**Лабораторная работа № 4**

**Тема: «Синхронизация процессов при помощи событий и мьютексов».**

**Объекты синхронизации и функции ожидания в Windows.**

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 три класса. К первому классу относятся объекты синхронизации, которые служат только для решения проблемы синхронизации параллельных потоков. К таким объектам синхронизации в Windows относятся:

* мьютекс (mutex);
* событие (event);
* семафор (semaphore).

Ко второму классу объектов синхронизации относится ожидающий таймер (waitable timer). К третьему классу объектов синхронизации относятся объекты, которые переходят в сигнальное состояние по завершении своей работы или при получении некоторого сообщения. Примерами таких объектов синхронизации являются потоки и процессы. Пока эти объекты выполняются, они находятся в несигнальном состоянии. Если выполнение этих объектов заканчивается, то они переходят в сигнальное состояние.

Теперь перейдем к функциям ожидания. Функции ожидания в Windows это такие функции, параметрами которых являются объекты синхронизации. Эти функции обычно используются для блокировки потоков, которая выполняется следующим образом. Если дескриптор объекта синхронизации является параметром функции ожидания, а сам объект синхронизации находится в несигнальном состоянии, то поток, вызвавший эту функцию ожидания, блокируется до перехода этого объекта синхронизации в сигнальное состояние. Сейчас мы будем использовать только две функции ожидания WaitForSingleObject и WaitForMultipleObject.

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

DWORD WaitForSingleObject(

HANDLE hHandle, // дескриптор объекта

DWORD dwMilliseconds // интервал ожидания в миллисекундах

);

Функция WaitForSingleObject в течение интервала времени, равного значению параметра dwMilliseconds, ждет пока объект синхронизации с дескриптором hHandle перейдет в сигнальное состояние. Если значение параметра dwMilliseconds равно нулю, то функция только проверяет состояние объекта. Если же значение параметра dwMilliseconds равно INFINITE, то функция ждет перехода объекта синхронизации в сигнальное состояние бесконечно долго.

В случае удачного завершения функция WaitForSingleObject возвращает одно из следующих значений:

WAIT\_OBJECT\_0

WAIT\_ABANDONED

WAIT\_TIMEOUT

Значение WAIT\_OBJECT\_0 означает, что объект синхронизации находился или перешел в сигнальное состояние. Значение WAIT\_ABANDONED означает, что объектом синхронизации является мьютекс , который не был освобожден потоком, завершившим свое исполнение. После завершения потока этот мьютекс освободился системой и перешел в сигнальное состояние. Такой мьютекс иногда называется забытым мьютексом (abandoned mutex). Значение WAIT\_TIMEOUT означает , что время ожидания истекло, а объект синхронизации не перешел в сигнальное состояние. В случае неудачи функция WaitForSingleObject возвращает значение WAIT\_FAILED.

Приведем пример простой программы, которая использует функцию WaitForSingleObject для ожидания завершения потока. Отметим также, что эта функция уже использовалась нами в Программе 2.1 для ожидания завершения работы потока Add.

Программа 4.1.

// Пример использования функциеи WaitForSingleObject

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

using namespace std;

void thread()

{

int i;

for (i = 0; i < 10 ; i++)

{

cout << i << ' ';

cout << flush << '\a';

Sleep(500);

}

cout << endl;

}

int main()

{

HANDLE hThread; DWORD dwThread;

hThread = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)thread, NULL, 0, &dwThread);

if (hThread == NULL)

return GetLastError();

// ждем завершения потока thread

if(WaitForSingleObject(hThread, INFINITE) != WAIT\_OBJECT\_0)

{

cout << "Wait for single object failed." << endl;

cout << "Press any key to exit." << endl;

}

// закрываем дескриптор потока thread

CloseHandle(hThread);

return 0;

}

Для ожидания перехода в сигнальное состояние нескольких объектов синхронизации или одного из нескольких объектов синхронизации используется функция WaitForMultipleObject, 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

);

Функция WaitForMultipleObjects работает следующим образом. Если значение параметра bWaitAll равно TRUE, то эта функция в течение интервала времени, равного значению параметра dwMilliseconds, ждет пока все объекты синхронизации, дескрипторы которых заданы в массиве lpHandles, перейдут в сигнальное состояние. Если же значение параметра bWaitAll равно FALSE, то эта функция в течение заданного интервала времени ждет пока любой из заданных объектов синхронизации перейдет в сигнальное состояние. Если значение параметра dwMilliseconds равно нулю, то функция только проверяет состояние объектов синхронизации. Если же значение параметра dwMilliseconds равно INFINITE, то функция ждет перехода объектов синхронизации в сигнальное состояние бесконечно долго. Количество объектов синхронизации, ожидаемых функцией WaitForMultipleObjects, не должно превышать значения MAXIMUM\_WAIT\_OBJECTS. Также отметим, что объекты синхронизации не должны повторяться.

