ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное образовательное учреждение
высшего профессионального образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
на тему:
«Параллельное программирование:
Win32 threads»

Ростов-на-Дону
2008

Методические указания разработаны кандидатом физико-математических наук, доцентом кафедры информатики и вычислительного эксперимента В.А. Савельевым.

Компьютерный набор и вёрстка автора.

Печатается в соответствии с решением кафедры информатики и вычислительного эксперимента факультета математики, механики и компьютерных наук (мехмата) ЮФУ, протокол № от « » 2008 года.

Процессы

Создание нового процесса

Процесс в Win32 создается вызовом функции:

```
BOOL CreateProcess(
     LPCTSTR lpszImageName,
     LPCTSTR lpszCommandLine
     LPSECURITY_ATTRIBUTES lpsaProcess,
     LPSECURITY_ATTRIBUTES lpsaThread,
     BOOL fInheritHandles,
     DWORD fdwCreate,
     LPVOID lpvEnvironment,
     LPTSTR lpszCurDir,
     LPSTARTUPINFO lpsiStartInfo,
     LPPROCESS_INFORMATION lppiProcInfo);
```

Когда вызывается CreateProcess, система создает объект ядра «процесс» со значением счетчика пользователей, равным единице. Этот объект — компактная структура данных, через которую операционная система управляет процессом и сохраняет в нём статистическую информацию о процессе. Затем система создает для нового процесса виртуальное адресное пространство размером в 4 Гб и загружает в него код и данные. После этого создается первичная нить процесса. Первичная нить начинает с исполнения стартового С-кода, вызывающего функцию WinMain для оконного приложения, или функцию main для консольного приложения. Если системе удалось создать процесс и его первичную нить, функция CreateProcess возвращает TRUE.

Кратко опишем параметры функции CreateProcess:

`lpszImageName` определяет имя исполняемого файла, который будет исполняться в новом процессе. Если параметр равен `NULL`, исполняемый файл

определяется по командной строке. Иначе можно передать строку с именем исполняемого файла — должно обязательно указанно расширение имени файла, и, если выполнимый файл не в рабочем каталоге, полный путь к выполнимому файлу.

`lpszCommandLine` определяет передаваемую процессу командную строку. Если `lpszImageName` содержит имя выполняемого файла, здесь приводятся только параметры. Если же в `lpszImageName` передается `NULL`, предполагается, что в командной строке первым элементом передается имя выполнимого файла, который ищется по следующим каталогам:

1. Каталог, содержащий выполнимый файл вызывающего процесса;
2. Текущий (рабочий) каталог вызывающего процесса;
3. Системный (`%WINDIR%\system32`) каталог Windows;
4. Основной каталог Windows (`%WINDIR%`);
5. Каталоги, перечисленные в переменной окружения `PATH`.

`lpsaProcess` атрибуты защиты для объекта «процесс». Устанавливаются по умолчанию, если передано `NULL`.

`lpsaThread` атрибуты защиты для объекта «нить». Устанавливаются по умолчанию, если передано `NULL`.

`fInheritHandles` логическое значение, определяющее, наследует ли новый процесс уже имеющиеся дескрипторы (handles) родительского процесса.

`fdwCreate` определяет флаги, влияющие на создаваемый процесс. Несколько флагов комбинируются булевским ИЛИ:

`DEBUG_PROCESS` позволяет родительскому процессу производить отладку дочернего процесса, а равно и всех процессов, которые могут быть порождены дочерним процессом;

DEBUG_ONLY_THIS_PROCESS позволяет родительскому процессу производить отладку только создаваемого дочернего процесса;

CREATE_SUSPENDED позволяет создать новый процесс и его первичную нить, но нить создается в приостановленном состоянии;

DETACHED_PROCESS отключает создаваемый процесс от созданных или унаследованных родительским процессом окон (консолей), используется, в частности, при создании системных служб;

