Белорусский государственный университет
Факультет прикладной математики и информатики
Кафедра технологии программирования
доцент Побегайло А.П.

4. Синхронизация потоков.

4.1. Взаимодействующие потоки.

Потоки, выполнение которых перекрывается во времени, называются *параллельными потоками*. Параллельные потоки, которые работают независимо друг от друга, называются *асинхронными потоками*. Если же параллельные потоки обмениваются некоторыми сигналами, то они называются *взаимодействующими потоками*. Сигналы, которыми обмениваются взаимодействующие потоки, могут быть двух типов: управляющие и данные. *Управляющие сигналы* предназначены для координации работы взаимодействующих потоков. *Сигналы данных* предназначены для обмена данными между параллельными потоками. Под *синхронизацией потоков* понимается обмен управляющими сигналами между параллельными потоками по установленному протоколу для обеспечения координации их действий. Под *протоколом* мы понимаем набор правил, по которым выполняется обмен этими сигналами. Если в результате синхронизации выполняется только один из взаимодействующих потоков, то такая синхронизация называется *жёсткой*. Отметим, что жесткая синхронизация эквивалентна последовательному выполнению программ и поэтому не дает выигрыша во времени при выполнении параллельных потоков на мультипроцессорных компьютерах.

В главе рассматриваются средства операционной системы Windows, предназначенные для синхронизации параллельных потоков.

4.2. 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;
}
```

4.3. 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.

```
void thread_1( )
{
        .
        .
        .
    if (n%2 == 0)
        n = a;
    else
        n = b;
        .
        .
        .
}
```

Поток 2.

```
void  thread_2( )
{
        .
        .
        .
        ++n;
        .
        .
        .
}
```

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.

4.4. Критические секции в Windows.

В операционных системах Windows проблема взаимного исключения для параллельных потоков, выполняемых в контексте одного процесса, решается при помощи объекта типа CRITICAL_SECTION, который не является объектом ядра операционной системы. Для работы с объектами этого типа используются следующие функции:

```
VOID   InitializeCriticalSection   (LPCRITICAL_SECTION  lpCriticalSection);
VOID   EnterCriticalSection        (LPCRITICAL_SECTION  lpCriticalSection);
BOOL   TryEnterCriticalSection     (LPCRITICAL_SECTION  lpCriticalSection);
VOID   LeaveCriticalSection        (LPCRITICAL_SECTION  lpCriticalSection);
VOID   DeleteCriticalSection       (LPCRITICAL_SECTION  lpCriticalSection);
```

каждая из которых имеет единственный параметр, указатель на объект типа CRITICAL_SECTION. Все эти функции, за исключением *TryEnterCriticalSection*, не возвращают значения. Отметим, что функция *TryEnterCriticalSection* поддерживается только операционной системой Windows 2000.

Кратко рассмотрим порядок работы с этими функциями. Для этого предположим, что при проектировании программы мы выделили некоторый разделяемый ресурс и критические секции в

параллельных потоках, которые имеют доступ к этому разделяемому ресурсу. Тогда для обеспечения корректной работы с этим ресурсом нужно выполнить следующую последовательность действий:

- определить в нашей программе объект типа CRITICAL_SECTION, имя которого логически связано с выделенным разделяемым ресурсом;
- проинициализировать объектом типа CRITICAL_SECTION при помощи функции *InitializeCriticalSection*;
- в каждом из параллельных потоков пред входом в критическую секцию вызвать функцию *EnterCriticalSection*, которая исключает одновременный вход в критические секции, связанные с нашим разделяемым ресурсом, для параллельно выполняющихся потоков;
- после завершения работы с разделяемым ресурсом, поток должен покинуть свою критическую секцию, что выполняется посредством вызова функции *LeaveCriticalSection*;
- после окончания работы с объектом типа CRITICAL_SECTION, необходимо освободить все системные ресурсы, которые использовались этим объектом. Для этой цели служит функция *DeleteCriticalSection*.

Теперь покажем работу этих функций на примере. Для этого сначала рассмотрим пример, в котором выполняются не синхронизированные параллельные потоки, а затем синхронизируем их работу, используя критические секции.

Программа 4.3.