В случае успешного завершения функция WaitForMultipleObjects возвращает их следующих значений:

от WAIT\_OBJECT\_0 до (WAIT\_OBJECT\_0 + nCount – 1);

от WAIT\_ABANDONED\_0 до (WAIT\_ABANDONED\_0 + nCount – 1); WAIT\_TIMEOUT.

Интерпретация значений, возвращаемых функцией WaitForMultipleObjects, зависит от значения входного параметра bWaitAll. Сначала рассмотрим случай, когда значение этого параметра равно TRUE. Тогда возвращаемые значения интерпретируются следующим образом:

− любое из возвращаемых значений, находящихся в диапазоне от WAIT\_OBJECT\_0 до (WAIT\_OBJECT\_0 + nCount – 1), означает, что все объекты синхронизации находились или перешли в сигнальное состояние;

− любое из возвращаемых значений, находящихся в диапазоне от WAIT\_ABANDONED\_0 до (WAIT\_ABANDONED\_0 + nCount – 1) означает, что все объекты синхронизации находились или перешли в сигнальное состояние и, по крайней мере, один их них был забытым мьютексом;

− возвращаемое значение WAIT\_TIMEOUT означает, что время ожидания истекло и не все объекты синхронизации перешли в сигнальное состояние.

Теперь рассмотрим случай, когда значение входного параметра bWaitAll равно FALSE. В этом случае значения, возвращаемые функцией WaitForMultipleObjects, интерпретируются следующим образом:

− любое из возвращаемых значений, находящихся в диапазоне от WAIT\_OBJECT\_0 до (WAIT\_OBJECT\_0 + nCount – 1), означает, что, по крайней мере, один из объектов синхронизации находился или перешёл в сигнальное состояние. Индекс дескриптора этого объекта в массиве определяется как разница между возвращаемым значением и величиной WAIT\_OBJECT\_0;

− любое из возвращаемых значений, находящихся в диапазоне от WAIT\_ABANDONED\_0 до (WAIT\_ABANDONED\_0 + nCount – 1) означает, что одним из объектов синхронизации, перешедшим в сигнальное состояние, является забытый мьютекс. Индекс дескриптора этого мьютекса в массиве определяется как разница между возвращаемым значением и величиной WAIT\_OBJECT\_0;

− возвращаемое значение WAIT\_TIMEOUT означает, что время ожидания истекло, и ни один из объектов синхронизации не перешел в сигнальное состояние.

В случае неудачи функция WaitForMultipleObjects возвращает значение WAIT\_FAILED.

Приведем пример программы, которая использует функцию WaitForSingleObject для ожидания завершения двух потоков.

Программа 4.2.

// Пример использования функциеи WaitForMultipleObjects

#include <windows.h>

#include <iostream>

using namespace std;

void thread\_0()

{

int i;

for (i = 0; i < 5 ; i++)

{

cout << i << ' ';

cout << flush << '\a';

Sleep(500);

}

cout << endl;

}

void thread\_1()

{

int i;

for (i = 5; i < 10 ; i++)

{

cout << i << ' ';

cout << flush << '\a';

Sleep(500);

}

cout << endl;

}

int main()

{

HANDLE hThread[2]; DWORD dwThread[2];

// запускаем первый поток

hThread[0] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)thread\_0, NULL, 0, &dwThread[0]);

if (hThread[0] == NULL)

return GetLastError();

// запускаем второй поток

hThread[1] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)thread\_1, NULL, 0, &dwThread[1]);

if (hThread[1] == NULL)

return GetLastError();

// ждем завершения потоков thread\_1 и thread\_2

if (WaitForMultipleObjects(2, hThread, TRUE, INFINITE) == WAIT\_FAILED)

{

cout << "Wait for multiple objects failed." << endl;

cout << "Press any key to exit." << endl;

}

// закрываем дескрипторы потоков thread\_0 и thread\_1 CloseHandle(hThread[0]);

CloseHandle(hThread[1]);

return 0;

}

**Проблема взаимного исключения.**

Любой ресурс, на доступ к которому претендуют не менее двух параллельных потоков, называется критическим или разделяемым ресурсом. Участок программы, на протяжении которого поток ведет работу с критическим ресурсом, называется критической секцией по отношению к этому ресурсу. Например, рассмотрим два параллельных потока:

Поток 1. Поток 2.

void thread\_1( ) void thread\_2( )

{ {

. .

