

Multithreading Application in Win 32

(Chp_2 : Getting a feel for Threads)

Eun Sung Lee
twoss@mmlab.net

[Multimedia Lab.](#)

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

0. 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

1. Main Objectives

- Introduces the win32 calls for creating, monitoring, and exiting thread, shows you some thread in action

2. 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
- importance of CloseHandle()
- GetExitCodeThread()
- ExitThread()

3.1 Creating a Thread(con't)

Thread

Createthread()

Its prototype is

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

- **succeeds - a handle to the thread**
- **failure - FALSE**

3.1 Creating a Thread(con't)

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.

- square() function main() return

-

- 1 thread

3.1 Creating a Thread(con't)

- **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 thread

- ThreadFunc() gets called **asynchronously**

- **asynchronously**
 - : ThreadFunc() does not necessarily complete before main() continues on, so return value cannot be passed back in the conventional way

3.1 Creating a Thread

look at a Program : Numbers.c

- 5 thread
Function call

3.2 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

3.2 Consequence of Using Multiple Thread

Order of Execution is not guaranteed

Take Switches can happen anywhere, anytime

- context switch happened while the thread was in the middle of displaying its results
- Type of race condition
- 333344444444 (figure 2-1 colume2)

Threads are sensitive to small changes

Threads do not always start immediately

This problem can be solved by compiling the program with `/MT /MD`

3.3 Kernel Objects (Con't)

Kind of Kernel Objects

- Processes
- Threads
- Files
- Events
- Semaphores...

Object : An *object* is a data structure that represents a system resource, such as a file, thread, or graphic image

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

3.3 Kernel Objects (Con't)

CreateThread() values return .

- HANDLE

- CreateThread() return value
- local in the process (process)
- refers to a kernel object

- threadID

- returned with lpThreadId(parameter return)
- global value (가)
process 가)

3.4 CloseHandle() (con't)

Why use CloseHandle

- Need to release kernel object when you are finished using them.

Use CloseHandle() API function

Its prototype is

```
BOOL CloseHandle(  
HANDLE hObject // handle to object to close  
);
```

Return value

Success : TRUE

Fail : FALSE

3.4 CloseHandle()

Why call closehandle..?

- Process가 thread handle closing
thread , kernel
objects 가 Objects OS 가
Manage .
- Such resource leaks can have a significant negative impact on system performance.

thread object is that the HANDLE is to a thread kernel object, not the thread itself

reference count on thread is two

When you call `CloseHandle()` the count drops
and When the thread terminates the count drops

3.5 Exit Code(con't)

Why use EXITCODE

- main thread
main() 가 ?
sleep() trick
- 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()`

3.5 Exit Code (con't)

```
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*

3.5 Exit Code

Look at a Program : EXITCODE.c

- 2 thread
- thread

STILL_ALIVE == 259

3.6 Exiting a Thread

Sometimes necessary 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.

Use the API function `ExitThread()`

3.6 Exiting a Thread

Look at a Program : EXITTHREAD.c

- thread 가 ExitThread() 가 .
- ExitCode Check , ExitThread() 가
가 .

3.7 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`

3.8 Microsoft Thread Model

Recommends that multithreaded applications

- printing
- spooling
- Have GUI thread and worker thread
- GUI Thread : take care of putting up windows and main message loop
 - has a messagr 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.

3.9 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

4. 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 without waiting for them to shut down.
4. Let the primary handle the user interface