Средства синхронизации потоков Win32 API

Критические секции

КС используются для реализации взаимных исключений для параллельных потоков, выполняемых в контексте одного процесса. КС не являются объектами ядра.

Инициализация КС:

```
VOID InitializeCriticalSection(
     LPCRITICAL_SECTION lpCriticalSection   // адрес объекта КС
);
```

Вход в КС:

```
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Попытка входа в КС (Windows 2000):

```
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Возвращает **TRUE** если поток вошел в КС или уже находится в ней и **FALSE** в противном случае.

Выход из КС:

```
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Разрушение объекта КС:

```
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Объекты синхронизации

В ОС Windows ***объектами синхронизации*** наз. объекты ядра, кот. могут находиться в одном из двух состояний: ***сигнальном*** и ***несигнальном***. Они могут быть разбиты на 4 класса.
Собственно объекты синхронизации:

- мютекс (mutex);
- событие (event);
- семафор (semaphore).

Второй класс представляет ***ожидающий таймер***, кот. переходит в сигнальное состояние по истечению заданного интервала времени.

К третьему классу относятся объекты, переходящие в сигнальное состояние по завершению работы:

- задание (job);
- процесс (process);
- поток (thread).

К четвертому классу относятся объекты, переходящие в сигнальное состояние после получения сообщения об изменении их содержимого:

- изменение состояния каталога (change notification);
- консольный ввод (console input).

Далее рассмотрим собственно объекты синхронизации: мьютексы, семафоры и события.

Функции ожидания

В ОС Windows средства синхронизации потоков не снабжаются специфичными средствами захвата объекта синхронизации. Вместо этого для всех объектов синхронизации используются единые механизмы ожидания их перехода в сигнальное состояние.

Для ожидания перехода в сигнальное состояние одного объекта синхронизации используется функция **`WaitForSingleObject`**:

```
DWORD WaitForSingleObject(
    HANDLE hHandle,           // handle of object to wait for
    DWORD dwMilliseconds      // time-out interval in milliseconds
);
```

Если параметр **`dwMilliseconds`** установлен в 0, функция только проверяет состояние объекта синхронизации, если его значение установить в **`INFINITE`** - ф-ция ждет перехода объекта в сигнальное состояние бесконечно долго.
При успешном завершении возвращается одно из значений:

- **`WAIT_OBJECT_0`** –объект перешел в сигнальное состояние;
- **`WAIT_ABANDONED`** –забытый мютекс;
- **`WAIT_TIMEOUT`** – истекло время ожидания.

В случае неудачи возвращается значение **`WAIT_FAILED`**.

Для ожидания перехода в сигнальное состояние нескольких объектов синхронизации используется функция **`WaitForMultipleObjects`**:

```
DWORD WaitForMultipleObjects(
    DWORD nCount,   // number of handles in the object handle array
    CONST HANDLE *lpHandles,  // pointer to the object-handle array
    BOOL bWaitAll,            // wait flag
    DWORD dwMilliseconds      // time-out interval in milliseconds
);
```

Если значение параметра **`bWaitAll`** равно **`TRUE`**, эта ф-ция в течение интервала **`dwMilliseconds`** ждет пока все объекты, дескрипторы которых заданы в массиве **`lpHandles[nCount]`** перейдут в сигнальное состояние.
Такой объект синхронизации еще называют ***барьером*** (barrier).
Если параметр **`bWaitAll`** равен **`FALSE`**, эта ф-ция в течение интервала **`dwMilliseconds`** ждет пока любой из заданных объектов синхронизации перейдет в сигнальное состояние.

Мьютексы

Мьютексы используются для реализации взаимных исключений для параллельных потоков, выполняемых в контекстах разных процессов. Мьютексы являются объектами ядра ОС с обслуживанием ожидающих их потоков в порядке FIFO.
Создается мьютекс вызовом ф-ции:

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,// security attributes
    BOOL bInitialOwner,  // flag for initial ownership
    LPCTSTR lpName       // pointer to mutex-object name
);
```

Для того чтобы получить доступ к уже созданному (другим потоком) мьютексу, поток может использовать ф-цию

```
HANDLE OpenMutex(
    DWORD dwDesiredAccess,    // access flag
    BOOL bInheritHandle,      // inherit flag
    LPCTSTR lpName            // pointer to mutex-object name
);
```

Мьютекс захватывается вызовом любой ф-ции ожидания, а освобождается ф-цией

