Windows Application Development

Chapter 7
Windows Thread Management

Threads: Benefits and Risks

- Benefits
  - Simpler program models
  - Faster code – in many cases
    - Exploit multiple processors
    - Exploit inherent application parallelism
  - Reliable, understandable, maintainable code
- Risks
  - Slower performance – in some cases
  - Potential defects

Contents

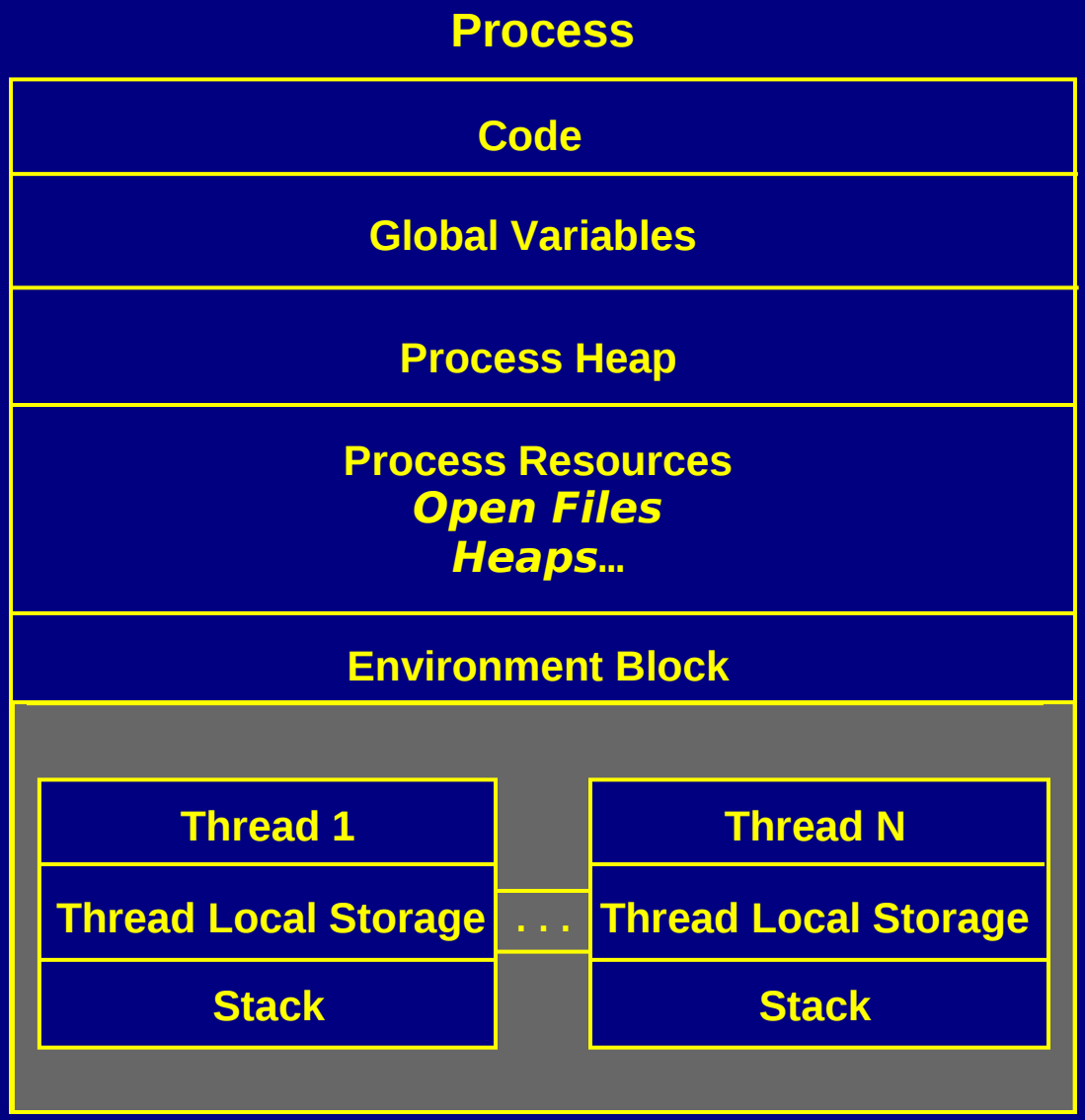
1. Process and Thread Overview

- Threads in a process share data and code
  - Each thread has its own stack for function calls
  - Calling thread can pass an argument to a thread at creation time
    - This argument is on the stack
  - Each thread can allocate its own Thread Local Storage (TLS) indices and set TLS values

Process and Thread Overview

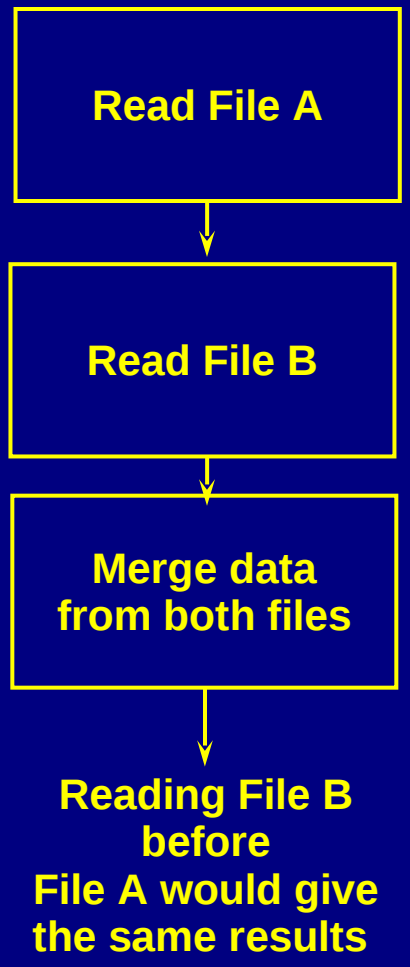
- Threads are scheduled and run independently
  - The executive schedules threads
  - Threads run asynchronously
  - Threads can be preempted
    - Or restarted at any time

