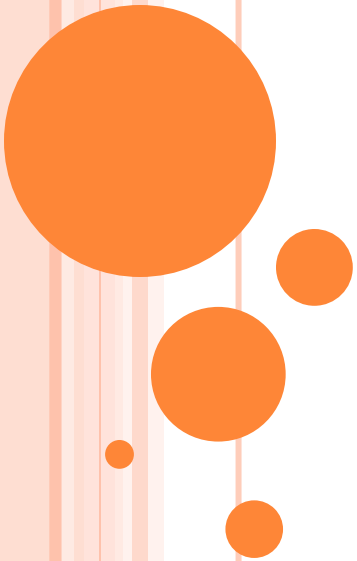

# Синхронизация потоков в Windows

## Критические секции

- Для работы с объектами типа **CRITICAL_SECTION** используются следующие функции:

```
// инициализация критической секции
VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// вход в критическую секцию
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// попытка войти в критическую секцию
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// выход из критической секции
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
// разрушение объекта критическая секция
VOID
```

```
#include <windows.h>
#include <iostream>
using namespace std;
DWORD WINAPI thread(LPVOID) {
int i, j ;
for (j = 0; j < 10; ++j) {
// выводим строку чисел j
for (i = 0; i < 10; ++i)
{
cout « j « ' ' « flush;
Sleep(17);
}
cout « endl;
return 0;
}
```

```
int main() {
int i, j ;
HANDLE hThread; DWORD IDThread;
hThread=CreateThread(NULL, 0, thread, NULL, 0,
   &IDThread) ;
if (hThread == NULL) return GetLastError();
for (j = 10; j < 20; ++j) {
for (i = 0; i < 10; ++i) {
// выводим строку чисел j
cout « j « ‘ ‘ « flush;
Sleep(17);
}
cout « endl;
}
// ждем, пока поток thread закончит свою работу
WaitForSingleObject(hThread, INFINITE);
return 0;
}
```

```
#include <windows.h> #include <iostream>
using namespace std;
CRITICAL_SECTION cs;
DWORD WINAPI thread(LPVOID)
{
int i,j;
for (j = 0; j < 10; ++j)
{
// входим в критическую секцию
EnterCriticalSection (&cs);
for (i = 0; i < 10; ++i)
{
cout « j « ' '« flush;
Sleep(7);
}
cout « endl;
// выходим из критической секции
LeaveCriticalSection(&cs);
}
return 0;
}
```

```
int main() {
int i,j;
HANDLE hThread; DWORD IDThread;
// инициализируем критическую секцию
InitializeCriticalSection(&cs);
hThread=CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
if (hThread == NULL) return GetLastError();
for (j = 10; j < 20; ++j) {
// входим в критическую секцию
EnterCriticalSection(&cs);
for (i = 0; i < 10; ++i) {
cout « j « ' ' « flush; Sleep(7) ;
}
cout « endl;
// выходим из критической секции
LeaveCriticalSection(&cs);
}
// ждем, пока поток thread закончит свою работу
WaitForSingleObject(hThread, INFINITE);
// закрываем критическую секцию
DeleteCriticalSection(&cs);
return 0;
}
```

## Объекты синхронизации и функции ожидания

- В операционных системах Windows *объектами синхронизации* называются объекты ядра, которые могут находиться в одном из двух состояний: *сигнальном* (signaled) и *несигнальном* (nonsignaled). Объекты синхронизации могут быть разбиты на четыре класса.

- К первому классу относятся объекты синхронизации, т. е. те, которые служат только для решения задач синхронизации параллельных потоков:
  - мьютекс (mutex);
  - событие (event);
  - семафор (semaphore).
- Ко второму классу объектов синхронизации относится *ожидающий таймер* (waitable timer), который переходит в сигнальное состояние по истечении заданного интервала времени.
- К третьему классу синхронизации относятся объекты, которые переходят в сигнальное состояние по завершении своей работы:
  - работа (job);
  - процесс (process);
  - поток (thread).
- К четвертому классу относятся объекты синхронизации, которые переходят в сигнальное состояние после получения сообщения об изменении содержимого объекта. К ним относятся:
  - изменение состояния каталога (change notification);
  - консольный ввод (console input).