```
// Пример работы не синхронизированных потоков

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

using namespace std;

DWORD WINAPI thread(LPVOID)
{
	int i,j;

	for (j = 0; j < 10; j++)
	{
		for (i = 0; i < 10; i++)
		{
			cout << j << ' ';
			cout << flush;
			Sleep(22);
		}
		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();

	// так как потоки не синхронизированы,
	// то выводимые строки непредсказуемы
```

```
	for (j = 10; j < 20; j++)
	{
		for (i = 0; i < 10; i++)
		{
			cout << j << ' ';
			cout << flush;
			Sleep(22);
		}
		cout << endl;
	}
	// ждем, пока поток thread закончит свою работу
	WaitForSingleObject(hThread, INFINITE);

	return 0;
}
```

В этой программе каждый из потоков main и thread выводит строки одинаковых чисел. Но из-за параллельной работы потоков, каждая выведенная строка может содержать не равные между собой элементы. Наша задача будет заключаться в следующем: нужно так синхронизировать потоки main и thread, чтобы в каждой строке выводились только равные между собой элементы. Следующая программа показывает решение этой задачи с помощью объекта типа CRITICAL_SECTION.

Программа 4.4.

```
// Пример работы синхронизированных потоков

#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 << ' ';
			cout.flush();
		}
		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();

	// потоки синхронизированы, поэтому каждая
	// строка содержит только одинаковые числа
	for (j = 10; j < 20; j++)
	{
		// входим в критическую секцию
		EnterCriticalSection(&cs);
		for (i = 0; i < 10; i++)
		{
			cout << j << ' ';
			cout.flush();
		}
		cout << endl;
		// выходим из критической секции
		LeaveCriticalSection(&cs);
	}
	// закрываем критическую секцию
	DeleteCriticalSection(&cs);
	// ждем, пока поток thread закончит свою работу
	WaitForSingleObject(hThread, INFINITE);

	return 0;
}
```

Теперь рассмотрим использование функции *TryEnterCriticalSection*. Для этого просто заменим в приведенной программе вызовы функции *EnterCriticalSection* на вызовы функции *TryEnterCriticalSection* и будем отмечать успешные входы потоков в свои критические секции. Еще раз подчеркнем, что функция *TryEnterCriticalSection* работает только на платформе операционной системы Windows 2000.

Прогамма 4.5.

```
// Пример работы синхронизированных потоков.
// Работает только в Windows 2000.

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

using namespace std;

CRITICAL_SECTION cs;

DWORD WINAPI thread(LPVOID)
{
	int i,j;

	for (j = 0; j < 10; j++)
	{
		// попытка войти в критическую секцию
		TryEnterCriticalSection (&cs);
		for (i = 0; i < 10; i++)
		{
			cout << j << " ";
```

```
                cout.flush();
            }
            cout << endl;
            // выход 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();

        // потоки синхронизированы, поэтому каждая
        // строка содержит только одинаковые числа
        for (j = 10; j < 20; j++)
        {
            // попытка войти в критическую секцию
            TryEnterCriticalSection(&cs);
            for (i = 0; i < 10; i++)
            {
                cout << j << " ";
                cout.flush();
            }
            cout << endl;
            // выход из критической секции
            LeaveCriticalSection(&cs);
        }
        // удаляем критическую секцию
        DeleteCriticalSection(&cs);
        // ждем завершения работы потока thread
        WaitForSingleObject(hThread, INFINITE);

        return 0;
}
```

В заключение данного параграфа отметим, что, так как объекты типа CRITICAL_SECTION не являются объектами ядра операционной системы, то работа с ними происходит несколько быстрее, чем с объектами ядра операционной системы, так как в этом случае программа меньше обращается к ядру операционной системы.

4.5. 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 4.6.

```
// 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 4.7.

```
// 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 4.8.

```
// 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 4.9.

```
// 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;
}
```

4.6. 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 4.10.

```
// 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. Однако заметим, что на платформе Windows NT/2000 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 4.11.

```
// 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 параграфа 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 4.12.

```
// 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 4.13.

```
// 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[] = "C:\\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.

4.7. Семафоры Дейкстры.

