Уважаемым читателям, которые
(пока еще) не знают кто такой
Дж. Рихтер и что такое SDK.

Процессы в Windows

Процессы

Процессом обычно называют экземпляр выполняемой программы.

Хотя на первый взгляд кажется что программа и процесс понятия практически одинаковые, они фундаментально отличаются друг от друга. Программа представляет собой статический набор команд, а процесс это набор ресурсов и данных, использующихся при выполнении программы. Процесс в Windows состоит из следующих компонентов:

- Структура данных, содержащая всю информацию о процессе, в том числе список открытых дескрипторов различных системных ресурсов, уникальный идентификатор процесса, различную статистическую информацию и т.д.;
- Адресное пространство – диапазон адресов виртуальной памяти, которым может пользоваться процесс;
- Исполняемая программа и данные, проецируемые на виртуальное адресное пространство процесса.

Потоки

Процессы инертны. Отвечают же за исполнение кода, содержащегося в адресном пространстве процесса, потоки. Поток (thread) – некая сущность внутри процесса, получающая процессорное время для выполнения. В каждом процессе есть минимум один поток. Этот первичный поток создается системой автоматически при создании процесса. Далее этот поток может породить другие потоки, те в свою очередь новые и т.д. Таким образом, один процесс может владеть несколькими потоками, и тогда они одновременно исполняют код в адресном пространстве процесса. Каждый поток имеет:

- Уникальный идентификатор потока;
- Содержимое набора регистров процессора, отражающих состояние процессора;
- Два стека, один из которых используется потоком при выполнении в режиме ядра, а другой – в пользовательском режиме;
- Закрытую область памяти, называемую локальной памятью потока (thread local storage, TLS) и используемую подсистемами, run-time библиотеками и DLL.

Планирование потоков

Чтобы все потоки работали, операционная система отводит каждому из них определенное процессорное время. Тем самым создается иллюзия одновременного выполнения потоков (разумеется, для многопроцессорных компьютеров возможен истинный параллелизм). В Windows реализована система вытесняющего планирования на основе приоритетов, в которой всегда выполняется поток с наибольшим приоритетом, готовый к выполнению. Выбранный для выполнения поток работает в течение некоторого периода, называемого квантом. Квант определяет, сколько времени будет выполняться поток, пока операционная система не прервет его. По окончании кванта операционная система проверяет, готов ли к выполнению другой поток с таким же (или большим) уровнем приоритета. Если таких потоков не оказалось, текущему потоку выделяется еще один квант. Однако поток может не полностью использовать свой квант. Как только другой поток с более высоким приоритетом готов к выполнению, текущий поток вытесняется, даже если его квант еще не истек.

Квант не измеряется в каких бы то ни было единицах времени, а выражается целым числом. Для каждого потока хранится текущее значение его кванта. Когда потоку *выделяется квант процессорного времени*, это значит, что его квант устанавливается в начальное значение. Оно зависит от операционной системы. Например, для Win2000 Professional начальное значение кванта равно 6, а для Win2000 Server – 36.

Это значение можно изменить вызвав Control Panel - > System -> Advanced -> Performance options. Значение «Applications» – как для Win2000 Professional; «Background Services» – как для Win2000 Server. Или напрямую в ключе реестра HKLM\System\CurrentControlset\Control\PriorityControl\ Win32PrioritySeparation.

Всякий раз, когда возникает прерывание от таймера, из кванта потока вычитается 3, и так до тех пор, пока он не достигнет нуля. Частота срабатывания таймера зависит от аппаратной платформы. Например, для большинства однопроцессорных x86 систем он составляет 10мс, а на большинстве многопроцессорных x86 систем – 15мс.

В любом случае операционная система должна определить, какой поток выполнять следующим. Выбрав новый поток, операционная система переключает контекст. Эта операция заключается в сохранении параметров выполняемого потока (регистры процессора, указатели на стек ядра и пользовательский стек, указатель на адресное пространство, в котором выполняется поток и др.), и загрузке аналогичных параметров для другого потока, после чего начинается выполнение нового потока.

Планирование в Windows осуществляется на уровне потоков, а не процессов. Это кажется понятным, так как сами процессы не выполняются, а лишь предоставляют ресурсы и контекст для выполнения потоков. Поэтому при планировании потоков, система не обращает внимания на то, какому процессу они принадлежат. Например, если процесс А имеет 10 готовых к выполнению потоков, а процесс Б – два, и все 12 потоков имеют одинаковый приоритет, каждый из потоков получит 1/12 процессорного времени.