CREATE_NEW_CONSOLE создает новую консоль для нового процесса, одновременная установка флагов CREATE_NEW_CONSOLE и DETACHED_PROCESS недопустима;

CREATE_NEW_PROCESS_GROUP служит для модификации списка процессов, уведомляемых о нажатии Ctrl+Break или Ctrl+C;

CREATE_DEFAULT_ERROR_MODE установка флага позволяет не наследовать режимы обработки ошибок у родительского процесса;

CREATE_SEPARATE_WOW_VDM позволяет запускать 16-битные приложения в отдельной виртуальной машине;

CREATE_UNICODE_ENVIRONMENT сообщает, что в переменные окружения содержат символы Unicode;

класс приоритета устанавливает приоритет нового процесса:

IDLE_PRIORITY_CLASS

NORMAL_PRIORITY_CLASS

HIGH_PRIORITY_CLASS

REALTIME_PRIORITY_CLASS

lpvEnvironment указывает на блок памяти, хранящий строки переменных окружения. Обычно передается NULL, в результате новый процесс наследует переменные окружения от родительского процесса.

`lpszCurDir` позволяет установить в дочернем процессе рабочий каталог, Если равен `NULL`, рабочий каталог будет тот же, что и родительского процесса. В путь необходимо включать и букву диска.

`lpsiStartInfo` указывает на структуру:

```
typedef struct _STARTUPINFO {
     DWORD     cb;
     LPSTR     lpReserved;
     LPSTR     lpDesktop
     LPSTR     lpTitle;
     DWORD     dwX;
     DWORD     dwY;
     DWORD     dwXSize;
     DWORD     dwYSize;
     DWORD     dwXCountChars;
     DWORD     dxYCountChars;
     DWORD     dwFillAttribute;
     DWORD     dwFlags;
     WORD      wShowWindows;
     WORD      cbReserved2;
     LPBYTE    lpReserved;
     HANDLE    hStdInput;
     HANDLE    hStdOutput;
     HANDLE    hStdError;
}    STARTUPINFO,   *LPSTARTUPINFO;
```

lppiProcInfo указывает на предварительно созданную структуру:

```
typedef struct _PROCESS_INFORMATION {
     HANDLE    hProcess;
     HANDLE    hThread;
     DWORD     dwProcessId;
     DWORD     dwThreadId;
}  PROCESS_INFORMATION,  *LPPROCESS_INFORMATION;
```

элементы которой инициализируются самой функцией CreateThread. При удачном выполнении функции поля hProcess и hThread содержат дескрипторы, а dwProcessId и dwThreadId содержат идентификаторы нового процесса и его первичной нити.

Процесс может узнать свой дескриптор и идентификатор с помощью функций:

```
HANDLE GetCurrentProcess(VOID);
DWORD GetCurrentProcessId(VOID);
```

Завершение процесса

Завершение процесса обеспечивают следующие две функции. Первая из них завершает процесс их которого она вызвана с указанным кодом завершения `fuExitCode`:

```
VOID ExitProcess(UINT fuExitCode);
```

Вторая функция позволяет завершить процесс с дескриптором hProcess вернув указанный код завершения `fuExitCode`:

```
BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode);
```

Управление нитями

Нить (thread) описывает последовательность исполнения кода внутри процесса. При инициализации процесса система создает первичную нить (primary thread) в которой запускается функция `WinMain()`. Процессы могут создавать дополнительные нити, чтобы уменьшить время реакции, улучшить структурирование алгоритма, и, на многопроцессорных системах, повысить производительность.

Создание и завершение нити

В Win32 нити создаются с помощью функции:

```
HANDLE CreateThread(
     LPSECURITY_ATTRIBUTES lpsa,
     DWORD cbStack,
     LPTHREAD_START_ROUTINE lpStartAddress,
     LPVOID lpvThreadParm,
     DWORD fdwCreate,
     LPDWORD lpIDThread);
```

Опишем параметры этой функции:

`lpsa` атрибуты защиты для создаваемого объекта «нить». Устанавливаются по умолчанию, если передано `NULL`.

