Windows Internals Course
Thread/Synchronization Exercises

金田憲二
kaneda@yl.is.s.u-tokyo.ac.jp

平成 15 年 7 月 25 日

目 次

1 実行環境の設定

Windows 用の C コンパイラでは，以下のようなものが取得可能です.

- **Platform SDK**
  `http://www.microsoft.com/msdownload/platformsdk/sdkupdate/`
  から取得可能.
- **Intel (R) C++ Compiler for Windows (30 日体験版)**
  `http://www.intel.com/jp/developer/software/products/global/eval.htm`
  から取得可能.
- **Borland C++ Compiler 5.5 (体験版)**
  `http://www.borland.co.jp/cppbuilder/freecompiler/` から取得可能.

1.1 実行例

Windows DDK をフルインストールすると，Microsoft (R) 32-bit C/C++ Optimizing Compiler (実行ファイル名 `cl`) が付属してきます．これを用いて C プログラムをコンパイルすることができます．ただし，Common Dialog などのライブラリを使用する際には，DDK の提供するヘッダファイルだけでは不十分で，Platform SDK などの提供するヘッダファイルが必要となります．

例えば Windows DDK が `C:\WINDDK` にインストールされており，Platform SDK が `C:\Program Files\Microsoft SDK` にインストールされているとします．このとき,「スタートメニュー」→「全てのプログラム」→「Development Kits」→「」→「Windows DDK 3663」→「Build Environments」→「Windows XP」→「Win XP Free Build Environment」を選択すると，コマンドプロンプトを立ち上がります．このコマンドプロンプト上で

```
cl /I "C:\Program Files\Microsoft SDK\include"
   /I C:\WINDDK\3663\inc\crt source.c
   /link /LIBPATH:C:\WINDDK\3663\lib\wxp\i386
```

というように入力することによって，プログラムをコンパイルすることができます．

2 Win32 API について

Win32 API は，Windows への，基盤となるインターフェースです．より詳しくは MSDN 等を参照して下さい．例えば，`http://www.microsoft.com/japan/msdn/default.asp`) の「Windows 開発」→ 「Windows ベースサービス」に，Win32 の提供する関数の説明があります．

表 1: windows.h で新たに定義されているデータ型

BOOL	Boolean variable (should be TRUE or FALSE).
BOOLEAN	Boolean variable (should be TRUE or FALSE).
CHAR	8-bit Windows (ANSI) character
UCHAR	Unsigned CHAR
WORD	16-bit unsigned integer
INT	32-bit signed integer
UINT	Unsigned INT
DWORD	32-bit unsigned integer
LONG	32-bit signed integer
ULONG	Unsigned LONG.
ULONGLONG	64-bit unsigned integer
FLOAT	Floating-point variable.
VOID	Any type
LPDWORD	DWORD へのポインタ
LPVOID	VOID へのポインタ
HANDLE	オブジェクト（ファイル，スレッドなど）へのハンドル
WINAPI	システム関数のための Calling convention

2.1 ヘッダファイル

windows.h は，様々なヴァージョンの Microsoft Windows 上でソースコードを書くことを助ける，定義やマクロやデータ構造の書かれたファイルです．Win32API を使う際は，windows.h をインクルードして下さい．

2.2 データ型

windows.h では，データ型が新しくいくつか定義されています。その一部を表 1 に載せます．

2.3 ハンドラの生成・削除

課題中では，以下のようなオブジェクトを扱います．

- スレッド
- イベント
- ミューテックス
- セマフォ

一般に，XXXX という名前のオブジェクトを作成し，そのハンドラを得るには，CreatexXXXX 関数を用います．例えば，CreateThread 関数はスレッドを作成し，その作成したスレッドのハンドラを返します．

CloseHandle 関数は，こうして作成オブジェクトのハンドルを閉じるのに用いられます．例えば，CreateThread 関数が返したスレッドオブジェクトのハンドルを閉じるに，CloseHandle 関数を用います．あるオブジェクトに対する全てのハンドルが閉じられると，そのオブジェクトはシステムから削除されます．

2.3.1 CloseHandle

開いているオブジェクトハンドルを閉じます．

```
BOOL CloseHandle(
  HANDLE hObject   // オブジェクトのハンドル
);
```

- `hObject` 開いているオブジェクトのハンドルを指定します．
- **戻り値** 関数が成功すると，0 以外の値が返ります．関数が失敗すると，0 が返ります．

3 Experiment 1 & 2: Priority Inversion

この章では，Experiment 1 と 2 で必要となる API について説明します．

まず，スレッドの生成・優先順位の設定方法について説明します．次に，Win32 の提供する同期プリミティブについて述べます．同期プリミティブには，以下のようなものがあります．

イベント イベントオブジェクトは，イベントが発生したときに待機中のスレッドに通知を行います．一つ以上のスレッドがイベントオブジェクトを待機することができます．

クリティカルセクション 一度に一つのスレッドしか共有リソースにアクセス出来ないようにします．同じプロセス内のスレッド間でのみしか，同期は取ることはできません．

ミューテックス クリティカルセクションと同様，一度に一つのスレッドしか共有リソースにアクセス出来ないようにします．ただクリティカルセクションと異なり，複数プロセスにまたがって同期を取ることができます．

セマフォ 共有リソースにアクセスできるスレッドの数を制限します．

また，この章の最後で，実行時間の計測方法について述べます．

3.1 スレッドの生成・削除

- CreateThread 関数によってスレッドを生成することができます．スレッドが実行する関数と，その関数へ渡す引数を指定します．
- CloseHandle 関数で，ハンドルを閉じ，リソースを解放します．

以下はスレッド生成の例です．

```
#include <windows.h>
#include <conio.h>

DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
    char szMsg[80];

    wsprintf( szMsg, "Parameter = %d.", *(DWORD*)lpParam );
    MessageBox( NULL, szMsg, "ThreadFunc", MB_OK );

    return 0;
}

VOID main( VOID )
{
    DWORD dwThreadId, dwThrdParam = 1;
    HANDLE hThread;
    char szMsg[80];

    hThread = CreateThread(
        NULL,                   // default security attributes
        0,                      // use default stack size
        ThreadFunc,             // thread function
        &dwThrdParam,           // argument to thread function
        0,                      // use default creation flags
        &dwThreadId);           // returns the thread identifier

   // Check the return value for success.

   if (hThread == NULL)
   {
      wsprintf( szMsg, "CreateThread failed." );
```

```
      MessageBox( NULL, szMsg, "main", MB_OK );
   }
   else
   {
      _getch();
      CloseHandle( hThread );
   }
}
```