Приоритеты

В Windows существует 32 уровня приоритета, от 0 до 31. Они группируются так: 31 – 16 уровни реального времени; 15 – 1 динамические уровни; 0 – системный уровень, зарезервированный для потока обнуления страниц (zero-page thread).

При создании процесса, ему назначается один из шести *классов приоритетов*:

Real time class (значение 24),
High class (значение 13),
Above normal class (значение 10),
Normal class (значение 8),
Below normal class (значение 6),
и Idle class (значение 4).

В Windows NT/2000/XP можно посмотреть приоритет процесса в Task Manager.

Above normal и Below normal появились начиная с Win2000.

Приоритет каждого потока (*базовый приоритет потока*) складывается из приоритета его процесса и *относительного приоритета* самого потока. Есть семь относительных приоритетов потоков:

Normal: такой же как и у процесса;
Above normal: +1 к приоритету процесса;
Below normal: –1;
Highest: +2;
Lowest: –2;

Time critical: устанавливает базовый приоритет потока для Real time класса в 31, для остальных классов в 15.

Idle: устанавливает базовый приоритет потока для Real time класса в 16, для остальных классов в 1.

В следующей таблице показаны приоритеты процесса, относительный и базовый приоритеты потока.

Приоритет потока	**Класс процесса**	**Класс процесса**					
		Idle class	**Below normal class**	**Normal class**	**Above normal class**	**High class**	**Real time class**
1		Idle	Idle	Idle	Idle	Idle	
2		Lowest					
3		Below ...					
4	**Idle class**	Normal	Lowest				
5		Above ...	Below ...				
6	**Below normal class**	Highest	Normal	Lowest			
7			Above ...	Below ...			
8	**Normal class**		Highest	Normal	Lowest		
9				Above ...	Below ...		
10	**Above normal class**			Highest	Normal		
11					Above ...	Lowest	
12					Highest	Below ...	
13	**High class**					Normal	
14						Above ...	
15						Highest	
15		Time critical	Time critical	Time critical	Time critical	Time critical	
16							Idle
17							
18							
19							
20							
21							
22							Lowest
23							Below ...
24	**Real time class**						Normal
25							Above ...
26							Highest
27							
28							
29							
30							
31							Time critical

Привязка к процессорам

Если операционная система выполняется на машине, где установлено более одного процессора, то по умолчанию, поток выполняется на любом доступном процессоре. Однако в некоторых случаях, набор процессоров, на которых поток может работать, может быть ограничен. Это явление называется привязкой к процессорам (processor affinity). Можно изменить привязку к процессорам программно, через Win32-функции планирования.

Память

Каждому процессу в Win32 доступно линейное 4-гигабайтное (2^{32} = 4 294 967 296) виртуальное адресное пространство. Обычно верхняя половина этого пространства резервируется за операционной системой, а вторая половина доступна процессу.

Виртуальное адресное пространство процесса доступно всем потокам этого процесса. Иными словами, все потоки одного процесса выполняются в едином адресном пространстве.

С другой стороны, механизм виртуальной памяти позволяет изолировать процессы друг от друга. Потоки одного процесса не могут ссылаться на адресное пространство другого процесса.

Виртуальная память может вовсе не соответствовать структуре физической памяти. Диспетчер памяти транслирует виртуальные адреса на физические, по которым реально хранятся данные. Поскольку далеко не всякий компьютер в состоянии выделить по 4 Гбайт физической памяти на каждый процесс, используется механизм подкачки (swapping). Когда оперативной памяти не хватает, операционная система перемещает часть содержимого памяти на диск, в файл (swap file или page file), освобождая, таким образом, физическую память для других процессов. Когда поток обращается к странице виртуальной памяти, записанной на диск, диспетчер виртуальной памяти загружает эту информацию с диска обратно в память.

Создание процессов

Создание Win32 процесса осуществляется вызовом одной из таких функций, как CreateProcess, CreateProcessAsUser (для Win NT/2000) и CreateProcessWithLogonW (начиная с Win2000) и происходит в несколько этапов:

- Открывается файл образа (EXE), который будет выполняться в процессе. Если исполняемый файл не является Win32 приложением, то ищется образ поддержки (support image) для запуска этой программы. Например, если исполняется файл с расширением .bat, запускается cmd.exe и т.п.