`cbStack` определяет размер стека нити. Если передан 0, размер стека берется из информации, занесенной в выполняемый файл редактором связей.

`lpStartAddress` адрес функции, которая будет выполняться в новой нити.

`lpvThreadParm` адрес параметров, передаваемых нити.

`fdwCreate` дополнительные флаги, управляющие свойствами создаваемой нити. Создание нити со свойствами по умолчанию, происходит если передан `NULL`.

`lpIDThread` адрес двойного слова в котором система разместит идентификатор созданной нити. Передача `NULL` в качестве этого параметра не допускается.

Завершение работы нити выполняют следующие две функции:

```
VOID ExitThread(UINT fuExitCode);
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Как и в случае процесса, первая функция завершает вызвавшую её нить. При этом стек нити разрушается. Вторая функция завершает нить с переданным дескриптором `hThread`, и стек этой нити не разрушается, а остается в

памяти до конца работы процесса. Это связано с тем, что такая нить завершается принудительно, и могли остаться действительные ссылки на её стек.

Заметим, что использование `TerminateThread` должно быть редким, аварийным средством. Нормальное завершение нити должно осуществляться с помощью `ExitThread`, вызываемой самой нитью по результатам анализа своего состояния и/или глобальных флагов завершения нити. С другой стороны, практика программирования, когда глобально доступны ссылки на данные в стеке, порочна и ведет ко многим потенциальным, а часто и действительным ошибкам.

Если завершение нити происходит не перед завершением процесса (которое приводит к автоматической очистке дескрипторов и освобождению ресурсов), также следует освободить дескриптор нити, вызвав для него функцию:

```
BOOL CloseHandle(Handle hObject);
```

Приведем простой пример запуска нескольких нитей в консольном приложении:

```
#include <stdio.h>
#include <windows.h>
const int numThreads = 4;

DWORD WINAPI helloFunc(LPVOID pArg)
{
     int *myNum = (int *)pArg;
     printf("Hello Thread %d\n", *myNum);
     return 0;
}

main()
{
     HANDLE hThread[numThreads];
     int tNum[numThreads];

     for (int i = 0; i < numThreads; i++)
     {
       tNum[i] = i;
```

```
        hThread[i] =
             CreateThread(NULL, 0, helloFunc, (LPVOID)&tNum[i],
                0, NULL );
     }

     WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE);

     return 0;
}
```

Состояние нити можно проверить из другой нити функцией:

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpdwExitCode);
```

Если нить ещё не завершилась, в двойном слове на которое указывает `lpdwExitCode` будет записано `STILL_ACTIVE`. Для завершенной нити будет записан её код завершения.

Сам нить может получить свой дексриптор и идентификатор с помощь функций:

```
HANDLE GetCurrentThread(VOID);
DWORD GetCurrentThreadId(VOID);
```

Функция GetCurrentThread() возвращает «псевдодескриптор», который осмыслен только в самой функции нити. Если необходимо использовать его повсюду, следует получить «полноценный» дескриптор объекта ядра с помощью функции:

```
BOOL DuplicateHandle(
     HANDLE hSourceProcess,
     HANDLE hSource,
     HANDLE hTargetProcess,
     LPHANDLE lphTarget,
     DWORD fdwAccess,
     BOOL fInherit,
     DWORD fdwOptional);
```

Применение этой функции для получения дескриптора может выполняться например, так:

```
HANDLE hThread;
DuplicateHandle(GetCurrentProcess(), //дескриптор процесса
  GetCurrentThread(), //псевдодескриптор нити
  GetCurrentProcess(), //дескриптор процесса для дескриптора нити
  &hThread, //новый, "настоящий" дескриптор нити
  0,      //игнорируется
  FALSE,  //не наследуется
  DUPLICATE_SAME_ACCESS); //те же права доступа
```