3.1.1 CreateThread

呼び出し側プロセスの仮想アドレス空間で実行するべき 1 個のスレッドを作成します.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // セキュリティ記述子
  DWORD dwStackSize,                        // 初期のスタックサイズ
  LPTHREAD_START_ROUTINE lpStartAddress,    // スレッドの機能
  LPVOID lpParameter,                       // スレッドの引数
  DWORD dwCreationFlags,                    // 作成オプション
  LPDWORD lpThreadId                        // スレッド識別子
);
```

- lpThreadAttributes　SECURITY_ATTRIBUTES 構造体へのポインタを指定します. この構造体で, 子プロセスが取得したハンドルを継承できるかどうかを指定します. NULL を指定すると, 取得したハンドルを継承できません.
- dwStackSize　スタックの初期のコミットサイズを, バイト単位で指定します. 0 または既定のコミットサイズより小さい値を指定すると, 呼び出し側スレッドのコミットサイズと同じサイズが, 既定のサイズとして割り当てられます.
- lpStartAddress　LPTHREAD_START_ROUTINE 型のアプリケーション定義関数へのポインタを指定します. この関数を, 生成されたスレッドは実行します.

  LPTHREAD_START_ROUTINE 型は,

  ```
  DWORD WINAPI ThreadProc(LPVOID lpParameter);
  ```

  となっています.

- lpParameter　スレッドに渡すパラメータの値を指定します.
- dwCreationFlags　スレッド作成に関する制御フラグを指定します. CREATE_SUSPENDED フラグを指定すると，新しいスレッドは中断された状態で作成され，ResumeThread 関数を呼び出すまでは動作しません. 0 を指定すると，作成と同時に新しいスレッドが動作します.
- lpThreadId　変数へのポインタを指定します. 関数から制御が返ると，この変数にスレッド識別子が格納されます. このパラメータで NULL を指定すると，スレッド識別子は格納されません.
- **戻り値**　関数が成功すると，新しいスレッドのハンドルが返ります. 関数が失敗すると，NULL が返ります.

3.1.2　GetCurrentProcess

現在のプロセスに対応する疑似ハンドルを取得します.

```
HANDLE GetCurrentProcess(VOID);
```

- **戻り値**　現在のスレッドの疑似ハンドルが返ります.

3.1.3　GetCurrentThread

現在のスレッドの擬似ハンドルを取得します.

```
HANDLE GetCurrentThread(VOID);
```

- **戻り値**　現在のスレッドの疑似ハンドルが返ります.

3.1.4　GetCurrentThreadId

呼び出し側スレッドのスレッド識別子を取得します.

```
DWORD GetCurrentThreadId(VOID);
```

- **戻り値**　呼び出し側スレッドのスレッド識別子が返ります.

3.2　スレッドの優先順位の設定

各スレッドの基本優先順位レベルは，所属プロセスの優先順位クラスと，そのスレッドの相対優先順位値によって決まります. 前者は SetPriorityClass 関数によって設定します. 後者は SetThreadPriority 関数によって設定します.

3.2.1 SetPriorityClass

指定されたプロセスの優先順位クラスを設定します.

```
BOOL SetPriorityClass(
  HANDLE hProcess,         // プロセスのハンドル
  DWORD dwPriorityClass    // 優先順位クラス
);
```

- hProcess　プロセスのハンドルを指定します.
- dwPriorityClass　プロセスの優先順位クラスを指定します.

  順位クラスの値としては,

  - IDLE_PRIORITY_CLASS (優先度レベル 4)
  - NORMAL_PRIORITY_CLASS (優先度レベル 8)
  - HIGHL_PRIORITY_CLASS (優先度レベル 13)
  - REALTIME_PRIORITY_CLASS (優先度レベル 24)

  などがあります.
- **戻り値**　関数が成功すると，0 以外の値が返ります．関数が失敗すると，0 が返ります.

REALTIME_PRIORITY_CLASS に設定するためには，ユーザは Administrator か Power User でなければいけません.

現在走っているプロセスの優先順位クラスを設定するには，以下のようにします.

```
SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS)
```

3.2.2 SetThreadPriority

指定されたスレッドの相対優先順位値を設定します.

```
BOOL SetThreadPriority(
  HANDLE hThread, // スレッドのハンドル
  int nPriority   // スレッドの相対優先順位値
);
```

- hThread　相対優先順位値を設定するべきスレッドのハンドルを指定します.

- nPriority　スレッドの相対優先順位値を指定します．次の値のいずれかを指定します．
  - THREAD_PRIORITY_ABOVE_NORMAL スレッド標準の相対優先順位値より 1 ポイント高い相対優先順位値を指定します．
  - THREAD_PRIORITY_BELOW_NORMAL スレッド標準の相対優先順位値より 1 ポイント低い相対優先順位値を指定します．
  - THREAD_PRIORITY_HIGHEST スレッド標準の相対優先順位値より 2 ポイント高い相対優先順位値を指定します．
  - THREAD_PRIORITY_LOWEST スレッド標準の相対優先順位値より 2 ポイント低い相対優先順位値を指定します．
  - THREAD_PRIORITY_IDLE プロセスの優先順位クラスが REALTIME_PRIORITY_CLASS の場合は，基本優先順位レベルとして 16 を指定します．それ以外の場合は，基本優先順位レベルとして 1 を指定します．
  - THREAD_PRIORITY_NORMAL スレッド標準の相対優先順位値を 0 ポイントに指定します．
- **戻り値**　関数が成功すると，0 以外の値が返ります．関数が失敗すると，0 が返ります．

すべてのスレッドは，開始の際に THREAD_PRIORITY_NORMAL を割り当てられています．

現在走っているスレッドの優先順位を設定するには，以下のようにします．

```
SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_ABOVE_NORMAL);
```

3.2.3 GetPriorityClass

指定されたプロセスの優先順位クラスを返します．

```
DWORD GetPriorityClass(
  HANDLE hProcess   // プロセスのハンドル
);
```

- hProcess　プロセスのハンドルを指定します．
- **戻り値**　関数が成功すると，指定したプロセスの優先順位クラスが返ります．関数が失敗すると，0 が返ります．

3.2.4 GetThreadPriority

指定されたスレッドの相対優先順位値を取得します.