- *Функции ожидания* в Windows это такие функции, параметрами которых являются объекты синхронизации. Эти функции обычно используются для блокировки потоков.

- Для ожидания перехода в сигнальное состояние одного объекта синхронизации используется функция **WaitForSingieObject,** которая имеет следующий прототип:

```
DWORD WaitForSingleObject(
HANDLE hHandle,        // дескриптор объекта
DWORD dwMilliseconds // интервал ожидания в миллисекундах
) ;
```

- Для ожидания перехода в сигнальное состояние нескольких объектов синхронизации или одного из нескольких объектов синхронизации используется функция **WaitForMuitipieObject,** которая имеет следующий прототип:

```
DWORD WaitForMultipleObjects(
DWORD nCount,            // количество объектов
CONST HANDLE *lpHandles,    // массив дескрипторов
                            объектов
BOOL bWaitAll,           // режим ожидания
DWORD dwMilliseconds        // интервал ожидания в
                               миллисекундах
) ;
```

# Мьютексы

- Для решения проблемы взаимного исключения между параллельными потоками, выполняющимися в контекстах разных процессов, в операционных системах Windows используется объект ядра *мьютекс*. Слово мьютекс происходит от английского слова mutex, которое в свою очередь является сокращением от выражения mutual exclusion, что на русском языке значит "взаимное исключение". Мьютекс находится в сигнальном состоянии, если он не принадлежит ни одному потоку. В противном случае мьютекс находится в несигнальном состоянии. Одновременно мьютекс может принадлежать только одному потоку.

- Создается мьютекс вызовом функции **CreateMutex,** которая имеет следующий прототип:

```
HANDLE CreateMutex(
LPSECURITY_ATTRIBUTES lpMutexAttributes,
// атрибуты защиты
BOOL blnitialOwner,
// начальный владелец мьютекса
LPCTSTR lpName
// имя мьютекса
);
```

```
#include <windows.h>
#include <iostream>
HANDLE hMutex;
char IpszAppName [] = "C:\\ConsoleProcess.exe";
STARTUPINFO si;
PROCESS_INFORMATION pi;
// создаем мьютекс
hMutex = CreateMutex(NULL, FALSE,
   "DemoMutex");
if (hMutex == NULL)
{
cout « "Create mutex failed." « endl;
cout « "Press any key to exit." « endl;
cin.get();
return GetLastError();
}
```

```
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
// создаем новый консольный процесс
if (!CreateProcess(IpszAppName, NULL, NULL, NULL,
    FALSE, NULL, NULL, NULL, &si, &pi))
{
cout « "The new process is not created." « endl;
cout « "Press any key to exit." « endl;
cin.get();
return GetLastError();
}
// выводим на экран строки
for (int j = 0; j < 10; ++j) {
// захватываем мьютекс
WaitForSingleObject(hMutex, INFINITE);
for (int i = 0; i < 10; i++) {
cout « j « ' ' « flush; Sleep(10);
```

```
}
cout « endl; // освобождаем мьютекс
ReleaseMutex (hMutex) ;
}
// закрываем дескриптор мьютекса
CloseHandle(hMutex);
// ждем пока дочерний процесс закончит работу
WaitForSingleObject (pi .hProcess, INFINITE) ;
// закрываем дескрипторы дочернего процесса в
текущем процессе
CloseHandle (pi .hThread) ;
CloseHandle(pi.hProcess);
return 0;
}
```

```
#include <windows.h>
#include <iostream>
int main() {
HANDLE hMutex;
 int i , j ;
// открываем мьютекс
hMutex = OpenMutex(SYNCHRONIZE,
   FALSE,"DemoMutex");
if (hMutex == NULL) {
cout « "Open mutex failed." « endl;
cout « "Press any key to exit." « endl;
cin.get();
return GetLastError();
for (j = 10; j < 20; j++) {
```

```
// захватываем мьютекс
WaitForSingleObject(hMutex, INFINITE);
for (i = 0; i < 10; i++) {
cout « j « ' ' « flush;
Sleep(5);
}
cout « endl; // освобождаем мьютекс
ReleaseMutex(hMutex);
}
// закрываем дескриптор объекта
 CloseHandle(hMutex);
return 0;
}
```