В WinNT/2000 для отладки программ реализовано следующее. CreateProcess, найдя исполняемый Win32 файл, ищет в SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Option раздел с именем и расширением запускаемого файла, затем ищет в нем параметр Debugger, и если строка не пуста, запускает то, что в ней написано вместо данной программы.

- Создается объект Win32 «процесс».
- Создается первичный поток (стек, контекст и объект «поток»).
- Подсистема Win32 уведомляется о создании нового процесса и потока.
- Начинается выполнение первичного потока.
- В контексте нового процесса и потока инициализируется адресное пространство (например, загружаются требуемые DLL) и начинается выполнение программы.

Завершение процессов

Процесс завершается если:

- Входная функция первичного потока возвратила управление.
- Один из потоков процесса вызвал функцию ExitProcess.
- Поток другого процесса вызвал функцию TerminateProcess.

Когда процесс завершается, все User- и GDI-объекты, созданные процессом, уничтожаются, объекты ядра закрываются (если их не использует другой процесс), адресное пространство процесса уничтожается.

Пример 1.
Программа создает процесс “Калькулятор”.

```
#include <windows.h>

int main(int argc, char* argv[])
{
     STARTUPINFO si;
     PROCESS INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( !CreateProcess( NULL, "c:/windows/calc.exe", NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
       return 0;
```

```
    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
       return 0;
}
```

Создание потоков

Первичный поток создается автоматически при создании процесса. Остальные потоки создаются функциями CreateThread и CreateRemoteThread (только в Win NT/2000/XP).

Завершение потоков

Поток завершается если

- Функция потока возвращает управление.
- Поток самоуничтожается, вызвав ExitThread.
- Другой поток данного или стороннего процесса вызывает TerminateThread.
- Завершается процесс, содержащий данный поток.

Если вы используете в своем приложении LibCMt C run-time библиотеку, Microsoft настоятельно рекомендует вместо Win32 API функций использовать их аналоги из C run-time: _beginthread, _beginthreadex, _endthread и _endthreadex.

Объекты ядра

Эти объекты используются системой и пользовательскими приложениями для управления множеством самых разных ресурсов: процессами, потоками, файлами и т.д. Windows позволяет создавать и оперировать с несколькими типами таких объектов, в том числе:

Kernel object	**Объект ядра**	**Kernel object**	**Объект ядра**
Access token	Маркер доступа	Module	Подгружаемый модуль (DLL)
Change notification	Уведомление об изменениях на диске	Mutex	Мьютекс
I/O completion ports	Порт завершения ввода-вывода	Pipe	Канал
Event	Событие	Process	Процесс
File	Файл	Semaphore	Семафор
File mapping	Проекция файла	Socket	Сокет
Heap	Куча	Thread	Поток
Job	Задание	Timer	Ожидаемый таймер
Mailslot	Почтовый слот		

Объект ядра это, по сути, структура, созданная ядром и доступная только ему. В пользовательское приложение передается только описатель (handle) объекта, а управлять объектом ядра можно с помощью функций Win32 API.

Wait функции

Как можно приостановить работу потока? Существует много способов. Вот некоторые из них.

Функция Sleep() приостанавливает работу потока на заданное число миллисекунд. Если в качестве аргумента вы укажите 0 ms, то произойдет следующее. Поток откажется от своего кванта процессорного времени, однако тут же появится в списке потоков готовых к выполнению. Иными словами произойдет намеренное переключение потоков.

(Вернее сказать, попытка переключения. Ведь следующим для выполнения потоком вполне может стать тот же самый.)

Функция WaitForSingleObject() приостанавливает выполнение потока до тех пор, пока не произойдет одно из двух событий:

- истечет таймаут ожидания;
- ожидаемый объект перейдет в сигнальное (signaled) состояние.

По возвращаемому значению можно понять, какое из двух событий произошло. Ожидать с помощью wait-функций можно большинство объектов ядра, например, объект «процесс» или «поток», чтобы определить, когда они завершат свою работу.

Функции WaitForMultipleObjects передается сразу массив объектов. Можно ожидать срабатывания сразу всех объектов или какого-то одного из них.