```
int GetThreadPriority(
  HANDLE hThread    // スレッドのハンドル
);
```

- hThread　スレッドのハンドルを指定します.
- **戻り値**　関数が成功すると，指定したスレッドの相対優先順位値が返ります．関数が失敗すると，THREAD_PRIORITY_ERROR_RETURN が返ります.

3.3 待機関数

待機関数は，スレッドが実行をブロックすることを可能にします．イベント，セマフォ，ミューテックスなどの同期オブジェクトなどによって利用されます．ここでは，以下の 2 つの待機関数を取り上げます.

- WaitForSingleObject 関数
  同期オブジェクト（スレッド，イベント，セマフォ，ミューテックスなど）のハンドルを引数としてとります．以下のうちどれかの条件が満たされると，制御を返します.
  - 指定されたオブジェクトがシグナル状態になった場合. 例えばミューテックスは，どのスレッドにも所有されていない場合にシグナル状態になります.
  - 指定したタイムアウト時間を経過した場合.
- WaitForMultipleObjects 関数
  一つまたは複数の同期オブジェクトのハンドルを引数としてとります. 以下のうちどれかの条件が満たされると，制御を返します.
  - 指定したオブジェクトのうち一つがシグナル状態になったか，すべてのオブジェクトがシグナル状態になった場合（fWailall 待機オプションにより条件は異なる）.
  - 指定したタイムアウト時間を経過した場合.

以下の例は， CreateEvent 関数で 2 つのイベントオブジェクトを生成し，WaitForMultipleObjects 関数を用いて，どちらかのオブジェクトの状態がシグナルになるのを待ちます.

```
HANDLE hEvents[2];
DWORD i, dwEvent;

// Create two event objects.

for (i = 0; i < 2; i++)
{
    hEvents[i] = CreateEvent(
        NULL,   // no security attributes
        FALSE,  // auto-reset event object
        FALSE,  // initial state is nonsignaled
        NULL);  // unnamed object

    if (hEvents[i] == NULL)
    {
        printf("CreateEvent error: %d\n", GetLastError() );
        ExitProcess(0);
    }
}

// The creating thread waits for other threads or processes
// to signal the event objects.

dwEvent = WaitForMultipleObjects(
    2,           // number of objects in array
    hEvents,     // array of objects
    FALSE,       // wait for any
    INFINITE);   // indefinite wait

// Return value indicates which event is signaled.

switch (dwEvent)
{
    // hEvent[0] was signaled.
    case WAIT_OBJECT_0 + 0:
        // Perform tasks required by this event.
        break;

    // hEvent[1] was signaled.
```

```
    case WAIT_OBJECT_0 + 1:
        // Perform tasks required by this event.
        break;

    // Return value is invalid.
    default:
        printf("Wait error: %d\n", GetLastError());
        ExitProcess(0);
}
```

3.3.1 WaitForSingleObject

次のいずれかが成立すると，制御を返します．

- 指定されたオブジェクトがシグナル状態になった．
- タイムアウト時間が経過した．

```
DWORD WaitForSingleObject(
  HANDLE hHandle,        // オブジェクトのハンドル
  DWORD dwMilliseconds   // タイムアウト時間
);
```

- hHandle　オブジェクトのハンドルを指定します．
- dwMilliseconds　タイムアウト時間を，ミリ秒（ms）単位で指定します．タイムアウト時間が経過すると，オブジェクトが非シグナル状態であっても制御を返します．0 を指定すると，この関数は指定されたオブジェクトの状態を調べ，即座に制御を返します．INFINITE を指定すると，オブジェクトがシグナル状態になるまで待機し続けます．
- **戻り値**　関数が成功すると，関数が制御を返した原因が返ります．次の値のいずれかになります．
  - WAIT_ABANDONED 指定されたオブジェクトは，放棄されたミューテックスオブジェクトでした（あるスレッドが所有権を持っていましたが，そのスレッドは所有権を解放しないで終了しました）．この関数を呼び出した結果，そのミューテックスオブジェクトの所有権は呼び出し側スレッドに移り，そのミューテックスは非シグナル状態に設定されました．
  - WAIT_OBJECT_0　指定したオブジェクトがシグナル状態になったことを意味します．

– WAIT_TIMEOUT タイムアウト時間が経過し，指定されたオブジェクトが非シグナル状態であったことを意味します．
関数が失敗すると，WAIT_FAILED が返ります．

3.3.2 WaitForMultipleObjects

次のいずれかが成立すると，制御を返します．

- 指定されたオブジェクトの 1 つまたはすべてがシグナル状態になった．
- タイムアウト時間が経過した．

```
DWORD WaitForMultipleObjects(
  DWORD nCount,             // 配列内のハンドルの数
  CONST HANDLE *lpHandles,  // オブジェクトハンドルからなる配列
  BOOL fWaitAll,            // 待機オプション
  DWORD dwMilliseconds      // タイムアウト時間
);
```

- nCount lpHandles パラメータが指す配列の中にあるオブジェクトハンドルの数を指定します．オブジェクトハンドルの最大数は，MAXIMUM_WAIT_OBJECTS です．
- lpHandles 複数のオブジェクトハンドルからなる配列へのポインタを指定します．この配列には，異なるタイプのオブジェクトのハンドルを格納することもできます．しかし，同じハンドルの複数のコピーを格納することはできません．
- fWaitAll 待機のタイプを指定します．TRUE を指定すると，lpHandles 配列内のすべてのオブジェクトがシグナル状態になったときに制御が返ります．FALSE を指定すると，lpHandles 配列内のオブジェクトのどれか 1 つがシグナル状態になったときに制御が返ります．この場合，戻り値は，関数が制御を返す状況をもたらした（つまり，シグナル状態になった）オブジェクトを表します．
- dwMilliseconds タイムアウト時間を，ミリ秒（ms）単位で指定します．タイムアウト時間が経過すると，fWaitAll パラメータで指定した条件が満たされなくても，制御が返ります．0 を指定すると，この関数は指定されたオブジェクトの状態を調べ，即座に制御を返します．INFINITE を指定すると，条件が満たされるまで待機し続けます．
- **戻り値** 関数が成功すると，関数が制御を返す状況をもたらしたイベントを示す値が返ります．次の値のいずれかになります．

- WAIT_OBJECT_0 以上（WAIT_OBJECT_0 + nCount - 1）以下 fWaitAll が TRUE の場合，指定されたすべてのオブジェクトがシグナル状態になったことを意味します．fWaitAll が FALSE の場合，lpHandles パラメータ内で（戻り値 - WAIT_OBJECT_0）番目のオブジェクトが待機条件を満たしたことを意味します．この関数を呼び出して実行している間に複数のオブジェクトがシグナル状態になった場合は，それらのオブジェクトのうち，最小のインデックス番号が返ります．
- WAIT_ABANDONED_0 以上 （WAIT_ABANDONED_0 + nCount - 1）以下 fWaitAll が TRUE の場合，指定されたすべてのオブジェクトがシグナル状態になったこと，およびこれらのオブジェクトのうち，少なくとも 1 つが，放棄されたミューテックスオブジェクト（あるスレッドが所有権を持っていましたが，そのスレッドは所有権を解放しないで終了しました）であったことを意味します．fWaitAll が FALSE の場合，lpHandles パラメータ内の（戻り値 - WAIT_ABANDONED_0）番目のオブジェクトが，待機条件を満たした，放棄されたミューテックスオブジェクトであったことを意味します．
- WAIT_TIMEOUT タイムアウト時間が経過し，fWaitAll パラメータで指定された条件が満たされていないことを意味します．
  関数が失敗すると，WAIT_FAILED が返ります．

3.4 イベント

イベントは同期オブジェクトであり，二つの状態（シグナルと非シグナル）を取ります．イベントは，一般にスレッドの実行の同期を取るのに用いられます．

- CreateEvent 関数は，イベントオブジェクトを作成します．
- SetEvent 関数は，イベントオブジェクトの状態をシグナル（イベント発生を通知している状態）に設定します．
- ResetEvent 関数は，イベントオブジェクトの状態を非シグナル（イベント発生を通知していない状態）に設定します．
- PulseEvent 関数は，指定されたイベントオブジェクトの状態をシグナルにしてから非シグナルにリセットします．待機中のスレッドを解放する場合に用います．
- WaitForSingleObject 関数は，イベント（または任意のオブジェクト）を待機します．

- CloseHandle 関数で，ハンドルを閉じ，リソースを解放します．

SetEvent または PulseEvent 関数を呼び出してイベントオブジェクトを明示的にシグナル状態に設定するまでは，イベントオブジェクトは非シグナル状態にとどまっています．

イベントオブジェクトが非シグナル状態である場合，待機関数（WaitForSingleObject など）を呼び出すと，その呼び出しスレッドの実行がブロックされます．

SetEvent または PulseEvent 関数を呼び出してイベントオブジェクトを明示的にシグナル状態に設定すると，待機関数は制御を返します．この時の動作は，イベントオブジェクトが手動リセットイベントに設定されているか自動リセットイベントに設定されているかによって異なります（この設定はオブジェクト生成時に行ないます）．

手動リセットオブジェクトの場合 即座に解放できる待機スレッドをすべて解放します．ResetEvent 関数を呼び出して明示的に非シグナル状態にするまでは，シグナル状態のままです．

自動リセットオブジェクトの場合 待機スレッドが存在しない場合は，シグナル状態のままです．

待機スレッドが存在する場合は，そのうちの 1 つだけを解放します．その後，イベントオブジェクトを非シグナル状態に戻し，制御を返します．

以下はサンプルプログラムです．

共有バッファが存在し，いくつかのスレッドが，その共有バッファから読み込みを行ないます．マススレッドが一つ存在し，このスレッドは共有バッファに書き込みを行ないます．この時イベントオブジェクトを用いて，読み込みと書き込みの同期を取ります．

マススレッドは，CreateEvent 関数を呼び出し，手動リセットオブジェクトを生成します．バッファに書き込みをする時に，このイベントを非シグナル状態にします．書き込みが終了した時に，シグナル状態にします．

また，読み込みスレッドと，そのスレッドごとに自動リセットオブジェクトを生成します．読み込みスレッドは，バッファを読んでいない時，このイベントオブジェクトをシグナル状態にします．