. .

. .

if (n%2 == 0) ++n;

n = a; .

else .

n = b; .

. }

}

Возможно, что после проверки условия (n%2 == 0) работа первого потока прервется , и процессорное время будет передано второму потоку. Второй поток увеличит значение переменной n на единицу и после этого процессор опять будет передан первому потоку. В этом случае первый поток присвоит переменной n неправильное значение. Для исключения такой ситуации, необходимо блокировать одновременный доступ потоков к переменной n. Следовательно, в этом примере переменная n или, более точно, область памяти, занимаемая этой переменной, является критическим ресурсом, а рассматриваемые участки программного кода являются критическими секциями по отношению к этому ресурсу. Для правильной работы потоков thread\_1 и thread\_2 необходимо обеспечить , чтобы приведенные участки программного кода не могли работать одновременно. Другими словами нам необходимо решить задачу исключения взаимного доступа потоков thread\_1 и thread\_2 к критическому ресурсу, которым является переменная n.

В общем случае проблема взаимного исключения формулируется следующим образом. Необходимо обеспечить такую работу параллельных потоков с критическим ресурсом, при которой гарантируется, что критические секции этих потоков по отношению к этому ресурсу не работают одновременно.

**Мьютексы в Windows.**

Для решения проблемы взаимного исключения между параллельными потоками, выполняющимися в контексте разных процессов, в операционных системах Windows используется объект ядра мьютекс. Слово мьютекс является переводом 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 bInitialOwner, // начальный владелец мьютекса

LPCTSTR lpName // имя мьютекса

);

Пока значение параметра LPSECURITY \_ATTRIBUTES будем устанавливать в NULL. Это означает, что атрибуты защиты заданы по умолчанию, то есть дескриптор мьютекса не наследуется и доступ к мьютексу имеют все пользователи. Теперь перейдем к другим параметрам.

Если значение параметра bInitialOwner равно TRUE, то мьютекс сразу переходит во владение потоку, которым он был создан. В противном случае вновь созданный мьютекс свободен. Поток, создавший мьютекс, имеет все права доступа к этому мьютексу.

Значение параметра lpName определяет уникальное имя мьютекса для всех процессов, выполняющихся под управлением операционной системы. Это имя позволяет обращаться к мьютексу из других процессов, запущенных под управлением этой же операционной системы. Длина имени не должна превышать значение MAX\_PATH. Значением параметра lpName может быть пустой указатель NULL. В этом случае система создает безымянный мьютекс. Отметим также, что имена мьютексов являются чувствительными к нижнему и верхнему регистрам.

В случае удачного завершения функция CreateMutex возвращает дескриптор созданного мьютекса. В случае неудачи эта функция возвращает значение NULL. Если мьютекс с заданным именем уже существует, то функция CreateMutex возвращает дескриптор этого мьютекса, а функция GetLastError, вызванная после функции CreateMutex вернет значение ERROR\_ALREADY\_EXISTS.

Мьютекс захватывается потоком посредством любой функции ожидания, а освобождается функцией ReleaseMutex, которая имеет следующий прототип:

BOOL ReleaseMutex(

HANDLE hMutex // дескриптор мьютекса

);

В случае успешного завершения функция ReleaseMutex возвращает значение TRUE, в случае неудачи – FALSE. Если поток освобождает мьютекс, которым он не владеет, то функция ReleaseMutex возвращает значение FALSE.