Пример 2. Программа создает два одинаковых потока и ожидает их завершения. Потоки просто выводят текстовое сообщение, которое передано им при инициализации.

```
#include <windows.h>
#include <process.h>

unsigned    stdcall ThreadFunc( void * arg)          // Функция потока
{
        char ** str = (char**)arg;
        MessageBox(0,str[0],str[1],0);

     endthreadex( 0 );
    return 0;
};

int main(int argc, char* argv[])
{
        char * InitStr1[2] = {"First thread running!","11111"};     // строка для первого потока
        char * InitStr2[2] = {"Second thread running!","22222"};    // строка для второго потока

        unsigned uThreadIDs[2];
        HANDLE   hThreads[2];

        hThreads[0] = (HANDLE) beginthreadex( NULL, 0, &ThreadFunc, InitStr1, 0, &uThreadIDs[0]);
        hThreads[1] = (HANDLE) beginthreadex( NULL, 0, &ThreadFunc, InitStr2, 0, &uThreadIDs[1]);

    // Ждем, пока потоки не завершат свою работу
    WaitForMultipleObjects(2, hThreads, TRUE, INFINITE ); // Set no time-out

    // Закрываем дескрипторы
    CloseHandle( hThreads[0] );
    CloseHandle( hThreads[1] );
        return 0;
}
```

Синхронизация потоков

Работая параллельно, потоки совместно используют адресное пространство процесса. Также все они имеют доступ к описателям (handles) открытых в процессе объектов. А что делать, если несколько потоков одновременно обращаются к одному ресурсу или необходимо как-то упорядочить работу потоков? Для этого используют объекты синхронизации и соответствующие механизмы.

Мьютексы.

Мьютексы (Mutex) это объекты ядра, которые создаются функцией CreateMutex(). Мьютекс бывает в двух состояниях – занятом и свободном. Мьютексом хорошо защищать единичный ресурс от одновременного обращения к нему разными потоками.

Пример 3. Допустим, в программе используется ресурс, например, файл или буфер в памяти. Функция WriteToBuffer() вызывается из разных потоков. Чтобы избежать коллизий при одновременном обращении к буферу из разных потоков, используем мьютекс. Прежде чем обратиться к буферу, ожидаем «освобождения» мютекса.

```
HANDLE hMutex;
int main()
{
hMutex = CreateMutex( NULL, FALSE, NULL);  // Создаем мьютекс в свободном состоянии
…
// Создание потоков, и т.д.
…
}

BOOL WriteToBuffer()
{
    DWORD dwWaitResult;

    // Ждем освобождения мьютекса перед тем как обратиться к буферу.
    dwWaitResult = WaitForSingleObject( hMutex, 5000L);   // 5 секунд на таймаут

    if (dwWaitResult == WAIT_TIMEOUT)       // Таймаут. Мьютекс за это время не освободился.
    {
       return FALSE; :
    }
    else       // Мьютекс освободился, и наш поток его занял. Можно работать.
    {
          Write_to_the_buffer().
            ……
         ReleaseMutex(hMutex);        // Освобождаем мьютекс.
    }
    return TRUE;
}
```

Семафоры.

Семафор (Semaphore) создается функцией CreateSemaphore(). Он очень похож на мьютекс, только в отличие от него у семафора есть счетчик. Семафор открыт если счетчик больше 0 и закрыт, если счетчик равен 0. Семафором обычно «огораживают» наборы равнозначных ресурсов (элементов), например очередь, список и т.п.

Пример 4. Классический пример использования семафора это очередь элементов, которую обрабатывают несколько потоков. Потоки «разбирают» элементы из очереди. Если очередь пуста, потоки должны «спать», ожидая появления новых элементов. Для учета элементов в очереди используется семафор.

```
class CMyQueue
 {
        HANDLE m hSemaphore; // Семафор для учета элементов очереди
        // Описание других объектов для хранения элементов очереди

 public:
        CMyQueue()
        {
           m hSemaphore = CreateSemaphore(NULL, 0, 1000, NULL);//начальное значение счетчика = 0
                                                               //максимальное значение = 1000
               // Инициализация других объектов
        }

        ~CMyQueue()
        {
              CloseHandle( m hSemaphore);
              // Удаление других объектов
        }

        bool AddItem(void * NewItem)
        {
              // Добавляем элемент в очередь
              // Увеличиваем счетчик семафора на 1.
              ReleaseSemaphore(m_hSemaphore,1, NULL);
        }

        bool GetItem(void * Item)
        {
              // Если очередь пуста, то потоки, вызвавшие этот метод,
              // будут находиться в ожидании…
             WaitForSingleObject(m_hSemaphore,INFINITE);

              // Удаляем элемент из очереди
        }
 };
```

Замечание. В этом примере мы считаем, что сами процедуры добавления элемента в очередь и удаления из очереди безопасны с точки зрения многопоточности. Не будем пока касаться деталей их реализации. Подробнее мы рассмотрим это в примере 9.