```
BOOL ReleaseMutex(
    HANDLE hMutex        // handle of mutex object
);
```

Возвращает **TRUE** в случае удачного завершения и **FALSE** в противном случае. Для уточнения информации об ошибке используйте ф-цию **GetLastError**.

Для уничтожения объекта мьютекса используется функция закрытия дескриптора **CloseHandle**.

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

Возвращает **TRUE** в случае удачного завершения и **FALSE** в противном случае. Для уточнения информации об ошибке используйте ф-цию **GetLastError**.

Это общий метод уничтожения объектов в ОС Windows.

Семафоры

В ОС Windows реализованы семафоры-счетчики являющиеся объектами ядра. Потоки, ждущие сигнального состояния семафора, обслуживаются в порядке FIFO («строгие» семафоры). Однако если поток ждет наступления асинхронного события, то ф-ции ядра могут исключить поток из очереди к семафору для обслуживания наступления этого события. После этого поток становится в конец очереди семафора.

Создаются семафоры вызовом ф-ции

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // security attributes
  LONG lInitialCount,    // initial count
  LONG lMaximumCount,    // maximum count
  LPCTSTR lpName         // pointer to semaphore-object name
);
```

Здесь **lInitialCount** -начальное значение семафора (не меньше **0**), а **lMaximumCount** -его максимально возможное значение. Параметр **lpName** может указывать на имя семафора или содержать **NULL**, тогда создается безымянный семафор.

Доступ к существующему семафору можно открыть используя ф-цию **OpenSemaphore**:

```
HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,    // access flag
    BOOL bInheritHandle,      // inherit flag
    LPCTSTR lpName            // pointer to semaphore-object name
);
```

Значение семафора уменьшается на 1 при вызове функции ожидания. Увеличивают его вызовом ф-ции **ReleaseSemaphore**:

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,        // handle of the semaphore object
    LONG lReleaseCount,       // amount to add to current count
    LPLONG lpPreviousCount    // address of previous count
);
```

Возвращает **TRUE** в случае удачного завершения и **FALSE** в противном случае. Для уточнения информации об ошибке используйте ф-цию **GetLastError**.

Для уничтожения объекта семафора используется функция закрытия дескриптора **CloseHandle**.

События

Событием называется оповещение о некотором выполненном действии. Сама задача оповещения одного потока о некотором действии, выполненном другим потоком, наз. ***задачей условной синхронизации***. В ОС Windows события описываются объектами ядра Events и имеются двух типов:

- события с ручным сбросом;
- события с автоматическим сбросом.

Событие с ручным сбросом можно перевести в несигнальное состояние только вызовом ф-ции **ResetEvent**, а событие с автоматическим сбросом, кроме того, и при помощи ф-ции ожидания. Если события с автоматическим сбросом ждут несколько потоков, используя вызов **WaitForSingleObject**, то из состояния ожидания освобождается только один из потоков.

Создаются события вызовом

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,// security attributes
    BOOL bManualReset,             // flag for manual-reset event
    BOOL bInitialState,            // flag for initial state
    LPCTSTR lpName                 // pointer to event-object name
);
```

Если значение параметра **bManualReset** равно **TRUE** –создается событие с ручным сбросом, а при **FALSE** - с автоматическим сбросом. Если значение параметра **bInitialState** равно **TRUE** –начальное состояние события является сигнальным, в противном случае – несигнальным. Параметр **lpName** может указывать на имя события или содержать **NULL**, тогда создается безымянное событие.

Доступ к существующему событию можно получить, используя ф-цию **OpenEvent**:

```
HANDLE OpenEvent(
    DWORD dwDesiredAccess,    // access flag
    BOOL bInheritHandle,      // inherit flag
    LPCTSTR lpName            // pointer to event-object name
);
```

Для перевода события в сигнальное состояние используется функция

```
BOOL SetEvent(HANDLE hEvent);
```

Возвращает **TRUE** в случае удачного завершения и **FALSE** в противном случае.

Для перевода события в несигнальное состояние используется функция

BOOL ResetEvent(HANDLE hEvent);

Возвращает **TRUE** в случае удачного завершения и **FALSE** в противном случае.

Для освобождения потоков, ждущих сигнального состояния события с ручным сбросом, используется ф-ция **PulseEvent**:

BOOL PulseEvent(HANDLE hEvent);

При вызове этой ф-ции соответствующее событие переводится в сигнальное состояние, все потоки, ожидающие его, выводятся из состояния ожидания, а само событие сразу переводится в несигнальное состояние.
Если эта ф-ция вызывается для события с автоматическим сбросом, то из состояния ожидания выводится только один из ожидающих потоков.
Если нет потоков, ожидающих сигнального состояния события с заданным дескриптором, состояние этого события остается несигнальным.

Для уничтожения объекта события используется функция закрытия дескриптора **CloseHandle**.