Для доступа к существующему мьютексу поток может использовать одну из функций CreateMutex или OpenMutex. Функция CreateMutex используется в тех случаях, когда поток не знает, создан или нет мьютекс с указанным именем другим потоком. В этом случае значение параметра bInitialOwner нужно установить в FALSE, так как невозможно определить какой из потоков создает мьютекс. Если поток использует для доступа к уже созданному мьютексу функцию CreateMutex, то он получает полный доступ к этому мьютексу. Для того чтобы получить доступ к уже созданному мьютексу, поток может также использовать функцию OpenMutex, которая имеет следующий прототип:

HANDLE OpenMutex(

DWORD dwDesiredAccess, // доступ к мьютексу

BOOL bInheritHandle // свойство наследования

LPCTSTR lpName // имя мьютекса

);

Параметр dwDesiredAccess этой функции может принимать одно из двух значений:

MUTEX\_ALL\_ACCESS

SYNCHRONIZE

В первом случае поток получает полный доступ к мьютексу. Во втором случае поток может использовать мьютекс только в функциях ожидания, чтобы захватить мьютекс, или в функции ReleaseMutex, для его освобождения. Параметр bInheritHandle определяет свойство наследования мьютекса . Если значение этого параметра равно TRUE, то дескриптор открываемого мьютекса является наследуемым. В противном случае – дескриптор не наследуется.

В случае успешного завершения функция OpenMutex возвращает дескриптор открытого мьютекса, в случае неудачи эта функция возвращает значение NULL.

Покажем пример использования мьютекса для синхронизации потоков из разных процессов. Для этого сначала рассмотрим пример не синхронизированных потоков.

Программа 4.3.

// Не синхронизированные потоки, выполняющиеся в разных процессах

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

int i,j;

for (j = 10; j < 20; j++)

{

for (i = 0; i < 10; i++)

{

cout << j << ' ';

cout.flush();

Sleep(5);

}

cout << endl;

}

return 0;

}

Программа 4.4.

// Не синхронизированные потоки, выполняющиеся в разных процессах

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

char lpszAppName[] = "D:\\os.exe";

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory(&si, sizeof(STARTUPINFO));

si.cb = sizeof(STARTUPINFO);

// создаем новый консольный процесс

if (!CreateProcess(lpszAppName, 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++)

{

for (int i = 0; i < 10; i++)

{

cout << j << ' ';

cout.flush();

Sleep(10);

}

cout << endl;

}

// 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;

}

Кратко опишем работу этих программ. Вторая из них запускает первую программу, после чего потоки из разных процессов начинают выводить числа в одну консоль. Из- за отсутствия синхронизации, числа в одной строке могут быть из разных потоков. Для того чтобы избежать перемешивания чисел, синхронизируем их вывод с помощью мьютекса. Ниже приведены модификации этих программ с использованием мьютекса для синхронизации работы этих потоков.

Программа 4.5.

// Синхронизация потоков, выполняющихся в

// разных процессах, с использованинм мьютекса

#include <windows.h>

#include <iostream>

using namespace std;

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 << ' ';

cout.flush();

Sleep(5);

}

cout << endl;

// освобождаем мьютекс

ReleaseMutex(hMutex);

}

// закрываем дескриптор объекта

CloseHandle(hMutex);

return 0;

}

Программа 4.6.

// Пример синхронизации потоков, выполняющихся

// в разных процессах, с использованием мьютекса

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

HANDLE hMutex;

char lpszAppName[] = "D:\\os.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(lpszAppName, 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 << ' ';

cout.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;

}

**События в Windows.**

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 или иногда задачей оповещения.

В операционных системах Windows события описываются объектами ядра Events. При этом различают два типа событий:

− события с ручным сбросом;

− события с автоматическим сбросом.

Различие между этими типами событий заключается в том, что событие с ручным сбросом можно перевести в несигнальное состояние только посредством вызова функции ResetEvent, а событие с автоматическим сбросом переходит в несигнальное состояние как при помощи функции ResetEvent, так и при помощи функции ожидания. При этом отметим, что если события с автоматическим сбросом ждут несколько потоков, используя функцию WaitForSingleObject, то из состояния ожидания освобождается только один из этих потоков.

Создаются события вызовом функции CreateEvent, которая имеет следующий прототип:

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES lpSecurityAttributes, // атрибуты защиты

BOOL bManualReset, // тип события

BOOL bInitialState, // начальное состояние события

LPCTSTR lpName // имя события

);

Как и обычно, пока значение параметра lpSecurityAttributes будем устанавливать в NULL. Основную смысловую нагрузку в этой функции несут второй и третий параметры. Если значение параметра bManualReset равно TRUE, то создается событие с ручным сбросом, в противном случае – с автоматическим сбросом. Если значение параметра bInitialState равно TRUE, то начальное состояние события является сигнальным, в противном случае – несигнальным. Парметр lpName задает имя события, которое позволяет обращаться к нему из потоков, выполняющихся в разных процессах. Этот параметр может быть равен NULL, тогда создается безымянное событие.