События.

События (Event), также как и мьютексы имеют два состояния – установленное и сброшенное. События бывают со сбросом вручную и с автосбросом. Когда поток дождался (wait-функция вернула управление) события с автосбросом, такое событие автоматически сбрасывается. В противном случае событие нужно сбрасывать вручную, вызвав функцию ResetEvent(). Допустим, сразу несколько потоков ожидают одного и того же события, и событие сработало. Если это было событие с автосбросом, то оно позволит работать только одному потоку (ведь сразу же после возврата из его wait-функции событие сбросится автоматически!), а остальные потоки останутся ждать. Если же это было событие со сбросом вручную, то все потоки получат управление, а событие так и останется в установленном состоянии, пока какой-нибудь поток не вызовет ResetEvent().

Пример 5.

Вот еще один пример многопоточного приложения. Программа имеет два потока; один готовит данные, а второй отсылает их на сервер. Разумно распараллелить их работу. Здесь потоки должны работать по очереди. Сначала первый поток готовит порцию данных. Потом второй поток отправляет ее, а первый тем временем готовит следующую порцию и т.д. Для такой синхронизации понадобится два event'а с автосбросом.

```
unsigned    stdcall CaptureThreadFunc( void * arg)   // Поток, готовящий данные
{
       while (bSomeCondition)
       {
            WaitForSingleObject(m hEventForCaptureTh,INFINITE);  // Ждем своего события
            ... // Готовим данные
            SetEvent(hEventForTransmitTh); // Разрешаем работать второму потоку
       }
    _endthreadex( 0 );
    return 0;
};

unsigned    stdcall TransmitThreadFunc( void * arg)  // Поток, отсылающий данные.
{
       while (bSomeCondition)
       {
            WaitForSingleObject(m hEventForTransmitTh,INFINITE);  // Ждем своего события
            ... // Данные готовы, формируем из них пакет для отправки
            SetEvent(hEventForCaptureTh); // Разрешаем работать первому потоку, а сами...
           // ... отправляем пакет
       }
     endthreadex( 0 );
    return 0;
};

int main(int argc, char* argv[])  // Основной поток
{
       // Создаем два события с автосбросом, со сброшенным начальным состоянием
       hEventForCaptureTh  = CreateEvent(NULL,FALSE,FALSE,NULL);
       hEventForTransmitTh  = CreateEvent(NULL,FALSE,FALSE,NULL);

       // Создаем потоки
       hCaptureTh = (HANDLE) beginthreadex( NULL, 0, &CaptureThreadFunc, 0, 0, &uTh1);
       hTransmitTh = (HANDLE) beginthreadex( NULL, 0, &TransmitThreadFunc, 0, 0, &uTh2);

       // Запускаем первый поток
      SetEvent(hEventForCaptureTh);
       ....
}
```

Пример 6.

Другой пример. Программа непрерывно в цикле производит какие-то вычисления. Нужно иметь возможность приостановить на время ее работу. Допустим, это просмотрщик видео файлов, который в цикле, кадр за кадром отображает информацию на экран. Не будем вдаваться в подробности видео функций. Реализуем функции Pause и Play для программы. Используем событие со сбросом вручную.

```
// Главная функция потока, которая в цикле отображает кадры
unsigned    stdcall VideoThreadFunc( void * arg)
{
        while (bSomeCondition)
        {
             WaitForSingleObject(m hPauseEvent,INFINITE);  // Если событие сброшено, ждем
             ... // Отображаем очередной кадр на экран
        }
    _endthreadex( 0 );
    return 0;
};

void Play()
{
        SetEvent(m_hPauseEvent);
};

void Pause()
{
        ResetEvent(m hPauseEvent);
};
```

Функция PulseEvent() устанавливает событие и тут же переводит его обратно в сброшенное состояние; ее вызов равнозначен последовательному вызову SetEvent() и ResetEvent(). Если PulseEvent вызывается для события со сбросом в ручную, то все потоки, ожидающие этот объект, получают управление. При вызове PulseEvent для события с автосбросом пробуждается только один из ждущих потоков. А если ни один из потоков не ждет объект-событие, вызов функции не дает никакого эффекта.