Семафор – это неотрицательная целая переменная, значение которой может изменяться только при помощи неделимых операций. Под понятием *неделимая операция* мы понимаем такую операцию, выполнение которой не может быть прервано. Семафор считается *свободным*, если его значение больше нуля, в противном случае семафор считается *занятым*. Пусть s – семафор, тогда над ним можно определить следующие неделимые операции:

```
P(s) {
	если s >0 то s = s – 1;			// поток продолжает работу
	иначе ждать освобождения s;		// поток переходит в состояние ожидания
}

V(s) {
	если потоки ждут освобождения s, то освободить один поток;
	иначе s = s + 1;
}
```

Семафор с операциями P и V называется *семафором Дейкстры*, голландского математика, который первым использовал семафоры для решения задач синхронизации. Из определения операций над семафором видно, что если поток выдает операцию P и значение семафора больше нуля, то значение семафора уменьшается на 1 и этот поток продолжает свою работу, в противном случае поток переходит в состояние ожидания до освобождения семафора другим потоком. Вывести из состояния ожидания поток, который ждет освобождения семафора, может только другой поток, который выдает операцию V над этим же семафором. Потоки, ждущие освобождения семафора, выстраиваются в очередь к этому семафору. Дисциплина обслуживания очереди зависит от конкретной реализации. Очередь может обслуживаться как по правилу FIFO, так и при помощи более сложных алгоритмов, учитывая приоритеты потоков.

Семафор, который может принимать только значения 0 или 1, называется *двоичным* или *бинарным* семафором. Чтобы подчеркнуть отличие бинарного семафора от не бинарного семафора, то есть такого семафора, значение которого может быть больше 1, последний обычно называют *считающими семафором*. Покажем, как бинарный семафор может использоваться для моделирования критических секций и событий. Для этого сначала рассмотрим следующие потоки, которые мы уже рассматривали в параграфе 4.3.

```
semaphor  s = 1;  // семафор свободен

void thread_1( )                                    void  thread_2( )
{                                                   {
                    .                                       .
                    .                                       .
                    .                                       .
          P(s);                                             P(s);
                    if (n%2 == 0)                                   n++;
                              n = a;                        V(s);
                    else                                    .
                              n = b;                        .
          V(s);                                             .
                    .                               }
                    .
                    .
}
```

Как следует из определения операций над семафором, данный подход решает проблему взаимного исключения одновременного доступа к переменной n для потоков thread_1 и thread_2. Таким образом, бинарный семафор позволяет решить проблему взаимного исключения.

Теперь предположим, что поток thread_1 должен производить проверку значения переменной n только после того, как поток thread_2 увеличит значение этой переменной. Для решения этой задачи модифицируем наши программы следующим образом:

```
semaphor  s = 0;  // семафор занят

void thread_1( )                                    void  thread_2( )
{                                                   {
                    .                                       .
                    .                                       .
                    .                                       .
          P(s);                                             n++;
                    if (n%2 == 0)                           V(s);
                              n = a;                        .
                    else                                    .
                              n = b;                        .
                    .                               }
                    .
                    .
}
```

Как видно из этих программ, бинарный семафор позволяет также решить задачу условной синхронизации.

4.7. Семафоры в Windows.

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 не сигнальном состоянии. Создаются семафоры посредством вызова функции *CreateSemaphore*, которая имеет следующий прототип:

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

Как и обычно, пока значение параметра lpSemaphoreAttributes будем устанавливать в NULL. Основную смысловую нагрузку в этой функции несут второй и третий параметры. Значение параметра lInitialCount устанавливает начальное значение семафора, которое должно быть не меньше 0 и не больше его максимального значения, которое устанавливается параметром lMaximumCount.

В случае успешного завершения функция *CreateSemaphore* возвращает дескриптор семафора, в случае неудачи – значение NULL. Если семафор с заданным именем уже существует, то функция *CreateSemaphor* возвращает дескриптор этого семафора, а функция *GetLastError*, вызванная после функции *CreateSemaphor* вернет значение ERROR_ALREADY_EXISTS.

Значение семафора уменьшается на 1 при его использовании в функции ожидания. Увеличить значение семафора можно посредством вызова функции *ReleaseSemaphore*, которая имеет следующий прототип:

```
BOOL ReleaseSemaphore(
        HANDLE      hSemaphore,         // дескриптор семафора
        LONG        lReleaseCount,      // положительное число,
                                        // на которое увеличивается значение семафора
        LPLONG      lpPreviousCount     // предыдущее значение семафора
);
```

В случае успешного завершения функция *ReleaseSemaphore* возвращает значение TRUE, в случае неудачи – FALSE. Если значение семафора плюс значение параметра lReleaseCount больше максимального значения семафора, то функция *ReleaseSemaphore* возвращает значение FALSE и значение семафора не изменяется.

Значение параметра lpPreviousCount этой функции может быть равно NULL. В этом случае предыдущее значение семафора не возвращается.

Приведем пример программы, в которой считающий семафор используется для синхронизации работы потоков. Для этого сначала рассмотрим не синхронизированный вариант этой программы.