В случае удачного завершения функция CreateEvent возвращает дескриптор события, а в случае неудачи – значение NULL. Если событие с заданным именем уже существует, то функция CreateEvent возвращает дескриптор этого события, а функция GetLastError, вызванная после функции CreateEvent вернет значение ERROR\_ALREADY\_EXISTS.

Ниже приведена программа, в которой безымянные события с автоматическим сбросом используются для синхронизации работы потоков, выполняющихся в одном процессе.

Программа 4.7.

// Пример синхронизации потоков при помощи

// событий с автоматическим сбросом

#include <windows.h>

#include <iostream>

using namespace std;

volatile int n;

HANDLE hOutEvent, hAddEvent;

DWORD WINAPI thread(LPVOID)

{

int i;

for (i = 0; i < 10; i++)

{

++n;

if (i == 4)

{

SetEvent(hOutEvent);

WaitForSingleObject(hAddEvent, INFINITE);

}

}

return 0;

}

int main()

{

HANDLE hThread;

DWORD IDThread;

cout << "An initial value of n = " << n << endl;

// 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 счетчик 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 << "An intermediate value of n = " << n << endl;

// разрешаем дальше работать потоку thread SetEvent(hAddEvent);

WaitForSingleObject(hThread, INFINITE);

cout << "A final value of n = " << n << endl;

CloseHandle(hThread);

CloseHandle(hOutEvent);

CloseHandle(hAddEvent);

return 0;

}

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

BOOL SetEvent(

HANDLE hEvent // дескриптор события

);

При успешном завершении эта функция возвращает значение TRUE, в случае неудачи – FALSE.

Для перевода любого события в несигнальное состояние используется функция ResetEvent, которая имеет следующий прототип:

BOOL ResetEvent(

HANDLE hEvent // дескриптор события

);

При успешном завершении эта функция возвращает значение TRUE, в случае неудачи – FALSE.

Для освобождения нескольких потоков, ждущих сигнального состояния события с ручным сбросом, используется функция PulseEvent, которая имеет следующий прототип:

BOOL PulseEvent(

HANDLE hEvent // дескриптор события

);

При вызове этой функции все потоки, ждущие события с дескриптором hEvent, выводятся из состояния ожидания, а само событие сразу переходит в несигнальное состояние. Если функция PulseEvent вызывается для события с автоматическим сбросом, то из состояния ожидания выводится только один из ожидающих потоков. Если нет потоков, ожидающих сигнального состояния события из функции PulseEvent, то состояние этого события остается несигнальным. Для выполнения этой функции требуется, чтобы в дескрипторе события был установлен режим доступа EVENT\_MODIFY\_STATE.

Ниже приведен пример программы, использующей для синхронизации события как с ручным, так и автоматическим сбросом.

Программа 4.8.

// Пример синхронизации потоков при помощи событий с ручным сбросом

#include <windows.h>

#include <iostream>

using namespace std;

volatile int n,m;

HANDLE hOutEvent[2], hAddEvent;

DWORD WINAPI thread\_1(LPVOID)

{

int i;

for (i = 0; i < 10; i++)

{

++n;

if (i == 4)

{

SetEvent(hOutEvent[0]);

WaitForSingleObject(hAddEvent, INFINITE);

}

}

return 0;

}

DWORD CALLBACK thread\_2(LPVOID)

{

int i;

for (i = 0; i < 10; i++)

{

++m;

if (i == 4)

{

SetEvent(hOutEvent[1]);

}

WaitForSingleObject(hAddEvent, INFINITE);

}

return 0;

}

int main()

{

HANDLE hThread\_1, hThread\_2;

DWORD IDThread\_1, IDThread\_2;

cout << "An initial values of n = " << n << ", m = " << m << endl;

// создаем события с автоматическим сбросом

hOutEvent[0] = CreateEvent(NULL, FALSE, FALSE, NULL);

if (hOutEvent[0] == NULL)

return GetLastError();

hOutEvent[1] = CreateEvent(NULL, FALSE, FALSE, NULL);

if (hOutEvent[1] == NULL)

return GetLastError();

// создаем событие с ручным сбросом

hAddEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (hAddEvent == NULL)

return GetLastError();

// создаем потоки счетчики

hThread\_1 = CreateThread(NULL, 0, thread\_1, NULL, 0, &IDThread\_1);

if (hThread\_1 == NULL)

return GetLastError();

hThread\_2 = CreateThread(NULL, 0, thread\_2, NULL, 0, &IDThread\_2);

if (hThread\_2 == NULL)

return GetLastError();

// ждем пока потоки счетчики выполнят половину работы

WaitForMultipleObjects(2, hOutEvent, TRUE, INFINITE);

cout << "An intermediate values of n = " << n <<", m = " << m << endl;

// разрешаем потокам счетчикам продолжать работу

SetEvent(hAddEvent);

// ждем завершения потоков

WaitForSingleObject(hThread\_1, INFINITE);

WaitForSingleObject(hThread\_2, INFINITE);

cout << "A final values of n = " << n << ", m = " << m << endl;

CloseHandle(hThread\_1);

CloseHandle(hThread\_2);

CloseHandle(hOutEvent[0]);

CloseHandle(hOutEvent[1]);

CloseHandle(hAddEvent);

return 0;

}

Доступ к существующему событию можно открыть с помощью одной из функций CreateEvent или OpenEvent. Если для этой цели используется функция CreateEvent, то значения параметров bManualReset и bInitialState этой функции игнорируются, так как они уже установлены другим потоком, а поток, вызвавший эту функцию, получает полный доступ к событию с именем, заданным параметром lpName. Теперь рассмотрим функцию OpenEvent, которая используется в случае, если известно, что событие с заданным именем уже существует. Эта функция имеет следующий прототип:

HANDLE OpenEvent(

DWORD dwDesiredAccess, // флаги доступа

BOOL bInheritHandle, // режим наследования

LPCTSTR lpName // имя события

);

Параметр dwDesiredAccess определяет доступ к событию, и может быть равен любой логической комбинации следующих флагов:

EVENT\_ALL\_ACCESS

EVENT\_MODIFY\_STATE

SYNCHRONIZE

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.

В завершение приведем пример синхронизации потоков , выполняющихся в разных процессах, при помощи события с автоматическим сбросом. В этом примере также используется функция OpenEvent для доступа к уже существующему событию.

Программа 4.9.

// Пример синхронизации потоков в разных процессах

// с использованием именованного события

#include <windows.h>

#include <iostream>

using namespace std;

HANDLE hInEvent;

CHAR lpEventName[]="InEventName";

int main()

{

char c;

hInEvent = OpenEvent(EVENT\_MODIFY\_STATE, FALSE, lpEventName);

if (hInEvent == NULL)

{

cout << "Open event failed." << endl;

cout << "Input any char to exit." << endl;

cin >> c;

return GetLastError();

}

cout << "Input any char: ";

cin >> c;

// устанавливаем событие о вводе символа

SetEvent(hInEvent);

// закрываем дескриптор события в текущем процессе

CloseHandle(hInEvent);

cout << "Now input any char to exit from the process: ";

cin >> c;

return 0;

}

Программа 4.10.

// Пример синхронизации потоков в разных процессах

// с использованием именованного события

#include <windows.h>

#include <iostream>

using namespace std;

HANDLE hInEvent;

CHAR lpEventName[] = "InEventName";

int main()

{

DWORD dwWaitResult;

char szAppName[] = "D:\\ConsoleProcess.exe";

STARTUPINFO si;

PROCESS\_INFORMATION pi;

// создаем событие, отмечающее ввод символа

hInEvent = CreateEvent(NULL, FALSE, FALSE, lpEventName);

if (hInEvent == 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 от этого процесса   
dwWaitResult = WaitForSingleObject(hInEvent, INFINITE);

if (dwWaitResult != WAIT\_OBJECT\_0)

return dwWaitResult;

cout << "A symbol has got." << endl;

CloseHandle(hInEvent);

cout << "Press any key to exit: ";

cin.get();

return 0;

}

Кратко опишем работу этих программ. Вторая из них запускает первую программу, после чего ждет, пока первая программа не введет какой-нибудь символ. После ввода символа обе программы заканчивают свою работу. Для оповещения второй программы о вводе символа используется именованное событие.