

Multithreading Application in Win 32 (Ch2 : Getting a feel for Threads)

Seong Jong Choi
chois@mmlab.net

[Multimedia Lab.](#)

Dept. of Electrical and Computer Eng.
University of Seoul
Seoul, Korea

Contents

1. Creating a Thread
2. Consequences of Using Multiple Threads
3. Kernel Objects
4. The Thread Exit Code
5. Exiting a Thread
6. Error Handling
7. Background Printing
8. Summary or conclusion

Main Objectives

- Win32 API calls for managing a thread
 - Creating
 - Monitoring
 - Exiting a thread
 - Closing a thread
- New Concepts:
 - Kernel objects and Handle
 - Reference Count
 - Primary thread

Overview

- how to creating a thread : `CreateThread()`
- function call vs. thread start up
- unpredictable
 - The results are depend on the speed of processor
 - what the threads are doing
 - how busy processor is
 - numerous other factors
- `CloseHandle()`
- `GetExitCodeThread()`
- `ExitThread()`

Creating a Thread

- Thread 를 생성하기 위하여 `CreateThread()` 를 이용한다.

- **prototype**

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // pointer to security
    attributes
    DWORD dwStackSize, // initial thread stack size (default size 1 MB : 0)
    LPTHREAD_START_ROUTINE lpStartAddress, // pointer to thread
    function (start address)
    LPVOID lpParameter, // argument for new thread
    DWORD dwCreationFlags, // creation flags
    LPDWORD lpThreadId // pointer to receive thread ID of the new thread
    (returned value)
);
```

Return Value

Success: a handle to the thread

Failure: FALSE

Creating a Thread

- Function Call vs. thread startup
 - listing 2-1
- Function Call
 - Control is transferred to the sub function, which executes and then returns control to the caller
- 순차적인 진행
- 1 개의 thread로 실행

Creating a Thread

Thread startup

- `CreateThread()` starts the new thread of execution, which then calls `ThreadFunc ()`
- once `ThreadFunc()` starts up, it is completely **separated from the caller**
- the ordering is not predictable
- main thread + new threads 실행

Creating a Thread

- look at a Program : Numbers.c
 - 5개의 **thread** 를 생성하고 실행한다.
 - **Function call** 과는 다른 점을 직관적으로 보여 준다.

Consequence of Using Multiple Thread

- Multithreaded Programs are **unpredictable**
 - in figure 2-1, you can see that different results
 - Every time NUMBERS is run, you get different output
 - the results are dependent on the speed of the processor,
 - what the threads are doing, how busy the processor is,
 - and numerous other factors.
 - the key goal is teach you how to get predictable result

Consequence of Using Multiple Thread

- Order of Execution is not guaranteed
- Context Switches can happen anywhere, anytime
 - context switch happened while the thread was in the middle of displaying its results
 - 333344444444 (figure 2-1 coloume2)
- Threads are sensitive to small changes
- Threads do not always start immediately

Kernel Object and Handle

- Kernel Object:
 - A *kernel object* is a data structure in kernel space that represents a system resource, such as a file, thread, or graphic image
- Examples of Kernel Objects
 - Processes
 - Threads
 - Files
 - Events
 - Semaphores...
- Object Handle:
 - An application cannot directly access object data or the system resource that an object represents. Instead, an application must obtain an *object handle*, which it can use to examine or modify the system resource

Kernel Objects

- `CreateThread()` returns two values.
 1. Thread HANDLE
 - `CreateThread()`에서 return된 value
 - local in the process (다른 process는 사용하지 못한다)
 - refers to a kernel object
 2. threadID
 - returned with `lpThreadId`(parameter 로 return)
 - global value (유일한 값을 가지기 때문에 시스템 내의 어떤 process라도 사용이 가능하다)

CloseHandle()

- Why use CloseHandle()?
 - Need to release kernel object when you are finished using them.
- Use CloseHandle() API function
- prototype

```
BOOL CloseHandle(  
HANDLE hObject // handle to object to close  
);
```
- Return value
 - Success : TRUE
 - Fail : FALSE

CloseHandle()

- Why call CloseHandle()?
 - Process가 thread handle을 closing 하지 않고 새로운 thread 를 자주 생성하게 된다면 수백, 수천개의 kernel objects 가 남게 되고 이 불필요한 모든 Objects를 OS 가 Manage 하여야 한다.
 - Such resource leaks can have a significant negative impact on system performance.

The difference Between a Thread Object and a Thread

- The thread object represents a thread.
 - Reference count on a thread is two, when the thread is created.
 - When you call `CloseHandle()` the count drops (-1) and when the thread terminates the count drops
 - When the reference count is zero, the thread object is destroyed from the kernel memory.

Exit Code

- Why use `ExitCode()`?
- `main` 함수에 의해 생성되어진 `thread` 들이 종료하기 전에 `main()` 이 먼저 종료를 하게 된다면 어떤 결과를 가져올까?
- 앞의 프로그램들에서 사용되어진 `sleep()`은 하나의 `trick` 이다
- Make sure that the threads have finished and examine there exit values before exiting the program
- This information can be obtained using the thread handle returned by `CreateThread()` and calling the function `GetExitCodeThread()`

Exit Code

- **Prototype**

```
BOOL GetExitCodeThread(  
    HANDLE hThread, // handle to the thread  
    LPDWORD lpExitCode // address to receive exit status  
);
```

Return value

Success : TRUE

Fail : FALSE

- On failure, call **GetLastError()** to find out why.
- If the thread has exited, then the thread's exit value will be written at *lpExitCode*
- If the thread is still running, then the values **STILL_ACTIVE** will be written at *lpExitCode*

Exit Code

- Look at a Program : `EXITCODE.c`
 - 2개의 `thread` 를 생성
 - `thread`의 현재 상태를 표시
- `STILL_ALIVE == 259`

Exiting a Thread

- To exit a thread without returning all the way up to the thread function.
- Use the API function `ExitThread()`
- Its prototype is

```
VOID ExitThread(  
    DWORD dwExitCode // exit code for this thread );
```

Return value
None

- Any code that comes after this call will never execute.

Exiting a Thread

- Look at a Program : EXITTHREAD.c
- 생성된 `thread` 가 실행되던 중에 `ExitThread()` 가 실행된다.
- `ExitCode` 값을 `Check` , `ExitThread()` 가 수행이 된 후에는 뒤에 나오는 화면 출력 함수가 실행되지 않는다.

Error Handling

- Experience has shown that it is easy to make a mistake when calling the various thread function, and proper error handling will prevent frustration and create a more reliable application
- Using a macro called `MTVERIFY()`
- If win32 function fails, `MTVERIFY()` will print out short textual description of what `GetLastError()` reported
- Look at a Program : `ERROR.c`

Microsoft Thread Model

- Multithreaded applications

- Printing

- Spooling

- Have GUI thread and worker thread

- GUI Thread:

- take care of putting up windows and main message loop has a message queue

- Worker thread:

- performing time-consuming task, such as recalculation or repagination, that would cause the primary thread's message queue to become unresponsive

- You can see more detail in CH 11.

Background Printing

- Printing in the background is so enticing
- Using multiple threads, can let one thread produce the data for the printer while another thread controls the user interface.
- Background thread is completely separate from the main thread
- It uses separate data structure, separate device contexts, and is noninteractive

Conclusion

1. Separate the data between thread. Avoid global variables.
2. Do not share GDI objects between threads
3. Make sure you know the state of your threads. Do not exit with out waiting for them to shut down.
4. Let the primary handle the user interface.