# События

- *Событием* называется оповещение о некотором выполненном действии. В программировании события используются для оповещения одного потока о том, что другой поток выполнил некоторое действие. Сама же задача оповещения одного потока о некотором действии, которое совершил другой поток, называется *задачей условной синхронизации*. В операционных системах Windows события описываются объектами ядра Events.
- При этом различают два типа событий:
  - ✔ события с ручным сбросом;
  - ✔ события с автоматическим сбросом.

- Создаются события вызовом функции **CreateEvent,** которая имеет следующий прототип:

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES
   lpSecurityAttributes,
// атрибуты защиты
BOOL bManualReset,
// тип события
BOOL blnitialState,
// начальное состояние события
LPCTSTR lpName
// имя события
) ;
```

- Для перевода любого события в сигнальное состояние используется функция **SetEvent,** которая имеет следующий прототип:

```
BOOL SetEvent(
HANDLE hEvent // дескриптор события
) ;
```

```
#include <windows.h>
 #include <iostream>
HANDLE hOutEvent, hAddEvent;
DWORD WINAPI thread (LPVOID) {
for (int i = 0; i < 10; ++i)
if (i == 4) {
SetEvent(hOutEvent);
WaitForSingleObject(hAddEvent, INFINITE);
}
return 0;
}
int main() {
HANDLE hThread; DWORD IDThread;
```

```
// создаем события с автоматическим сбросом
hOutEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
 if (hOutEvent == NULL)
return GetLastError();
hAddEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hAddEvent == NULL) return GetLastError();
// создаем поток thread
hThread = CreateThread(NULL, 0, thread, NULL, 0,
   &IDThread);
if (hThread == NULL)
return GetLastError();
// ждем, пока поток thread выполнит половину работы
WaitForSingleObject(hOutEvent, INFINITE);
```

```
// выводим значение переменной
cout « "A half of the work is done." « endl;
cout « "Press any key to continue." « endl;
cin.get();
// разрешаем дальше работать потоку thread
SetEvent (hAddEvent) ;
WaitForSingleObject(hThread, INFINITE);
CloseHandle(hThread);
CloseHandle(hOutEvent);
CloseHandle(hAddEvent);
cout « "The work is done." « endl;
return 0;
}
```

- Доступ к существующему событию можно открыть с помощью функции

**CreateEvent** или **OpenEvent.**

```
HANDLE OpenEvent(
DWORD dwDesiredAccess, // флаги доступа
BOOL blnheritHandle,     // режим наследования
LPCTSTR lpName           // имя события
) ;
```

- Параметр **dwDesiredAccess** определяет доступ к событию и может быть равен любой логической комбинации следующих флагов:
- ✔ **EVENT_ALL_ACCESS** — полный доступ;
- ✔ **EVENT_MODIFY_STATE** — модификация состояния;
- ✔ **SYNCHRONIZE** — синхронизация.
- Эти флаги устанавливают следующие режимы доступа к событию:
- ✔ флаг **EVENT_ALL_ACCESS** означает, что поток может выполнять над событием любые действия;
- ✔ флаг **EVENT_MODIFY_STATE** означает, что поток может использовать функции **SetEvent** и **ResetEvent** для изменения состояния события;
- ✔ флаг **SYNCHRONIZE** означает, что поток может использовать событие в функциях ожидания.