```
#define NUMTHREADS 4

HANDLE hGlobalWriteEvent;

void CreateEventsAndThreads(void)
{
    HANDLE hReadEvents[NUMTHREADS], hThread;
```

```
DWORD i, IDThread;

// Create a manual-reset event object. The master thread sets
// this to nonsignaled when it writes to the shared buffer.

hGlobalWriteEvent = CreateEvent(
    NULL,         // no security attributes
    TRUE,         // manual-reset event
    TRUE,         // initial state is signaled
    "WriteEvent"  // object name
    );

if (hGlobalWriteEvent == NULL)
{
    // error exit
}

// Create multiple threads and an auto-reset event object
// for each thread. Each thread sets its event object to
// signaled when it is not reading from the shared buffer.

for(i = 1; i <= NUMTHREADS; i++)
{
    // Create the auto-reset event.
    hReadEvents[i] = CreateEvent(
        NULL,     // no security attributes
        FALSE,    // auto-reset event
        TRUE,     // initial state is signaled
        NULL);    // object not named

    if (hReadEvents[i] == NULL)
    {
        // Error exit.
    }

    hThread = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE) ThreadFunction,
        &hReadEvents[i],  // pass event handle
        0, &IDThread);
```

```
        if (hThread == NULL)
        {
            // Error exit.
        }
    }
}
```

マススレッドは書き込みを行なう前に, ResetEvent 関数で, hGlobalWriteEvent を非シグナル状態にします. これは, 読み込みスレッドが読み込みを開始するのをブロックします.

マススレッドは, 全ての読み込みスレッドが今現在行なっている読み込みが終了するのを, WaitForMultipleObjects 関数を用いて待ちます. WaitForMultipleObjects の制御が返ったら, マススレッドは, 安全にバッファに書き込むことができます. 書き込み終了後, hGlobalWriteEvent と読み込みスレッドのイベントをシグナル状態にします. これにより, 読み込みスレッドは, 読み込みを再開することができます.

```
VOID WriteToBuffer(VOID)
{
    DWORD dwWaitResult, i;

    // Reset hGlobalWriteEvent to nonsignaled, to block readers.

    if (! ResetEvent(hGlobalWriteEvent) )
    {
        // Error exit.
    }

    // Wait for all reading threads to finish reading.

    dwWaitResult = WaitForMultipleObjects(
        NUMTHREADS,   // number of handles in array
        hReadEvents,  // array of read-event handles
        TRUE,         // wait until all are signaled
        INFINITE);    // indefinite wait

    switch (dwWaitResult)
    {
        // All read-event objects were signaled.
        case WAIT_OBJECT_0:
```

```
            // Write to the shared buffer.
            break;

        // An error occurred.
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitProcess(0);
    }

    // Set hGlobalWriteEvent to signaled.

    if (! SetEvent(hGlobalWriteEvent) )
    {
        // Error exit.
    }

    // Set all read events to signaled.
    for(i = 1; i <= NUMTHREADS; i++)
        if (! SetEvent(hReadEvents[i]) )
        {
            // Error exit.
        }
}
```

読み込みスレッドは，読み込みを開始する前に，hGlobalWriteEvent と自分の読み込みイベントがシグナル状態になるのを，WaitForMultipleObjects 関数を用いて待ちます．WaitForMultipleObjects 関数の制御が返ったら，読み込みスレッドの自動リセットオブジェクトは，非シグナル状態に戻ります．これにより，マスタースレッドの書き込みは，読み込みスレッドが読み込みを終え SetEvent 関数を呼び出すまで，ブロックします．

```
VOID ThreadFunction(LPVOID lpParam)
{
    DWORD dwWaitResult;
    HANDLE hEvents[2];

    hEvents[0] = *(HANDLE*)lpParam;  // thread's read event
    hEvents[1] = hGlobalWriteEvent;

    dwWaitResult = WaitForMultipleObjects(
```

```
        2,            // number of handles in array
        hEvents,      // array of event handles
        TRUE,         // wait till all are signaled
        INFINITE);    // indefinite wait

    switch (dwWaitResult)
    {
        // Both event objects were signaled.
        case WAIT_OBJECT_0:
            // Read from the shared buffer.
            break;

        // An error occurred.
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitThread(0);
    }

    // Set the read event to signaled.

    if (! SetEvent(hEvents[0]) )
    {
        // Error exit.
    }
}
```

3.4.1 CreateEvent

名前付きまたは名前なしのイベントオブジェクトを作成または開きます.

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes, // セキュリティ記述子
  BOOL bManualReset,                       // リセットのタイプ
  BOOL bInitialState,                      // 初期状態
  LPCTSTR lpName                           // イベントオブジェクトの名前
);
```

- lpEventAttributes　子プロセスが，取得したハンドルを継承できるかどうかを決定する SECURITY_ATTRIBUTES 構造体へのポインタを指定します．NULL を指定すると，取得したハンドルを継承できません.