Выяснить приоритет нити позволяет функция:

```
int GetThreadPriority(HANDLE hThread);
```

Изменить относительный (относительно класса приоритета процесса) приоритет нити позволяет следующая функция:

```
BOOL SetThreadPriority(HANDLE hThread, int nPriority);
```

Относительные приоритеты задаются константами:

- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_ABOVE_NORMAL
- THREAD_PRIORITY_HIGHEST

Следующие функции позволяют проверить или установить режим динамического приоритета (dynamic boosting) нити:

```
BOOL GetThreadPriorityBoost(
     HANDLE hThread,
     PBOOL pDisablePriorityBoost);
BOOL WINAPI SetThreadPriorityBoost(
     HANDLE hThread,
     BOOL DisablePriorityBoost);
```

Включение этого режима приводит к тому, что проснувшаяся нить, получает временно повышенный динамический приоритет.

Приостанавливать и возобновлять нити можно с помощью функций:

```
DWORD SuspendThread(HANDLE hThread);
```

```
DWORD ResumeThread{HANDLE hThread);
```

Эти функции считающие, поэтому их вызовы должны быть сбалансированы. Нить можно приостанавливать не более 127 раз. Указанием флага CREATE_SUSPEND в функции CreateThread() можно получить приостановленную при создании нить, это равносильно одному вызову SuspendThread().

Синхронизация

Критические секции

Критические секции в Win32 API используются только для синхронизации нитей одного процесса.

Для создания критических секций, определяется переменная типа CRITICAL_SECTION. Хотя эта структура полностью определена в заголовочном файле winnt.h, безопасней её считать чёрным ящиком. Перед использованием эта переменная должна быть инициализирована с помощью функции

```
VOID InitializeCriticalSection(
     LPCRITICAL_SECTION lpCrititcalSection)
```

После инициализации эту переменную можно использовать для организации критических секций. В начале критической секции вызывается функция

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

В конце критической секции вызывается функция

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

По завершении работы ресурсы следует освободить функцией:

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

Приведем пример использования критической секции:

```
CRITICAL_SECTION cs;
```

```
HANDLE hThread[2];

int WINAPI WinMain()
{
     ...
// инициализируем критическую секцию
     InitializeCriticalSection(&cs);
// создаем две нити
     hThread[0]=CreateThread(..., FirstThread,...);
     hThread[1]=CreateThread(..., SecondThread,...);
     ...
     ...
// ждем, когда нити заверщаться
     WaitForMultipleObjects(2, hThreads, TRUE, INFINITE);
// закрываем дескрипторы
     CloseHandle(hThread[0]);
     CloseHandle(hThread[1]);
// удаляем критическую секцию
     DeleteCriticalSection(&cs);
     ...
}

DWORD WINAPI FirstThread(LPVOID lpvThreadParm)
{
     ...
     EnterCriticalSection(&cs);
     ...
     LeaveCriticalSection(&cs);
     ...
}

DWORD WINAPI SecondThread(LPVOID lpvThreadParm)
{
     ...
     EnterCriticalSection(&cs);
     ...
     LeaveCriticalSection(&cs);
     ...
}
```

Синхронизация нитей с объектами ядра

Для синхронизации можно использовать следующие объекты ядра:

- процессы;
- нити;
- файлы;
- консольный ввод;
- уведомления об изменении файлов;
- мьютексы;
- семафоры;
- события.

Все эти объекты могут быть в двух состояниях signaled или nonsignaled. Как правило состояние signaled может трактоваться как освобождение объекта, а состояние nonsignaled — объект занят.