Processes and Threads

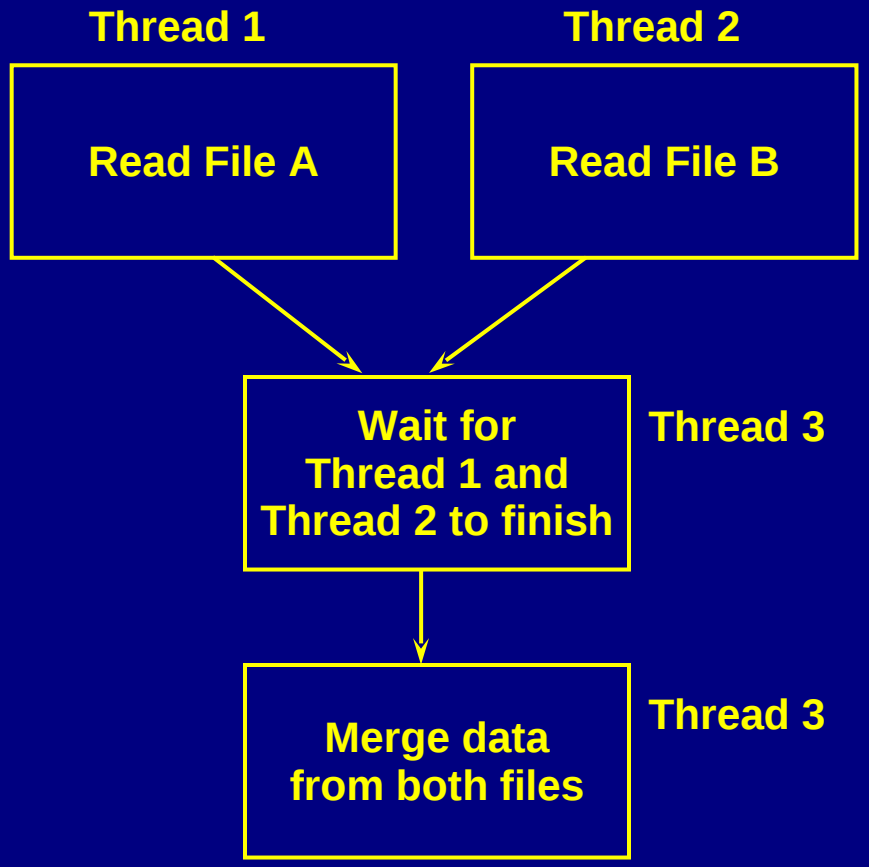


Threads Performing Parallel Tasks

Single-Threaded Program



Multithreaded Program



2. Thread Management

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

Creating a Thread (1 of 6)

- 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 (2 of 6)

- 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 (3 of 6)

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

Creating a Thread (4 of 6)

- 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 (5 of 6)

`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 (6 of 6)

dwCreate

- If zero, the thread is immediately ready to run
- If **CREATE_SUSPENDED**, the new thread will be in the suspended state, requiring a **ResumeThread** function call to move the thread to the ready state

lpIDThread

- Points to a **DWORD** that receives the new thread's identifier; **NULL** OK on W2000/NT

The Thread Function

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

Thread Termination (1 of 3)

- 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 Termination (2 of 3)

- **`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 Termination (3 of 3)

- 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 (1 of 2)

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

Thread Identities (2 of 2)

```
HANDLE GetCurrentThread (VOID);
DWORD GetCurrentThreadId (VOID);
HANDLE OpenThread (
    DWORD dwDesiredAccess,
    BOOL InheritableHandle,
    DWORD ThreadId );
        /* >= Windows 2000 only */
```

Suspend & Resume Threads (1 of 2)

- 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:

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

Suspend & Resume Threads (2 of 2)

```
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

3. 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 (2 of 2)

- **`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

The Wait Functions (1 of 2)

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

The Wait Functions (2 of 2)

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

- Return: The cause of the wait completion

Wait Options (1 of 2)

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

Wait Options (2 of 2)

- **`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 Function Return Values (1 of 3)

- **`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 Function Return Values (2 of 3)

- **`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 Function Return Values (3 of 3)

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

4. The C Library and Threads

- Nearly all programs (and thread functions) use the C library
- But the normal C library is not “thread safe”
- The C function **`_beginthreadex`** has exactly the same parameters as **`CreateThread`**

Using _beginthreadex (1 of 3)

- Cast the **_beginthreadex** return value to **(HANDLE)**
- Use **_endthreadex** in place of **ExitThread**
- **#include <process.h>**

Using _beginthreadex (2 of 3)

- Set the multithreaded environment as follows:
  - **`#define _MT`** in every source file before **`<windows.h>`**
  - Link with **`LIBCMT.LIB`**
    - Override the default library

Using _beginthreadex (3 of 3)

- *Preferred method* using Visual C++
- From the menu bar:
  - *Build Settings* — C/C++ Tab
  - *Code Generation category*
  - Select a multithreaded run-time library