- bManualReset　手動リセットオブジェクトを作成するか，自動リセットオブジェクトを作成するかを指定します．TRUE を指定すると，手動リセットオブジェクトが作成されます．FALSE を指定すると，自動リセットオブジェクトが作成されます．
- bInitialState　イベントオブジェクトの初期状態を指定します．TRUE を指定すると，シグナル状態に設定されます．FALSE を指定すると，非シグナル状態に設定されます．
- lpName　イベントオブジェクトの名前を保持している文字列へのポインタを指定します．lpName パラメータで NULL を指定すると，名前なしのイベントオブジェクトが作成されます．
- **戻り値**　関数が成功すると，イベントオブジェクトのハンドルが返ります．指定した名前付きイベントオブジェクトが既に存在していた場合も，そのオブジェクトのハンドルが返ります．関数が失敗すると，NULL が返ります．

3.4.2 SetEvent

指定されたオブジェクトをシグナル状態に設定します．

```
BOOL SetEvent(
  HANDLE hEvent   // イベントオブジェクトのハンドル
);
```

- hEvent　イベントオブジェクトのハンドルを指定します．
- **戻り値**　関数が成功すると，0 以外の値が返ります．関数が失敗すると，0 が返ります．

3.4.3 ResetEvent

指定されたイベントオブジェクトを非シグナル状態に設定します．

```
BOOL ResetEvent(
  HANDLE hEvent   // イベントオブジェクトのハンドル
);
```

- hEvent　イベントオブジェクトのハンドルを指定します．
- **戻り値** 関数が成功すると，0 以外の値が返ります．関数が失敗すると，0 が返ります．

3.4.4 PulseEvent

指定されたイベントオブジェクトをシグナル状態に設定し，待機スレッドが存在する場合は適切な数のスレッドを解放し，その後でイベントオブジェクトを非シグナル状態に戻します.

```
BOOL PulseEvent(
  HANDLE hEvent   // イベントオブジェクトのハンドル
);
```

- hEvent　イベントオブジェクトのハンドルを指定します.
- **戻り値**　関数が成功すると，0 以外の値が返ります．関数が失敗すると，0 が返ります.

3.5 クリティカルセクション

クリティカルセクションは，一度に一つのスレッドしか共有メモリにアクセス出来ないようにします．1 つのプロセス内にある複数のスレッド間で，相互排他の同期を行えます.

- クリティカルセクションオブジェクトの型は CRITICL_SECTION です.
- InitializeCriticalSection 関数は，クリティカルセクションを初期化します.
- 保護対象のリソースにアクセスする任意のコードセクションを実行する前に，EnterCriticalSection 関数を呼び出します.
- 保護対象のリソースにアクセスするコードを実行し終わると，スレッドは LeaveCriticalSection 関数を使って所有権を放棄します.
- DeleteCriticalSection 関数で，リソースを解放します.

以下はクリティカルセクションの使用例です.

```
CRITICAL_SECTION CriticalSection;

void main()
{
    ...

    // Initialize the critical section one time only.
    if (!InitializeCriticalSection(&CriticalSection)
```

```
        return;
    ...

    // Release resources used by the critical section object.
    DeleteCriticalSection(&CriticalSection)
}

DWORD WINAPI ThreadProc( LPVOID lpParameter )
{
    ...

    // Request ownership of the critical section.
    EnterCriticalSection(&CriticalSection);

    // Access the shared resource.

    // Release ownership of the critical section.
    LeaveCriticalSection(&CriticalSection);

    ...
}
```

3.5.1 InitializeCriticalSection

指定されたクリティカルセクションオブジェクトを初期化します.

```
VOID InitializeCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection  // クリティカルセクション
);
```

- lpCriticalSection クリティカルセクションオブジェクトへのポインタを指定します.

3.5.2 EnterCriticalSection

指定されたクリティカルセクションオブジェクトの所有権を取得するまで待機します. 呼び出し側スレッドが所有権を取得すると, この関数は制御を返します.

```
VOID EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection  // クリティカルセクション
);
```

- lpCriticalSection クリティカルセクションオブジェクトへのポインタを指定します.

3.5.3 LeaveCriticalSection

指定されたクリティカルセクションオブジェクトの所有権を解放します.

```
VOID LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection   // クリティカルセクション
);
```

- lpCriticalSection クリティカルセクションオブジェクトへのポインタを指定します.

3.5.4 DeleteCriticalSection

所有されていないクリティカルセクションオブジェクトを指定し，そのオブジェクトが使っているすべてのリソースを解放します.

```
VOID DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection   // クリティカルセクション
);
```

- lpCriticalSection クリティカルセクションオブジェクトへのポインタを指定します.

3.6 ミューテックス

ミューテックスは，プロセス全体で使える点を除き，クリティカルセクションと非常に良く似ています.

- CreateMutex 関数は，ミューテックスオブジェクトを作成します.
- ReleaseSemaphore 関数は，指定されたミューテックスオブジェクトの所有権を解放します.

- WaitForSingleObject 関数は，ミューッテックスの所有権を取得するまで待機します.
- CloseHandle 関数は，ハンドルを閉じリソースを解放します.

ミューテックスは，どのスレッドにも所有されていない場合に，シグナル状態とします．ミューテックスは，スレッドに所有されている場合に，非シグナル状態とします.

ミューテックスの所有権を要求するには，待機関数（WaitForSingleObject など）を使わなければなりません．非シグナル状態の時，待機関数を呼び出したスレッドはブロックします．その状態で，ミューテックスがシグナル状態になると，待機スレッドの 1 つに所有権が割り当てられます．所有権の割り当てられたミューテックスは非シグナル状態へ変化し，その待機関数は制御を返します．特定のミューテックスを所有できるスレッドは，一度に 1 つだけです．所有権を解放するには，ReleaseMutex 関数を使います.