Для ожидания перехода объекта (объектов) в состояние signaled, используются две функции — для одного объекта:

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dw Timeout);
```

и для нескольких (максимум 64) объектов:

```
DWORD WaitForMultipleObjects(DWORD cObjects, LPHANDLE lpHandles,
     BOOL bWaitAll,
     DWORD dwTimeout);
```

`lpHandler` — указатель на массив дескрипторов объектов, `cObjects` — размер этого массива.

Параметр `dwTimeout` позволяет задавать интервал ожидания в миллисекундах. Если указать `dwTimeout` равным нулю, ожидания не будет, будет просто произведена проверка состояния объекта (объектов). Если указать `dwTimeout` равным `INFINITY`, функция будет не ограничено долго ждать перехода объекта (объектов) в состояние signaled.

Параметр `bWaitAll` определяет, должны ли ВСЕ объекты перейти в состояние signaled.

Эти функции могут возвращать одно из следующих значений:

`WAIT_OBJECT_0 .. WAIT_OBJECT_0 + cObjects - 1` объект перешел в состояние signaled

`WAIT_TIMEOUT` объект (объекты) не перешли в состояние signaled за указанный промежуток времени.

`WAIT_ABANDONED .. WAIT_ABANDONED + cObject - 1`

`WAIT_FAILED` произошла ошибка.

Все перечисленные объекты ядра могут быть использованы для синхронизации, но прежде всего следует использовать объекты, специально предназначенные для синхронизации — мьютексы и семафоры.

Мьютексы

Мьютекс — объект ядра. По своей функциональности мьютексы аналогичны критическим секциям, но мьютексы позволяют синхронизировать также и нити разных процессов. Получить мьютес можно двумя способами — создать новый мьютекс:

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpsa,
     BOOL fInitialOwner,
     LPSTR lpszMutexName)
```

или открыть уже существующий мьютекс:

```
HANDLE OpenMutex(DWORD fdwAccess,
     BOOL fInherit,
     LPSTR lpszMutexName);
```

В Vista и MS Windows Server 2008 доступна также функция

```
HANDLE CreateMutexEx(
     LPSECURITY_ATTRIBUTES lpMutexAttributes,
     LPCTSTR lpName,
     DWORD dwFlags,
     DWORD dwDesiredAccess);
```

позволяющая создать или открыть мьютекс.

Мьютексы в этих функциях распознаются по имени. Если вместо имени функции CreateMutex передан NULL, будет создан новый мьютекс, но другие процессы не смогут открывать его по имени. Функция OpenMutex должна получать имя ранее кем-либо созданного мьютекса. Дополнительно можно получить доступ к мьютексу без имени с помощью функции DuplicateHandle.

Захват мьютекса осуществляется вызовом одной из Wait...-функций. Особенностью мьютекса является то, что он не просто занят, он занят определенной нитью. Если wait-функцию вызывает другая нить, не владелец мьютекса, то она сразу приостанавливается. Если для занятого мьютекса вызывается одна из wait-функций нитью-владельцем, то она не приостанавливается, а только увеличивается счетчик. Для освобождения используется функция:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Если владелец несколько раз вызывал wait-функцию, он должен столько же раз вызвать и ReleaseMutex.

Семафоры

Семафоры в Win32 API представляют классические примитивы синхронизации, формализованные Э.Дейкстра в 1968году. Отметим, что по своей семантике, семафоры не принадлежат какому-либо процессу или нити. Это глобальные объекты операционной системы, и выполнять операции с семафором может любой процесс, который знает имя семафора или каким-то образом сумел получить дубликат дескриптора семафора. Создаются семафоры функцией:

```
HANDLE CreateSemaphor(LPSECURITY_ATTRIBUTES lpsa,
    LONG cSemInitial, LONG cSemMax,
    LPTSTR lpszSemName);
```

Параметры этой функции означают:

`lpsa` атрибуты безопасности объекта ядра. Для настроек по умолчанию, передаем NULL;

`cSemMax` ≥ 1 — максимальное значение счетчика семафора.

`cSemInitial` — начальное значение счетчика семафора, $0 \leq \text{cSemInitial} \leq \text{cSemMax}$.