Пример 7.
Реализуем функцию NextFrame() для предыдущего примера для промотки файла вручную по кадрам.

```
void NextFrame()
{
        PulseEvent(m hPauseEvent);
};
```

Ожидаемые таймеры

Пожалуй, ожидаемые таймеры – самый изощренный объект ядра для синхронизации. Появились они, начиная с Windows 98. Таймеры создаются функцией CreateWaitableTimer и бывают, также как и события, с автосбросом и без него. Затем таймер надо настроить функцией SetWaitableTimer. Таймер переходит в сигнальное состояние, когда истекает его таймаут. Отменить «тиканье» таймера можно функцией CancelWaitableTimer. Примечательно, что можно указать callback функцию при установке таймера. Она будет выполняться, когда срабатывает таймер.

Пример 8. Напишем программу-будильник используя WaitableTimer-ы. Будильник будет срабатыват раз в день в 8 утра и “пикать” 10 раз. Используем для этого два таймера, один из которых с callback-функцией.

```
#include <process.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>

#define HOUR   (8)     // время, когда срабатывает будильник (часы)
#define RINGS  (10)    // сколько раз пикать

HANDLE hTerminateEvent ;

// callback функция таймера
VOID CALLBACK TimerAPCProc(LPVOID, DWORD, DWORD)
{
        Beep(1000,500); // звоним!
};
```

```
// функция потока
unsigned    stdcall ThreadFunc(void *)
{
        HANDLE hDayTimer = CreateWaitableTimer(NULL,FALSE,NULL);
        HANDLE hAlarmTimer = CreateWaitableTimer(NULL,FALSE,NULL);

        HANDLE h[2];   // мы будем ждать эти объекты
        h[0] = hTerminateEvent; h[1] = hDayTimer;

        int iRingCount=0;      // число "звонков"
        int iFlag;
        DWORD dw;

        // немного помучаемся со временем, т.к. таймер принимает его только в формате FILETIME
        LARGE_INTEGER liDueTime, liAllDay;
        liDueTime.QuadPart=0;
        // сутки в 100-наносекундных интервалах = 10000000 * 60 * 60 * 24 = 0xC92A69C000
        liAllDay.QuadPart = 0xC9;
        liAllDay.QuadPart=liAllDay.QuadPart << 32;
        liAllDay.QuadPart |= 0x2A69C000;

        SYSTEMTIME st;
        GetLocalTime(&st);     // узнаем текущую дату/время
        iFlag = st.wHour > HOUR; // если назначенный час еще не наступил, то ставим будильник на
сегодня, иначе - на завтра
        st.wHour = HOUR;
        st.wMinute = 0;
        st.wSecond =0;

        FILETIME ft;
        SystemTimeToFileTime( &st, &ft);
        if (iFlag)
                ((LARGE_INTEGER *)&ft)->QuadPart  =
                        ((LARGE_INTEGER *)&ft)->QuadPart  +liAllDay.QuadPart ;
        LocalFileTimeToFileTime(&ft,&ft);

       // Устанавливаем таймер,
        // он будет срабатывать раз в сутки (24*60*60*1000ms),
        // начиная со следующего "часа пик" - HOUR
    SetWaitableTimer(hDayTimer, (LARGE INTEGER *) &ft, 24*60*60000, 0, 0, 0);

        do {
                dw = WaitForMultipleObjectsEx(2,h,FALSE,INFINITE,TRUE);

                if (dw == WAIT_OBJECT_0 +1) // сработал hDayTimer
                {
                        // Устанавливаем таймер, он будет вызывать callback ф-ию раз в секунду,
                        // начнет с текущего момента
                        SetWaitableTimer(hAlarmTimer, &liDueTime, 1000, TimerAPCProc, NULL, 0);
                        iRingCount=0;
                }
                if (dw == WAIT IO COMPLETION) // закончила работать callback ф-ия
                {
                        iRingCount++;
                        if (iRingCount==RINGS)
                                CancelWaitableTimer(hAlarmTimer);
                }
        }
        while (dw!= WAIT OBJECT 0); // пока не сработало hTerminateEvent крутимся в цикле

        // закрывае handles, выходим
        CancelWaitableTimer(hDayTimer);
        CancelWaitableTimer(hAlarmTimer);
        CloseHandle(hDayTimer);
        CloseHandle(hAlarmTimer);

    _endthreadex( 0 );
    return 0;
};

int main(int argc, char* argv[])
{

        // это событие показывае потоку когда надо завершаться
        hTerminateEvent = CreateEvent(NULL,FALSE,FALSE,NULL);

        unsigned uThreadID;
        HANDLE   hThread;
```

```
        // создаем поток
        hThread = (HANDLE) beginthreadex( NULL, 0, &ThreadFunc, 0, 0, &uThreadID);

        puts("Press any key to exit.");
        // ждем any key от пользователя для завершения программы
        getch();
        // выставляем событие
        SetEvent(hTerminateEvent);

    // ждем завершения потока
    WaitForSingleObject(hThread, INFINITE );

    // закрываем handle
    CloseHandle( hThread );

        return 0;
}
```