以下はサンプルコードです．CreateMutex 関数によってミューテックスを生成します.

```
HANDLE hMutex;

// Create a mutex with no initial owner.

hMutex = CreateMutex(
    NULL,                       // no security attributes
    FALSE,                      // initially not owned
    "MutexToProtectDatabase");  // name of mutex

if (hMutex == NULL)
{
    // Check for error.
}
```

共有リソースであるデータベースにアクセスする前に，ミューテックスの所有権を取得します．ミューテックスの所有後，データベースに書き込みを行い，所有権を解放します.

```
BOOL FunctionToWriteToDatabase(HANDLE hMutex)
{
    DWORD dwWaitResult;

    // Request ownership of mutex.
```

```
    dwWaitResult = WaitForSingleObject(
        hMutex,   // handle to mutex
        5000L);   // five-second time-out interval

    switch (dwWaitResult)
    {
        // The thread got mutex ownership.
        case WAIT_OBJECT_0:
            __try {
                // Write to the database.
            }

            __finally {
                // Release ownership of the mutex object.
                if (! ReleaseMutex(hMutex))
                {
                    // Deal with error.
                }

                break;
            }

        // Cannot get mutex ownership due to time-out.
        case WAIT_TIMEOUT:
            return FALSE;

        // Got ownership of the abandoned mutex object.
        case WAIT_ABANDONED:
            return FALSE;
    }

    return TRUE;
}
```

3.6.1 CreateMutex

名前付きまたは名前なしのミューテックス（mutually exclusive；相互排他）オブジェクトを作成または開きます.

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,  // セキュリティ記述子
  BOOL bInitialOwner,                       // 最初の所有者
  LPCTSTR lpName                            // オブジェクトの名前
);
```

- lpMutexAttributes　子プロセスが，取得したハンドルを継承できるかどうかを決定する SECURITY_ATTRIBUTES 構造体へのポインタを指定します．NULL を指定すると，子プロセスはそのハンドルを継承できません．
- bInitialOwner　ミューテックスオブジェクトの初期の所有者を指定します．TRUE を指定し，呼び出し側がミューテックスを作成していた場合，呼び出し側スレッドはそのミューテックスオブジェクトの所有権を取得します．FALSE を指定すると，呼び出し側スレッドはそのミューテックスの所有権を取得しません．
- lpName　ミューテックスオブジェクトの名前を保持している文字列へのポインタを指定します．NULL を指定すると，名前なしのミューテックスが作成されます．
- **戻り値** 関数が成功すると，ミューテックスオブジェクトのハンドルが返ります．この関数を呼び出す以前にそのミューテックスオブジェクトが存在していた場合は，この関数は既存のオブジェクトに対するハンドルに返します．それ以外の場合，呼び出し側は指定されたミューテックスを作成します．関数が失敗すると，NULL が返ります．

3.6.2 ReleaseMutex

指定されたミューテックスオブジェクトの所有権を解放します．

```
BOOL ReleaseMutex(
  HANDLE hMutex   // ミューテックスオブジェクトのハンドル
);
```

- hMutex　ミューテックスオブジェクトのハンドルを指定します．
- **戻り値** 関数が成功すると，0 以外の値が返ります．関数が失敗すると，0 が返ります．

3.7 セマフォ

セマフォは，リソースを使うスレッドの数を制限するために用いられます．

セマフォはリソースカウンタを持ちます．セマフォによって制御されているリソースにアクセスすると，リソースカウンタは 1 つ減ります．カウンタが 0 の場合は，リソースは無くなっている（定員オーバ）なので，これ以降のリソースアクセス要求はブロックします．スレッドは，リソースを使い終えた時点でセマフォを解放し，それに応じてリソースカウンタは増えます．

- CreateSemaphore 関数は，セマフォを作成し，そのリソースカウントの初期値を設定します．
- ReleaseSemaphore 関数は，セマフォを解放し，そのリソースカウントを増やします．
- WaitForSingleObject 関数は，セマフォ（または任意のオブジェクト）を待機します．
- CloseHandle 関数は，ハンドルを閉じ，リソースを解放します．

セマフォオブジェクトは，カウントが 0 より大きい場合は，シグナル状態とします．カウントが 0 の場合は非シグナル状態とします．

リソースにアクセスするには，待機関数（WaitForSingleObject など）を使わなければいけません．非シグナル状態の時，待機関数を呼び出したスレッドはブロックします．その状態で，セマフォオブジェクトがシグナル状態になると，その待機関数は制御を返し，セマフォのカウントが 1 減ります．

リソースへのアクセスが終了したら，そのスレッドは ReleaseSemaphore 関数を呼び出して，セマフォのカウントを 1 増やします．

以下はサンプルコードです．CreateSemaphore 関数によって，セマフォオブジェクトを作成します．

```
HANDLE hSemaphore;
LONG cMax = 10;
LONG cPreviousCount;

// Create a semaphore with initial and max. counts of 10.

hSemaphore = CreateSemaphore(
    NULL,   // no security attributes
    cMax,   // initial count
    cMax,   // maximum count
    NULL);  // unnamed semaphore
```

```
if (hSemaphore == NULL)
{
    // Check for error.
}
```

WaitForSingleObject 関数によって，セマフォの現在のカウンタによって，共有リソースへのアクセス（ウィンドウの作成）を許可するかどうか決めます．

```
DWORD dwWaitResult;

// Try to enter the semaphore gate.

dwWaitResult = WaitForSingleObject(
        hSemaphore,   // handle to semaphore
        0L);          // zero-second time-out interval

switch (dwWaitResult)
{
    // The semaphore object was signaled.
    case WAIT_OBJECT_0:
        // OK to open another window.
        break;

    // Semaphore was nonsignaled, so a time-out occurred.
    case WAIT_TIMEOUT:
        // Cannot open another window.
        break;
}
```

スレッドはウィンドウを閉じる時，ReleaseSemaphore 関数を呼びカウンタを 1 つ増やします．

```
// Increment the count of the semaphore.

if (!ReleaseSemaphore(
        hSemaphore,  // handle to semaphore
        1,           // increase count by one
        NULL) )      // not interested in previous count
{
    // Deal with the error.
```

```
}
```

3.7.1 CreateSemaphore

名前付きまたは名前なしのセマフォオブジェクトを作成または開きます.

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // セキュリティ記述子
  LONG lInitialCount,                          // 初期のカウント
  LONG lMaximumCount,                          // 最大カウント
  LPCTSTR lpName                               // オブジェクトの名前
);
```

- lpSemaphoreAttributes 子プロセスが，取得したハンドルを継承できるかどうかを決定する SECURITY_ATTRIBUTES 構造体へのポインタを指定します．NULL を指定すると，子プロセスはそのハンドルを継承できません.
- lInitialCount セマフォオブジェクトの初期のカウントを指定します．0 以上，lMaximumCount 以下の値を指定します.
- lMaximumCount セマフォオブジェクトの最大カウントを指定します．0 より大きい値を指定しなければなりません.
- lpName セマフォオブジェクトの名前を保持している文字列へのポインタを指定します．lpName パラメータで NULL を指定すると，名前なしのセマフォオブジェクトが作成されます.
- **戻り値** 関数が成功すると，セマフォオブジェクトのハンドルが返ります．指定したセマフォオブジェクトが既に存在していた場合も，そのオブジェクトのハンドルが返ります．関数が失敗すると，NULL が返ります.