Программа 4.14.

```
// Несинхронизированные потоки

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

using namespace std;

volatile int a[10];

DWORD WINAPI thread(LPVOID)
{
        int i;

        for (i = 0; i < 10; i++)
        {
                a[i] = i + 1;
                Sleep(17);
        }

        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 поток, который готовит элементы массива
	hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
	if (hThread == NULL)
		return GetLastError();

	// поток main выводит элементы массива
	cout << "A modified state of the array: ";
	for (i = 0; i < 10; i++)
	{
		cout << a[i] << ' ';
		cout.flush();
		Sleep(17);
	}
	cout << endl;

	CloseHandle(hThread);

	return 0;
}
```

Теперь кратко опишем работу этой программы. Поток thread последовательно присваивает элементам массива «a» значения, которые на единицу больше чем их индекс. Поток main последовательно выводит элементы массива «a» на консоль. Так как потоки thread и main не синхронизированы, то неизвестно, какое состояние массива на консоль поток main. Наша задача состоит в том, чтобы поток main выводил на консоль элементы массива «a» сразу после их подготовки потоком thread. Для этого мы используем считающий семафор. Следующая программа показывает, как этот считающий семафор используется для синхронизации работы потоков.

Программа 4.15.

```
// Пример синхронизации потоков с использованием семафора

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

using namespace std;

volatile int a[10];
HANDLE hSemaphore;

DWORD WINAPI thread(LPVOID)
{
	int i;

	for (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] << ' ';
		cout.flush();
	}
	cout << endl;

	CloseHandle(hSemaphore);
	CloseHandle(hThread);

	return 0;
}
```

Может возникнуть следующий вопрос: почему для решения этой задачи используется именно считающий семафор и почему его максимальное значение равно 10. Конечно, поставленную задачу можно было бы решить и другими способами. Но дело в том, что считающие семафоры предназначены именно для решения подобных задач. Подробнее, считающие семафоры используются для синхронизации доступа к однотипным ресурсам, которые производятся некоторым потоком или несколькими потоками, а потребляются другим потоком или несколькими потоками. В этом случае значение считающего семафора равно количеству произведенных ресурсов, а его максимальное значение устанавливается равным максимально возможному количеству таких ресурсов. При производстве единицы ресурса значение семафора увеличивается на единицу, а при потреблении единицы ресурса значение семафора уменьшается на единицу. В нашем примере ресурсами являются элементы массива, заполненные потоком thread, который является *производителем* этих ресурсов. В свою очередь поток main является *потребителем* этих ресурсов, которые он выводит на консоль. Так как в общем случае мы не можем сделать предположений о скоростях работы параллельных потоков, то максимальное значение считающего семафора должно быть установлено в максимальное количество производимых ресурсов. Если поток потребитель ресурсов работает быстрее чем поток производитель ресурсов, то, вызвав функцию ожидания считающего семафора, он вынужден будет ждать, пока поток-производитель не произведет очередной ресурс. Если же наоборот, поток-производитель работает быстрее чем поток-потребитель, то первый поток произведет все ресурсы и закончит свою работу, не ожидая, пока второй поток потребит их. Такая синхронизация потоков производителей и потребителей обеспечивает их максимально быструю работу.

Доступ к существующему семафору можно открыть с помощью одной из функций *CreateSemaphore* или *OpenSemaphore*. Если для этой цели используется функция *CreateSemaphore*, то значения параметров lInitialCount и lMaximalCount этой функции игнорируются, так как они уже установлены другим потоком, а

поток, вызвавший эту функцию, получает полный доступ к семафору с именем, заданным параметром lpName. Теперь рассмотрим функцию *OpenSemaphore*, которая используется в случае, если известно, что семафор с заданным именем уже существует. Эта функция имеет следующий прототип:

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

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

```
SEMAPHORE_ALL_ACCESS
SEMAPHORE_MODIFY_STATE
SYNCHRONIZE
```

Флаг SEMAPHORE_ALL_ACCESS устанавливает для потока полный доступ к семафору. Это означает, что поток может выполнять над семафором любые действия. Флаг SEMAPHORE_MODIFY_STATE означает, что поток может использовать только функцию *ReleaseSemaphore* для изменения значения семафора. Флаг SYNCHRONIZE означает, что поток может использовать семафор только в функциях ожидания. Отметим, что последний режим поддерживается только на платформе Windows NT/2000.