Критические секции. Синхронизация в пользовательском режиме.

Критическая секция гарантирует вам, что куски кода программы, огороженные ей, не будут выполняться одновременно. Строго говоря, критическая секция не является объектом ядра. Она представляет собой структуру, содержащую несколько флагов и какой-то (не важно) объект ядра. При входе в критическую секцию сначала проверяются флаги, и если выясняется, что она уже занята другим потоком, то выполняется обычная wait-функция. Критическая секция примечательна тем, что для проверки, занята она или нет, программа не переходит в режим ядра (не выполняется wait-функция) а лишь проверяются флаги. Из-за этого считается, что синхронизация с помощью критических секций наиболее быстрая. Такую синхронизацию называют «*синхронизация в пользовательском режиме*».

Пример 9.
Снова рассмотрим очередь элементов. Один из вариантов ее реализации - двусвязный список. С точки зрения многопоточности, опасными являются операции добавления и удаления элементов из очереди. Существует вероятность, что несколько потоков одновременно начнут перестраивать указатели и связность очереди нарушится. Чтобы этого избежать, используем критическую секцию.

```
typedef … ItemData;

// Элемент очереди: данные и два указателя на предыдущий и следующий элементы
typedef struct _ItemStruct
{
    ItemData data;
    struct  ItemStruct * prev,*next;
} Item;

// Описание класса "Очередь"
class CMyQueue
 {
       CRITICAL SECTION m crisec;    // Критическая секция
       Item * m Begin;               // Указатель на первый элемент
       Item * m End;                 // Указатель на последний элемент
       int    m_Count;               // Количество элементов

  public:
        CMyQueue()
        {
              // Инициализируем критическую секцию
              InitializeCriticalSection(&m_crisec);
              // Инициализируем переменные
              m Count = 0;
              m Begin = m End = NULL;
        }

        ~CMyQueue()
        {
              // Удаляем все элементы очереди
              while(m Count) GetItem();
              // Удаляем критическую секцию
              DeleteCriticalSection(&m_crisec);
        }
```

```
void AddItem(ItemData data)
	{
		Item * NewItem;
		Item * OldFirstItem;

		NewItem = new Item();			// New item
		NewItem->next = NULL;
		NewItem->prev = NULL;
		NewItem->data = data;

		// ------------------------ Этот кусок не может выполняться параллельно
		EnterCriticalSection(&m_crisec);	// Заходим в к.с. (ждем входа)
		OldFirstItem = m Begin;
		m Begin = NewItem;
		NewItem->next = OldFirstItem;

		if (OldFirstItem)
			OldFirstItem->prev = NewItem;
		else
			m End = NewItem;

		m_Count++;
		LeaveCriticalSection(&m_crisec);	// Выходим из к.с.
		// ------------------------ Этот кусок не может выполняться параллельно
	}

	ItemData GetItem()
	{
		ItemData data;

		// ------------------------ Этот кусок не может выполняться параллельно
		EnterCriticalSection(&m crisec);	// Заходим в к.с. (ждем входа)
		if (!m_End)
			data = NULL;
		else
		{
			data = m End->data ;
			if (m_End->prev )
			{
				m_End->prev ->next = NULL;
				m End = m End->prev ;
			}
			else
			{
				m_End = NULL;
				m_Begin = NULL;
			}
			m Count --;
		}

		LeaveCriticalSection(&m_crisec);	// Выходим из к.с.
		// ------------------------ Этот кусок не может выполняться параллельно
		return data;
	};
};
```

Синхронизация процессов

Описатели объектов ядра зависимы от конкретного процесса (process specific). Проще говоря, handle объекта, полученный в одном процессе, не имеет смысла в другом. Однако существуют способы работы с одними и теми же объектами ядра из разных процессов.