3.7.2 ReleaseSemaphore

指定されたセマフォオブジェクトのカウントを，指定された数だけ増やします.

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,        // セマフォのハンドル
  LONG lReleaseCount,       // カウントを増やすべき数
```

```
  LPLONG lpPreviousCount    // それまでのカウント
);
```

- hSemaphore　セマフォオブジェクトのハンドルを指定します.
- lReleaseCount　セマフォオブジェクトの現在のカウントの増分（増やすべき量）を指定します. 0 より大きい値を指定しなければなりません. この増分を加算すると，セマフォの作成時に指定した最大カウントを超えてしまう場合は，この関数はカウントを変更せず，FALSE を返します.
- lpPreviousCount　1 個の変数へのポインタを指定します. 関数から制御が返ると，この変数に関数を呼び出す前のカウントが格納されます. この情報が不要な場合，NULL を指定します.
- **戻り値**　関数が成功すると，0 以外の値が返ります. 関数が失敗すると，0 が返ります.

3.8　同期簡単にアクセスを行なうための API

複数のスレッドが共有する変数に対しての同期アクセスを簡単に行うための関数が提供されています. 変数の値を更新する際に，一度に一つのスレッドしかその更新を行わないことを保証します.

3.8.1　InterlockedIncrement

指定された変数の値をインクリメントし（1 つ増やします），その結果得られた値をチェックします.

```
LONG InterlockedIncrement(
  LPLONG lpAddend    // インクリメントするべき変数
);
```

- lpAddend　インクリメントされる変数へのポインタを指定します.
- **戻り値**　インクリメント後の値が返ります.

3.8.2　InterlockedDecrement

指定された変数の値をデクリメントし（1 つ減らします），その結果得られた値をチェックします.

```
LONG InterlockedDecrement(
  LPLONG lpAddend   // 変数へのポインタ
);
```

- lpAddend　デクリメントされる変数へのポインタを指定します.
- 戻り値　デクリメント後の値が返ります.

3.8.3 InterlockedExchange

指定された変数の内容と，もうひとつの値の交換を一括して行います.

```
LONG InterlockedExchange(
  LPLONG Target, // 交換に使われる変数
  LONG Value     // 新しい値
);
```

- Target　値を交換するべき変数へのポインタを指定します．変数から制御が返ると，この変数に，Value パラメータで指定した値が格納されます.
- Value　Target パラメータが指す変数に格納するべき，新しい値を指定します.
- **戻り値**　Target パラメータが指す変数の，交換前の値が返ります.

3.8.4 InterlockedExchangeAdd

加数変数への増分値の原子加算を実行します.

```
LONG InterlockedExchangeAdd (
  PLONG Addend,  // 加数へのポインタ
  LONG Increment // 増分値
);
```

- Addend　加数が入った変数へのポインタを指定します．この値に Increment が加算されます.
- Increment　Addend が指す変数に加算する数を指定します.
- **戻り値**　Addend が指す変数の加算前の値が返ります.

3.9 実行時間計測

QueryPerformanceFrequency 関数を用いて，実行時間の計測を行うプログラムを書くことができます．まず QueryPerformanceFrequency 関数について説明し，その後それを用いた時間計測プログラムについて述べます．

3.9.1 QueryPerformanceFrequency

高分解能パフォーマンスカウンタが存在する場合，そのカウンタの周波数（更新頻度）を取得します．システムが動作している間は，周波数を変更できません．

```
BOOL QueryPerformanceFrequency(
  LARGE_INTEGER *lpFrequency    // 現在の周波数
);
```

- lpFrequency　変数へのポインタを指定します．

  関数から制御が返ると，この変数に高分解能パフォーマンスカウンタの周波数が格納されます．周波数は，1 秒あたりのカウント数として表現されます．インストール先のハードウェアが高分解能パフォーマンスカウンタをサポートしていない場合，この変数に 0 が格納されることがあります．

- **戻り値**　ハードウェアが高分解能パフォーマンスカウンタをサポートしている場合は，0 以外の値が返ります．関数が失敗すると，0 が返ります．

3.9.2 実行時間を秒単位で計算する関数

QueryPerformanceFrequency 関数を用いて，マシーン起動時からの時間を秒単位で計算する関数を書くことができます．

```
#include <assert.h>
#include <windows.h>

double
TokGetTime(
    VOID
    )
{
    LARGE_INTEGER curCount, frequency;
```

```
    double period, retval;

    if (TRUE == QueryPerformanceFrequency(&frequency))
    {
        // Figure out how many seconds elapse / count
        period = 1.0 / frequency.QuadPart;
        QueryPerformanceCounter(&curCount);
        retval = curCount.QuadPart * period;
    } else
    {
        // All the systems we are dealing with should support
        // querying the performance counter.
        assert(FALSE);
    }
    return (retval);
}
```

4 Experiment 3: Threads and Fibers

この章では，ファイバについて説明します.

4.1 ファイバ

以下の 2 種類の方法でファイバを作成・実行することができます.

- ConvertThreadFiber 関数を呼び出す.
- CreateFiber 関数を呼び出す．この関数だけではファイバは実行を開始しません．実行を開始させるには SwitchToFiber 関数を呼び出す必要があります.

スレッドが SwitchToFiber 関数を使ってファイバをスケジューリングするためには，事前に ConvertThreadToFiber 関数を呼び出し，そのスレッドに関連付けられたファイバが存在するようにしておく必要があります.

ファイバを削除するには，DeleteFiber 関数を呼び出します.

以下はサンプルコードです.