`lpszSemName` - имя семафора.

Уже созданный семафор можно открыть функцией:

```
HANDLE OpenSemaphor(DWORD fdwAccess,
     BOOL fInherit,
     LPTSTR lpszSemName);
```

Уменьшение счётчика семафора производится вызовом какой-либо из wait-функций. Увеличение счётчика осуществляется функцией:

```
BOOL ReleaseSemaphor(HANDLE hSemaphore,
     LONG cRelease,
     LPLONG lplPrevious);
```

Эта функция позволяет увеличивать счётчик на несколько единиц, задавая величину изменения в параметре cRelease. Параметр lplPrevious позволяет получить значение счётчика до вызова функции.

События

Win32 API предоставляет также события для синхронизации. Используя события нити могут уведомлять друг друга. События бывают со сбросом вручную (manual-reset events) и с автоматическим сбросом (auto-reset events). События со сбросом вручную используются для уведомления нескольких нитей, а с автоматическим сбросом — для уведомления единственной нити.

События создаются следующей функцией:

```
HANDLE CreateEvent(
     LPSECURITY_ATTRIBUTES lpEventAttributes,
     BOOL bManualReset,  // TRUE => manual reset
```

```
    BOOL bInitialState, // TRUE => begin signaled
    LPCSTR lpName);     // text name for object
```

Параметры определяют, является ли создаваемое событие событием с ручным сбросом (bManualReset), и будет ли оно сразу по создании в состоянии signaled (bInitialState).

Открывает уже созданное событие функция:

```
HANDLE OpenEvent(DWORD fdwAccess,
    BOOL fInherit,
    LPSTR lpName);
```

Более ненужное событие можно освободить вызывая `CloseHandle()`

Получение/ожидание события осуществляется с помощью wait-функций. Обратите внимание, что использование множественного ожидание с взведенным флагом bWaitAll небезопасно. За перевод события в состояние signaled отвечает функция:

```
BOOL SetEvent(HANDLE event);
```

Сброс состояния signaled для событий с автоматическим сбросом происходит сразу после обработки события первой же wait-функцией. Для событий с ручным сбросом требуется вызвать специальную функцию:

```
BOOL ResetEvent(HANDLE event);
```

Следующая функция не должна использоваться в новых программах, она не безопасна и ведет себя по разному в различных версиях Windows.

```
BOOL PulseEvent(HANDLE event);
```

Локальная память нити

В WinAPI предусмотрена локальная память нити (TLS-Thread Local Storage). Если в программе используется многонитевая версия стандартной системной библиотеки C, это означает, что TLS уже используется. Её можно использовать и в создаваемых программах и библиотеках. Особенно это может

быть полезно для библиотек, позволяя им использовать глобальные переменные в многозадачной среде.

Следующие функции позволяют манипулировать TLS.

```
DWORD TlsAlloc(VOID);
BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpvTlsValue);
LPVOID TlsGetValue(DWORD dwTlsIndex);
BOOL TlsFree(DWORD dwTlsIndex);
```

Работа с TLS осуществляется следующим образом:

1. Для размещения переменной запрашиваем индекс функцией TlsAlloc. Если в TLS нет больше свободной памяти, возвращается значение TLS_OUT_OF_INDEXES. Если индекс выделен, сохраняем его в глобальной переменной (он должен быть доступен всем нитям процесса).
2. С помощью функции TlsSetValue можем по полученному индексу сохранять 32-битовое значение lpvTlsValue. Каждая нить сделавшая это сохраняет гарантировано различные и недоступные другим нитям значения.
3. Прочитать сохраненное по индексу значение можно функцией TlsGetValue.
4. Более ненужный элемент TLS можно освободить вызовом TlsFree.

Литература

1. *Рихтер Дж.* Windows для профессионалов. / Пер. с англ. - М.: Русская редакция, 1995. - 720 с.: ил.
2. MSDN Library. Windows API. - http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx