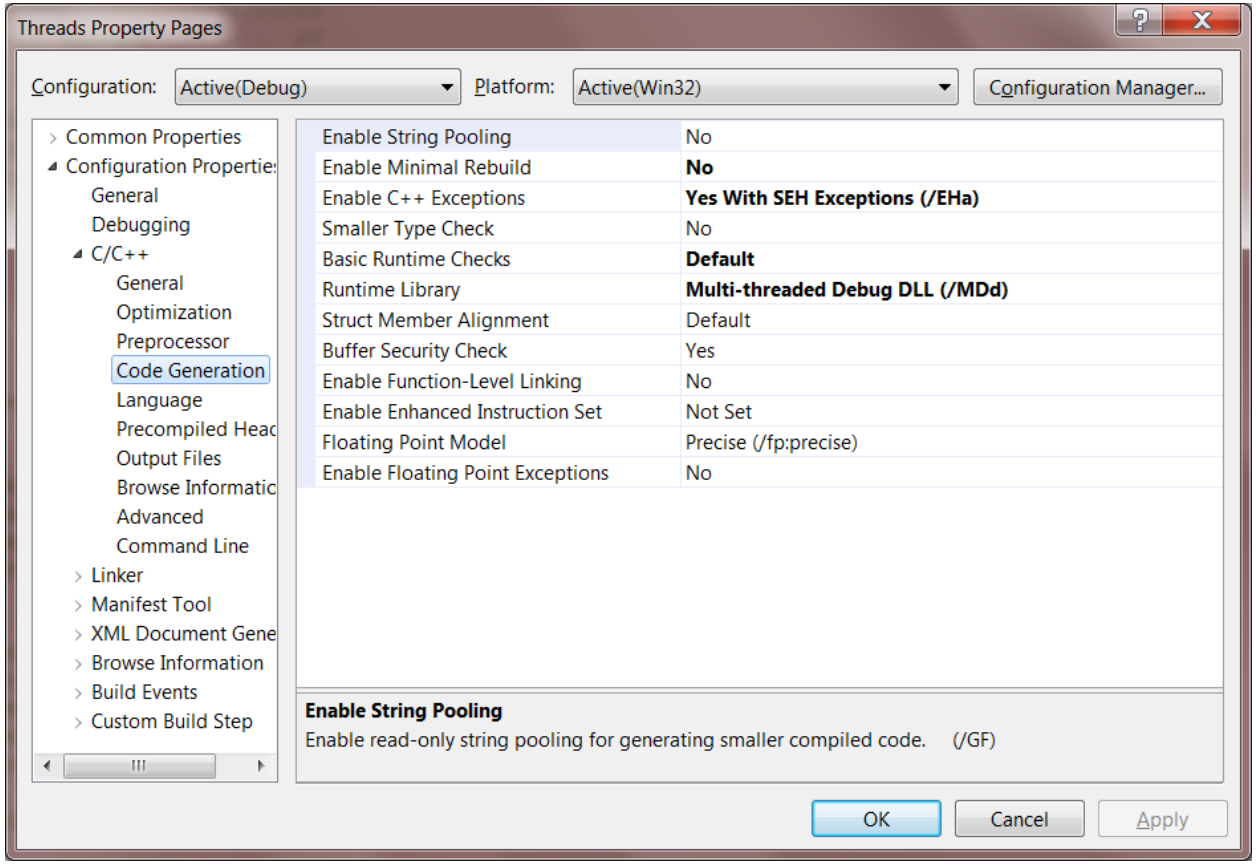
Creating and Using Threads – the Details

Jim Fawcett
Core Technologies
Spring 2010

You need to select multi-threaded debug or release C/C++ libraries in order to create and use threads in your programs. This is done with project settings, as shown below.



Creating and Using a New Thread

1. One way is to call `_beginthread`, passing a pointer to a function that the thread will execute. When the function returns the thread terminates.

```
unsigned long _beginthread(
  void(_cdecl *start_address)(void *),    // function pointer
  unsigned stack_size,                    // specify stack size
  void *arglist                           // pointer to struct of arguments
);
```