Примеры

1. Использование семафоров и критической секции для решения задачи читателей-писателей.

```
/*                    ЗАДАЧА ЧИТАТЕЛЕЙ - ПИСАТЕЛЕЙ
Имеется РЕСУРС (данные в файле, блоке памяти и т.п.), совместно
используемый рядом процессов. Одни процессы могут только читать из
РЕСУРСа (читатели), другие - только записывать в РЕСУРС (писатели).
Любое число читателей могут одновременно читать РЕСУРС.
Записывать в РЕСУРС одномоментно может только один писатель.
Если писатель пишет в РЕСУРС, ни один читатель не может его читать.

Решение справедливо для любого числа процессов читателей и писателей.
Реализация для Dev-C++ (uses Mingw port of GCC) версия 4.9.9.2.
Решение с приоритетом по чтению: */

#include <windows.h>
#include <iostream>

using namespace std;

#define NR 10                                   // число читателей
#define NW 5                                    // число писателей
#define MAX 10                        // каждый поток обращается MAX раз

volatile int resource = 200;             // это и есть наш РЕСУРС
volatile int readcount = 0; // инициализ. счетчик активных читателей

CHAR lpRCsemName[] = "readcountLOCK";   // имя семафора для readcount
CHAR lpREsemName[] = "resourceLOCK";    // имя семафора для РЕСУРСа

CRITICAL_SECTION ConsoleCS;                     // для защиты консоли
```

```
void ReadRes(int iNum){                  // процедура чтения из РЕСУРС
   EnterCriticalSection(&ConsoleCS);
   cout << resource << " from reader " << iNum << endl;
   LeaveCriticalSection(&ConsoleCS);
}

void WriteRes(int iNum){                 // процедура записи в РЕСУРС
   resource = resource - (iNum + 1);
   EnterCriticalSection(&ConsoleCS);
   cout << resource << " FROM WRITER " << iNum << endl;
   LeaveCriticalSection(&ConsoleCS);
}

DWORD WINAPI reader(LPVOID iNum) {          // процесс читатель

   HANDLE hRCsem, hREsem;                   // дескрипторы семафоров

         // открываем семафоры:
   hRCsem = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, lpRCsemName);
   hREsem = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, lpREsemName);

   for(int i=0; i<MAX; i++) {           // абонемент на MAX посещений
     WaitForSingleObject(hRCsem, INFINITE);  // начало 1 КР счетчика
     readcount++;
     if(readcount == 1)           // если это единственный читатель
                    // установим блокировку на РЕСУРС для писателей:
          WaitForSingleObject(hREsem, INFINITE);
     ReleaseSemaphore(hRCsem, 1, NULL);     // конец 1 КР счетчика
     ReadRes((int)iNum);                    // ЧИТАЕМ ...
     WaitForSingleObject(hRCsem, INFINITE); // начало 2 КР счетчика
     readcount--;
     if(readcount == 0)           // если это был последний читатель
          ReleaseSemaphore(hREsem, 1, NULL);// снимаем блок. с РЕСУРС
     ReleaseSemaphore(hRCsem, 1, NULL);     // конец 2 КР счетчика
     Sleep(500*((int)iNum+1)); //перерыв для обдумывания у всех разный
   }
   CloseHandle(hRCsem);           // закрываем дескрипторы семафоров
   CloseHandle(hREsem);
}

DWORD WINAPI writer(LPVOID iNum) {          // процесс писатель

   HANDLE hREsem;                           // дескриптор семафора

                  // открываем семафор:
   hREsem = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, lpREsemName);

   for(int i=0; i<MAX; i++) {           // абонемент на MAX посещений
     WaitForSingleObject(hREsem, INFINITE); // начало КР РЕСУРС
     WriteRes((int)iNum);                   // ПИШЕМ ...
     ReleaseSemaphore(hREsem, 1, NULL);     // конец КР РЕСУРС
     Sleep(1000*((int)iNum+1));  // творческий перерыв у всех разный
   }
   CloseHandle(hREsem);            // закрываем дескриптор семафора
}
```

```
int main(void){

   int n;
   HANDLE hReader[NR], // дескрипторы потоков читателей
          hWriter[NW]; // и писателей

   HANDLE hRCsem, hREsem;          // дескрипторы семафоров
                         // создаем семафоры:
   hRCsem = CreateSemaphore(NULL, 1, 1, lpRCsemName);
   hREsem = CreateSemaphore(NULL, 1, 1, lpREsemName);

   InitializeCriticalSection(&ConsoleCS); //создаем критическую секцию

   for(n = 0;n < NR; n++)          // стартуем потоки-читатели
      hReader[n] = CreateThread(NULL, 0, reader, (void*)n, 0, NULL);

   for(n = 0;n < NW; n++)          // стартуем потоки-писатели
      hWriter[n] = CreateThread(NULL, 0, writer, (void*)n, 0, NULL);

               // ждем завершения потоков-читателей:
   WaitForMultipleObjects((DWORD)NR, hReader, TRUE, INFINITE);
   for(n = 0;n < NR; n++)  // закрываем дескрипторы потоков-читателей
      CloseHandle(hReader[n]);

               // ждем завершения потоков-писателей:
   WaitForMultipleObjects((DWORD)NW, hWriter, TRUE, INFINITE);
   for(n = 0;n < NW; n++)  // закрываем дескрипторы потоков-писателей
      CloseHandle(hWriter[n]);

   CloseHandle(hRCsem);            // закрываем дескрипторы семафоров
   CloseHandle(hREsem);

   DeleteCriticalSection(&ConsoleCS); //ликвидируем критическую секцию

   system("PAUSE");
   return EXIT_SUCCESS;
}
```