```
#include <windows.h>
#include <iostream>
HANDLE hlnEvent;
char IpEventName[ ] = "InEventName";
int main() {
DWORD dwWaitResult;
char szAppName[] = "D:\\ConsoleProcess.exe";
STARTUPINFO si;
PROCESS_INFORMATION pi;
// создем событие, отмечающее ввод символа
hlnEvent = CreateEvent(NULL, FALSE, FALSE,
IpEventName);
if (hlnEvent == NULL)
return GetLastError();
```

```
// запускаем процесс, который ждет ввод символа
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
if (!CreateProcess(szAppName, NULL, NULL, NULL, FALSE,
CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
return 0;
// закрываем дескрипторы этого процесса
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
// ждем оповещение о наступлении события о вводе символа
dwWaitResult = WaitForSingieObject(hlnEvent, INFINITE);
if (dwWaitResult != WAIT_OBJECT_0) return dwWaitResult;
cout « "A symbol has got. " « endl;
CloseHandie (hlnEvent) ;
cout « "Press any key to exit.";
cin.get();
return 0;
}
```

```
#include <windows.h>
#include <iostream>
HANDLE hlnEvent;
CHAR lpEventName[]="InEventName" ;
int main () {
char c;
hlnEvent = OpenEvent (EVENT_MODIFY_STATE,
FALSE, lpEventName);
if (hlnEvent == NULL) {
cout « "Open event failed." « endl;
cout « "Input any char to exit." « endl;
 cin.get();
return GetLastError();
}
```

```
cout « "Input any char: ";
cin » c;
// устанавливаем событие о вводе символа
SetEvent(hlnEvent);
// закрываем дескриптор события в текущем процессе
CloseHandie(hlnEvent);
cin.get();
cout << "Press any key to exit." << endl;
cin.get();
return 0;
}
```

## Семафоры

- Семафоры в операционных системах Windows описываются объектами ядра **semaphores.** Семафор находится в сигнальном состоянии, если его значение больше нуля. В противном случае семафор находится в несигнальном состоянии.
- Создаются семафоры посредством вызова функции **CreateSemaphore,** которая имеет следующий прототип:

```
HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES lpSemaphoreAttribute,
// атрибуты защиты
LONG llnitialCount,
// начальное значение семафора
LONG IMaximumCount,
// максимальное значение семафора
LPCTSTR lpName
// имя семафора
);
```

```
BOOL ReleaseSemaphore
(
HANDLE hSemaphore,// дескриптор семафора

LONG IReleaseCount,  // положительное число, на
   которое увеличивается значение семафора

LPLONG lpPreviousCount // предыдущее значение
   семафора
) ;
```

```
HANDLE OpenSemaphore(
DWORD dwDesiredAccess,  // флаги доступа
BOOLblnheritHandle,   // режим наследования
LPCTSTR lpName              // имя события
);
```

- Параметр **dwDesiredAccess** определяет доступ к семафору и может быть равен любой логической комбинации следующих флагов:
- ✔ **SEMAPHORE_ALL_ACCESS** — полный доступ к семафору;
- ✔ **SEMAPHORE_MODIFY_STATE** — изменение состояния семафора;
- ✔ **SYNCHRONIZE** — **синхронизация.**

```
#include <windows.h>
#include <iostream>
volatile int a[10];
HANDLE hSemaphore;
DWORD WINAPI thread(LPVOID)
{
for (int i = 0; i < 10; i++)
{
a [i] = i + 1;
// отмечаем, что один элемент готов
ReleaseSemaphore(hSemaphore,1,NULL);
Sleep(500) ;
}
return 0;
}
```

```
int main()
{
int i;
HANDLE hThread;
DWORD IDThread;
cout « "An initial state of the array: ";
for (i = 0; i < 10; i++)
cout « a[i] «' ';
cout « endl;
// создаем семафор
hSemaphore=CreateSemaphore(NULL, 0, 10, NULL);
if (hSemaphore == NULL) return GetLastError();
// создаем поток, который готовит элементы массива
hThread = CreateThread(NULL, 0, thread, NULL, 0,
&IDThread);
if (hThread == NULL) return GetLastError();
```

```
// поток main выводит элементы массива только после их
   подготовки потоком thread
cout « "A final state of the array: ";
for (i = 0; i < 10; i++)
{
WaitForSingleObject(hSemaphore, INFINITE);
cout « a[i] « " \a" « flush;
}
cout « endl;
CloseHandle(hSemaphore);
CloseHandle(hThread);
return 0;
}
```