Lecture

Review of Last Lecture

- DLLs and threads
- _beginthread()
- _endthread()
- CreateThread()
- Thread procedure: THREADPROC
- ExitThread()
- Supend/ResumeThread(), Sleep()
- Thread object and thread handles

Some OS concepts

- Co-operative vs. Pre-emptive multitasking

- CreateThread():
  Thread object remain in the Win32 system until CloseHandle() is called for all handles to a thread.

CreateThread()

```
enum Shape { RECTANGLE, ELLIPSE };
DWORD WINAPI drawThread(LPVOID shape);
SYSTEMTIME st;

HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
  DWORD dwStackSize,                        // initial stack size
  LPTHREAD_START_ROUTINE lpStartAddress,    // thread function
  LPVOID lpParameter,                       // thread argument
  DWORD dwCreationFlags,                    // creation option
  LPDWORD lpThreadId                        // thread identifier
);
```

Threads example

```
hThread1 = CreateThread(NULL, 0, drawThread,
(LPVOID)RECTANGLE, CREATE_SUSPENDED, &dwThread1);
hThread2 = CreateThread(NULL, 0, drawThread,
(LPVOID)ELLIPSE, CREATE_SUSPENDED, &dwThread2);

hDC = GetDC(hWnd);
hBrushRectangle=CreateSolidBrush(RGB(170,220,160));
hBrushEllipse =
  CreateHatchBrush(HS_BDIAGONAL,RGB(175,180,225));

InitializeCriticalSection(&cs);

srand( (unsigned)time(NULL) );
ResumeThread(hThread2);
ResumeThread(hThread1);
```

Threads example

```
DWORD WINAPI drawThread(LPVOID type)
{
   int i;

   if((enum Shape)type == RECTANGLE)
   {
       for(i=0; i<10000; ++i)
       {
              EnterCriticalSection(&cs);
              SelectObject(hDC, hBrushRectangle);
Rectangle(hDC, 50, 1, rand()%300, rand()%100);
              GetLocalTime(&st);
   LeaveCriticalSection(&cs);
              Sleep(10);
       }
   }
```

Threads Synchronization

- The problem: GDI handle is shared among threads. Only one thread must draw using this handle, at one time
- SYSTEMTIME
- The solution: Critical Section Kernel Object

Critical Section Object

- CRITICAL_SECTION object
- InitializeCriticalSection
- EnterCriticalSection
- LeaveCriticalSection
- DeleteCriticalSection

Synchronisation Objects

- Other Kernel Objects basics
- Wait functions: Signalled and non-signalled states
- Mutex Object: *Mut*ual *Ex*clusion
- Event Object: Similar to a flag
- Semaphore Object: A counter based object

The Mutex Object

- Description

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,  // SD
  BOOL bInitialOwner,                       // initial owner
  LPCTSTR lpName                            // object name
);

DWORD WaitForSingleObject(
  HANDLE hHandle,          // handle to object
  DWORD dwMilliseconds     // time-out interval
   INFINITE, WAIT_TIMEOUT, WAIT_OBJECT_0
);
```

The Mutex Object

```
BOOL ReleaseMutex(
   HANDLE hMutex    // handle to mutex
);
```

When the Mutex object is no longer needed, call

```
CloseHandle( handle );
```

The Mutex Object

- Named and un-named Mutex objects
- Opening a handle to a Mutex
- When we try to create Mutex with some name and a Mutex with that name already exists, `GetLastError()` returns `ERROR_ALREADY_EXISTS`

Threads example with Mutex

```
hThread1 = CreateThread(NULL, 0, drawThread,
  (LPVOID)RECTANGLE, CREATE_SUSPENDED,
  &dwThread1);
  hThread2 = .. .. ..

hBrushRectangle =
  CreateSolidBrush(RGB(170,220,160));
hBrushEllipse=CreateHatchBrush(HS_BDIAGONAL,
  RGB(175,180,225));

hMutex=CreateMutex(NULL, 0, NULL);

  srand( (unsigned)time(NULL) );
  ResumeThread(hThread2);
```

Threads example

```
      for(i=0; i<10000; ++i)
      {
Switch(WaitForSingleObject(hMutex,
  INFINITE))
{
  case WAIT_OBJECT_0:
  SelectObject(hDC, hBrushRectangle);
   Rectangle(hDC, 50, 1, rand()%300,
  rand()%100);
  GetLocalTime(&st);
ReleaseMutex(hMutex);
          Sleep(10);

};
```

Synchronisation Objects

- Critical Section can't be accessed outside a process while named Mutex objects can be!

- Problem: Detecting whether another instance of the same Win32 application is running
- In Win32 2nd parameter, `hPrevInstance` passed to `WinMain()` is always `NULL`

Multiple instance of the same application

- Create a Named Mutex with some unique name using **`CreateMutex()`**
- If there is error creating Mutex object, call **`GetLastError()`**
- If **`GetLastError()`** returns **`ERROR_ALREADY_EXISTS`**, another instance of the same application is already running

Event Object

Description:

- CreateEvent()
- Manual Reset and Auto-reset events
- SetEvent(), ResetEvent()
- PulseEvent()

Event Object

- Description: Used as a flag to mark the occurrence of an event

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
                                  // Security Descriptor
  BOOL bManualReset,              // reset type
  BOOL bInitialState,             // initial state
  LPCTSTR lpName                  // object name
);
```

Waiting for Multiple Objects

```
DWORD WaitForMultipleObjects(
  DWORD nCount,     // number of handles in array
  CONST HANDLE *lpHandles,  // object-handle array
  BOOL fWaitAll,            // wait option
  DWORD dwMilliseconds      // time-out interval
);
```

Returns:

`WAIT_OBJECT_0` to `(WAIT_OBJECT_0 + nCount - 1)`

Semaphore Object

- Description: Limiting the maximum number of threads in a system. Utilisation of a resource

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // SD
  LONG lInitialCount,     // initial count
  LONG lMaximumCount,     // maximum count
  LPCTSTR lpName          // object name
);
```

Semaphore Object

`lInitialCount`

Specifies an initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to lMaximumCount. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the ReleaseSemaphore() function.

User Interface threads

```
DWORD MsgWaitForMultipleObjects(
  DWORD nCount,          // number of handles in array
  CONST HANDLE pHandles, // object-handle array
  BOOL fWaitAll,         // wait option
  DWORD dwMilliseconds,  // time-out interval
  DWORD dwWakeMask       // input-event type
);
```

Thread Local Storage

```
declspec (thread) int global;
declspec (thread) static inside_fuction;
```

```
DWORD TlsAlloc(VOID);
```

Allocates a TLS index

```
BOOL TlsSetValue(
  DWORD dwTlsIndex,   // TLS index
  LPVOID lpTlsValue   // value to store
);
```

Stores a value in the calling thread's thread local storage (TLS) slot for the specified TLS index. Each thread of a process has its own slot for each TLS index.

Thread Local Storage

```
LPVOID TlsGetValue(
  DWORD dwTlsIndex    // TLS index
);

BOOL TlsFree(
  DWORD dwTlsIndex    // TLS index
);
```

Advantage and Disadvantage of DLLs

- Advantages
  - Common services can be grouped
  - ISVs (Independent Software Vendors) can ship DLLs independently
  - Lesser code to be written by programmers
- Disadvantages
  - DLLs are slower than statically linked code
  - Versioning may be cumbersome
  - C-runtime routines are statically linked with every EXE and DLL. When a single process loads multiple DLLs, statically linked C-runtime gets duplicated.