   a. `void(__cdecl *start_address)(void *)` is the declaration of a function pointer.
      i. The pointer is declared to point to a function with the signature `void start_address(void *pVoid)`.
      ii. `__cdecl` specifies the default calling convention for C and C++ programs. Note that windows programs typically use the `__stdcall` or PASCAL calling convention.
   b. `stack_size` allows you to specify a stack size. If you pass 0 a default size is used.
   c. `void *arglist` provides a way to pass an arbitrary number of arbitrary types as arguments to the thread function. Often we declare a struct to package up the arguments and pass its address, using a `void *` cast. Inside the thread function we declare a pointer to a struct of that type and assign the incoming pointer, using a cast, to get access to the arguments. Here's an example:

```
struct args { int x; double y; };

void _cdecl threadProc(void *pVoid) {
    struct args *pArgs = (args*)pVoid;
    int x = pArgs->x;
    double d = pArgs->y;

// do stuff with args

}
// inside a function that creates the thread

struct args myArgs = { 1, -3.5e-5 };
HANDLE hThrd = _beginthread(threadProc,0,(void*)&myArgs);
```

Creating and Using a New Thread (continued)

2. Another way is to call `_beginthreadex` which has some added functionality over `_beginthread`:

```
unsigned long _beginthreadex(
  void *security,                                 // pointer to security attributes
  unsigned stacksize,                             // specified stack size
  unsigned(__stdcall *start_address)(void),       // function pointer
  void *arglist,                                  // pointer to struct with arguments
  unsigned initflag,                              // 0 or CREATE_SUSPEND
  unsigned *thrdaddr                              // pointer to thread ID
);
```

   Note the difference in signatures between the function pointer here and the one used with `_beginthread`.

   a. If we pass `security = NULL` we get the default security settings for our platform.
   b. The `initflag` allows us to start the thread suspended. We might do this to perform some initialization before we let the thread start.

3. The parent must not terminate before its child thread finishes. We can arrange for that using:

```
DWORD WaitForSingleObject(
  HANDLE hThrd,             // handle to a kernal object (thread in our case)
  DWORD dwMilliseconds      // time-out : you may use INFINITE
);
```

   This function returns one of the values:

   a. `WAIT_OBJECT_0` - state of the object is signaled (thread terminated).
   b. `WAIT_TIMEOUT` - time-out interval elapsed, object is not signaled.
   c. `WAIT_ABANDONED` - waited on a mutex no longer owned
   d. `WAIT_FAILED` - failed for some other reason, object is not signaled

   The parent thread, after calling `_beginthread(ex)` simply calls `WaitForSingleObject` on the handle returned by the thread creation function.

Creating and Using Critical Sections

You may use a critical section to serialize access to a resource shared between two or more threads in the same process. You do that as follows:

1. Declare a critical section:

```
CRITICAL_SECTION cs;
```

2. Before using, you must initialize cs or the first use will throw an exception. You do this as follows:

```
InitializeCriticalSection(&cs);
```

3. You serialize access to a shared resource, e.g., variable, stream, or file, as follows:

```
EnterCriticalSection(&cs);
  // use the shared resource here
LeaveCriticalSection(&cs);
```

4. Finally, you must destroy the critical section this way:

```
DeleteCriticalSection(&cs);
```

It makes a lot of sense to define a locking class that declares a critical section as a data member. It would then:

1. initialize the critical section in its constructor
2. enter the critical section by calling `lock.set();`
3. leave the critical section by calling `lock.release();`
4. delete the critical section in its destructor

Any object you wish to make thread safe could then declare an instance of the lock class as one of its data members.

Creating and Using Mutex Objects

You may use a mutex to serialize access to a resource shared between two or more threads in the same process or in different processes. You can do that as follows:

5. Create a mutex kernel object:

```
HANDLE hMutex = CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,    // NULL for default platform security attributes
  BOOL bInitialOwner,                         // false if there is no initial owner
  LPCSTR lpName                               // NULL for an anonymous mutex
);
```

6. You serialize access to a shared resource, e.g., variable, stream, or file, as follows:

```
dwWaitResult = WaitForSingleObject(hMutex, INFINITE);
  // use the shared resource here
ReleaseMutex(hMutex);
```

7. Finally, you must destroy the mutex this way:

```
CloseHandle(hMutex);
```

We could define a locking class that declares a mutex as a data member. It would then:

5. create the mutex in its constructor
6. capture the mutex by calling `lock.set();`
7. release the mutex by calling `lock.release();`
8. destroy the mutex in its destructor

Any object you wish to make thread safe could then declare an instance of the lock class as one of its data members.

Creating and Using Event Objects

You may use an event object to serialize access to a resource shared between two or more threads in the same process or in different processes. You can do that as follows:

8. Create an event kernel object:

```
HANDLE hEvent = CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,    // NULL for default platform security attributes
  BOOL bManualReset,                          // if false automatically reset after single wait
  BOOL bInitialState,                         // if true initial state is signaled
  LPCSTR lpName                               // NULL for an anonymous event
);
```

9. You serialize access to a shared resource, e.g., variable, stream, or file, as follows:

```
dwWaitResult = WaitForSingleObject(hEvent, INFINITE);
  // use the shared resource here, event is unsignaled
SetEvent(hEvent);
  // now event is signaled again
```

10. Finally, you must destroy the event this way:

```
CloseHandle(hEvent);
```

We could define a locking class that declares an event as a data member. It would then:

9. create the event in its constructor
10. unsignal the event by calling `lock.set();`
11. signal the event by calling `lock.release();`
12. destroy the event in its destructor

Any object you wish to make thread safe could then declare an instance of the lock class as one of its data members.

Communication between Threads

How threads exchange information depends on whether they run in the same process, or different processes, and perhaps on different machines.

1. Threads in the same process can communicate by using:
    a. Passed references, e.g., a pointer to a queue or struct in the parent thread.
    b. Shared static queue or struct
    c. Shared global variables
    d. Posting messages to the window of another thread, using the handle of the receiving window.

2. Threads in different processes can communicate by using:
    a. Memory mapped files
    b. Posting messages to a window in the receiving process, using the handle of the receiving window.
    c. Sockets[1] or named pipes.

3. Threads on different machines can communicate by using:
    a. Sockets or named pipes over TCP/IP[2].
    b. Remote procedure calls (RPCs).
    c. Simple Object Access Protocol (SOAP), which uses XML[3] over HTTP[4].

We have already discussed 1a, 1b, and 1c. We will discuss 2a, 2c, and 3a.

[1] Sockets provide an I/O abstraction that uses TCP/IP protocols, and others, to transfer data between processes and machines.
[2] Transmission Control Protocol / Internet Protocol (TCP/IP)
[3] eXtensible Markup Language (XML)
[4] HyperText Transfer Protocol (HTTP)

Threads and Message Processing

1. Windows messages are processing using a GetMessage loop:

```
MSG msg;
While (GetMessage(&msg, NULL, 0, 0) {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

2. This code loops, reading each message in the message queue, and calls the window procedure for each message. The loop exits when `GetMessage` returns `FALSE`, e.g., when a `WM_QUIT` message is processed.

3. Each thread that processes messages has its own message queue. The operating system routes messages for a particular window to the message queue of the thread that created it.

   Even though you post a message from another thread, by:

```
BOOL PostMessage(hWnd, msgType, wParam, lParam);
```

   The message will be posted to the queue of the thread that created the window with `handle hWnd;`

There is a function for synchronously sending a message to a window:

```
LRESULT SendMessage(hWnd, msgType, wParam, lParam);
```

If the message is being sent to a window on the same thread, `SendMessage` simply calls the window procedure of the window. If the message goes to a window created by another thread, the operating system puts the calling thread to sleep, activates the window thread to process the message, then wakes the calling thread, passing it the return result.

Sample Code in the class directories:

In ecs.syr.edu/faculty/fawcett/handouts/CoreTechnologies/ThreadsAndSynchronization/code you will find the folders:

1. spawnWin32Proc:
   illustrates how one process can create another, pass it command line arguments, and read its output.
2. dialogDemo:
   demonstrates how User Interface (UI) threads and worker threads can operate together to make a responsive interface.
3. QueuedMessagesWithBlockingQueue
   shows how to communicate between threads using the thread-safe blockingQueue.
4. SYNCH:
   has four subfolders which all use the same basic example to illustrate:
   a. NOSYNCH – shared memory conflicts if you don't use synchronization
   b. CSSYNCH – how to synchronize using critical sections
   c. EVSYNCH – how to synchronize using an event object
   d. MTSYNCH – how to synchronize using a mutex object

   Two additional folders are included:
   e. processSynch – illustrates how a named mutex can be used to synchronize threads in different processes.
   f. utils – provides some useful utility functions for handling errors and other stuff.
5. thread problems:
   has five subfolders that illustrate:
   a. deadlock – how multiple threads that share two or more resources can deadlock
   b. race – how the behavior of a multi-theaded program can depend on race conditions, e.g., which thread finishes first.
   c. shareConflict – another example of corruption of shared resources, in this case a shared C++ string object.
   d. shareNoConflict – removal of the conflict with critical sections.
   e. starve1 – starvation of a thread that often blocks, when competing with a CPU intensive thread.