```
LPVOID pFiber[2];
LPVOID pMainFiber;
```

```
VOID WINAPI DoFiber(DWORD fiberCount)
{
    if (fiberCount == 0) {
        SwitchToFiber(pFiber[1]);
    } else {
        SwitchToFiber(pMainFiber);
    }
}

DWORD WINAPI DoMainFiber( LPVOID lpParam )
{
    pMainFiber = ConvertThreadToFiber(NULL);

    SwitchToFiber(pFiber[0]);

    return 0;
}

VOID main( VOID )
{
    HANDLE hThread;
    int i;

    for(i = 0; i < 2; i++) {
        pFiber[i] = CreateFiber(
            0,
            (LPFIBER_START_ROUTINE)DoFiber,
            (LPVOID)i);
    }

    hThread = CreateThread(
        NULL,
        0,
        (LPTHREAD_START_ROUTINE)DoMainFiber,
        NULL,
        0,
        NULL);

    // wait until the thread exits
```

```
    WaitForSingleObject(hThread, INFINITE);

    CloseHandle(hThread);
}
```

4.1.1 CreateFiber

ファイバオブジェクトを確保し，そのオブジェクトにスタックを割り当て，指定された開始アドレス (通常はファイバ関数) から実行を開始するための準備を行います．この関数は，ファイバをスケジューリングしません．

```
LPVOID CreateFiber(
  DWORD dwStackSize,    // スレッドの初期のスタックサイズ（バイト数）
  LPFIBER_START_ROUTINE lpStartAddress,
                          // ファイバ関数へのポインタ
  LPVOID lpParameter    // 新しいファイバの引数
);
```

- dwStackSize　新しいファイバに割り当てるスタックのサイズをバイト単位で指定します．0 を指定すると，スタックは既定値でメインスレッドと同じサイズになります．dwStackSize で指定したサイズをコミットできないと，関数は失敗します．

- lpStartAddress　ファイバによって実行する LPFIBER_START_ROUTINE 型のアプリケーション定義関数へのポインタを指定します．新しく作成したファイバの実行は，このアドレスを指定して SwitchToFiber 関数を呼び出すまで開始されません．

  LPFIBER_START_ROUTINE 型は，

  ```
  VOID CALLBACK FiberProc(PVOID lpParameter);
  ```

  となっています．

- lpParameter　ファイバに渡す引数を 1 つだけ指定します．

- **戻り値**　関数が成功すると，ファイバのアドレスが返ります．関数が失敗すると，NULL が返ります．

4.1.2 ConvertThreadToFiber

現在のスレッドをファイバに変換します.

```
LPVOID ConvertThreadToFiber(
  LPVOID lpParameter  // 新しいファイバのファイバデータ
);
```

- `lpParameter` ファイバに渡す引数を 1 つだけ指定します.
- **戻り値** 関数が成功すると，ファイバのアドレスが返ります．関数が失敗すると，`NULL` が返ります.

4.1.3 SwitchToFiber

ファイバをスケジューリングします.

```
VOID SwitchToFiber(
  LPVOID lpFiber  // 切り替え先ファイバへのポインタ
);
```

- `lpFiber` 切り替え先ファイバのアドレスを指定します.

次の呼び出しは避けてください.

```
SwitchToFiber(GetCurrentFiber());
```

この操作を行うと，予期しないトラブルが発生します.

4.1.4 DeleteFiber

既存のファイバを削除します.

```
VOID DeleteFiber(
  LPVOID lpFiber   // 削除対象のファイバへのポインタ
);
```

- `lpFiber` 削除するファイバのアドレスを指定します.

5 Experiment 4: Non-blocking Data Structures

この章では，Interlocked Singly Linked Lists (SLists) について説明します.

5.1 Interlocked Singly Linked Lists

Interlocked Singly Linked Lists (SLists) によって，linked list への挿入と削除を簡単に行なえます．non-blocking アルゴリズムを用いて，リストへのアクセスの同期をとります．

- SList のヘッダ型は，SLIST_HEADER です．SList のエントリの型は，SLIST_ENTRY です．
- InitializeSListHead 関数によって SList を初期化します．
- InterlockedPushEntrySList 関数によって，SList に項目を挿入します．
- InterlockedPopEntrySList 関数によって，SList から項目を削除します．

以下はサンプルプログラムです．InitializeSListHead 関数によって，SList を初期化し，InterlockedPushEntrySList 関数によって，項目を 10 個挿入します．InterlockedPopEntrySList 関数によって，項目を 10 個取り除きます．そして，InterlockedFlushSList 関数によって，リストが空であることを確認します．

```
#include <windows.h>
#include <malloc.h>

// Structure to be used for a list item. Typically, the first member
// is of type SLIST_ENTRY. Additional members are used for data.
// Here, the data is simply a signature for testing purposes.

typedef struct _PROGRAM_ITEM {
    SLIST_ENTRY ItemEntry;
    ULONG Signature;
} PROGRAM_ITEM, *PPROGRAM_ITEM;

void main( )
{
    ULONG Count;
    PSLIST_ENTRY FirstEntry, ListEntry;
    SLIST_HEADER ListHead;
    PPROGRAM_ITEM ProgramItem;

    // Initialize the list header.
```

```
    InitializeSListHead(&ListHead);

    // Insert 10 items into the list.
    for( Count = 1; Count <= 10; Count += 1 )
    {
        ProgramItem = (PPROGRAM_ITEM)malloc(sizeof(*ProgramItem));
        ProgramItem->Signature = Count;
        FirstEntry = InterlockedPushEntrySList(&ListHead,
                        &ProgramItem->ItemEntry);
    }

    // Remove 10 items from the list.
    for( Count = 10; Count >= 1; Count -= 1 )
    //

    {
        ListEntry = InterlockedPopEntrySList(&ListHead);
        free(ListEntry);
    } Flush the list and verify that the items are gone.
    ListEntry = InterlockedFlushSList(&ListHead);
    FirstEntry = InterlockedPopEntrySList(&ListHead);
    if (FirstEntry != NULL)
    {
        printf("Error: List is not empty.");
    }
}
```

5.1.1 InitializeSListHead

SList のヘッダを初期化します.

```
void InitializeSListHead(
  PSLIST_HEADER ListHead
);
```

- `ListHead` SList のヘッダを指す `SLIST_HEADER` 構造体へのポインタ.

5.1.2 InterlockedPushEntrySList

SList の先頭に項目を追加します．このリストへのアクセスは同期がとられます．

```
PSLIST_ENTRY InterlockedPushEntrySList(
  PSLIST_HEADER ListHead,
  PSLIST_ENTRY ListEntry
);
```

- `ListHead`　SList のヘッダを指す `SLIST_HEADER` 構造体へのポインタ．
- `ListEntry`　Push する `SLIST_ENTRY` 構造体へのポインタ．
- **返り値**　関数実行前における，リストの先頭の項目を返します．もし間実行前にリストが空であったら，`NULL` を返します．

5.1.3 InterlockedPopEntrySList

SList の先頭から項目を取り出します．このリストへのアクセスは同期がとられます．

```
PSLIST_ENTRY InterlockedPopEntrySList(
  PSLIST_HEADER ListHead
);
```

- `ListHead`　SList のヘッダを指す `SLIST_HEADER` 構造体へのポインタ．
- **返り値**　リストから取り除かれた項目へのポインタを返します．もしリストが空であれば，`NULL` を返します．