Во-первых, это наследование описателя. При создании объекта можно указать будет ли его описатель наследоваться дочерними (порожденными этим процессом) процессами.

Во-вторых, дублирование описателя. Функция DuplicateHandle дублирует описатель объекта одного процесса в другой, т.е. по сути, берет запись в таблице описателей одного процесса и создает ее копию в таблице другого.

И, наконец, именование объекта ядра. При создании объекта ядра для синхронизации (мьютекса, семафора, ожидаемого таймера или события) можно задать его имя. Оно должно быть уникальным в системе. Тогда другой процесс может открыть этот объект ядра, указав в функции Open…(OpenMutex, OpenSemaphore, OpenWaitableTimer, OpenEvent) это имя.

На самом деле, при вызове функции Create... () система сначала проверяет, не существует ли уже объект ядра с таким именем. Если нет, то создается новый объект. Если да, ядро проверяет тип этого объекта и права доступа. Если типы не совпадают или же вызывающий процесс не имеет полных прав на доступ к объекту, вызов Create… функции заканчивается неудачно и возвращается NULL. Если все нормально, то просто создается новый описатель (handle) существующего уже объекта ядра. По коду возврата функции GetLastError() можно понять что произошло: создался новый объект или Create() вернула уже существующий.

Поэтому, синхронизировать потоки внутри разных процессов можно точно также как и в пределах одного. Нужно только правильно передать описатель синхронизирующего объекта от одного процесса к другому любым из перечисленных выше способов.

Пример 10. Многие приложения при запуске проверяют, запущен ли еще один экземпляр этой программы. Стандартный способ реализации этой проверки – использование поименованного Мьютекса.

```
int main(int argc, char* argv[])
{
        HANDLE Mutex;

        Mutex = CreateMutex(NULL,FALSE,"MyMutex");
        if (ERROR ALREADY EXISTS == GetLastError()) // Такой мьютекс уже кем-то создан...
        {
                MessageBox(0,"Приложение уже запущено","Error",0);
                CloseHandle( Mutex);
                exit(0);
        }
}
```

Взаимодействие между процессами

Потоки одного процесса не имеют доступа к адресному пространству другого процесса. Однако существуют механизмы для передачи данных между процессами.

Разделяемая память

Как уже говорилось, система виртуальной памяти в Win32 использует файл подкачки - swap file (или файл размещения - page file), имея возможность преобразования страниц оперативной памяти в страницы файла на диске и наоборот. Система может проецировать на оперативную память не только файл размещения, но и любой другой файл. Приложения могут использовать эту возможность. Это может использоваться для обеспечения более быстрого доступа к файлам, а также для совместного использования памяти.

Такие объекты называются *проекциями файлов* (на оперативную память) (file-mapping object). Для создания проекции файла сначала вызывается функция CreateFileMapping(). Ей передается дескриптор (уже открытого) файла или указывается, что нужно использовать page file операционной системы. Кроме этого, в параметрах ей передается флаг защиты, максимальный размер проекции и имя объекта. Затем вызывается функция MapViewOfFile(). Она отображает представление файла (view of a file) в адресное пространство процесса. По окончании работы вызывается функция UnmapViewOfFile(). Она освобождает память и записывает данные в файл (если это не файл подкачки). Чтобы записать данные на диск немедленно, используется функция FlushViewOfFile(). Проекция файла, как и другие объекты ядра, может использоваться другими процессами через наследование, дублирование дескриптора или по имени.

Пример 11. Вот пример программы, которая создает проекцию в page file и записывает в нее данные.

```
#include <windows.h>
void main()
{
        HANDLE hMapping;
        char*     lpData;
        char*     lpBuffer;
        ….
        //Создание или открытие существующей проекции файла
        hMapping = CreateFileMapping( (HANDLE)(-1), // используем page file
                NULL, PAGE READWRITE, 0, 0x0100, “MyShare”);
        if (hMapping == NULL)  exit(0);
        // Размещаем проекцию hMapping в адресном пространстве нашего процесса;
        // lpData получает адрес размещения
        lpData = (char*) MapViewOfFile(hMapping, FILE MAP ALL ACCESS,0,0,0);
        if (lpData == NULL) exit(0);
        // Копируем в проекцию данные
        memcpy ( lpData , lpBuffer );
        ….
        // Заканчиваем работу. Освобождаем память.
        UnmapViewOfFile(lpData);
        // