uses the **CreateMutex** or **CreateMutexEx** function to create a mutex object. The creating thread can request immediate ownership of the mutex object and can also specify a name for the mutex object. It can also create an unnamed mutex. For additional information about names for mutex, event, semaphore, and timer objects, see InterprocessSynchronization.

Mutex

Threads in other processes can open a handle to an existing named mutex object by specifying its name in a call to the **OpenMutex** function. To pass a handle to an unnamed mutex to another process, use the **DuplicateHandle** function or parent-child handle inheritance.

Mutex

Any thread with a handle to a mutex object can use one of the wait functions to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object using the **ReleaseMutex** function. The return value of the wait function indicates whether the function returned for some reason other than the state of the mutex being set to signaled.

Mutex

If more than one thread is waiting on a mutex, a waiting thread is selected. Do not assume a first-in, first-out (FIFO) order. External events such as kernel-mode APCs can change the wait order. After a thread obtains ownership of a mutex, it can specify the same mutex in repeated calls to the wait-functions without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. To release its ownership under such circumstances, the thread must call **ReleaseMutex** once for each time that the mutex satisfied the conditions of a wait function.

Mutex

If more than one thread is waiting on a mutex, a waiting thread is selectedIf a thread terminates without releasing its ownership of a mutex object, the mutex object is considered to be abandoned. A waiting thread can acquire ownership of an abandoned mutex object, but the wait function will return WAIT_ABANDONED to indicate that the mutex object is abandoned. An abandoned mutex object indicates that an error has occurred and that any shared resource being protected by the mutex object is in an undefined state. If the thread proceeds as though the mutex object had not been abandoned, it is no longer considered abandoned after the thread releases its ownership. This restores normal behavior if a handle to the mutex object is subsequently specified in a wait function.

Note that critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process.

. Do not assume a first-in, first-out (FIFO) order. External events such as kernel-mode APCs can change the wait order.

After a thread obtains ownership of a mutex, it can specify the same mutex in repeated calls to the wait-functions without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. To release its ownership under such circumstances, the thread must call **ReleaseMutex** once for each time that the mutex satisfied the conditions of a wait function.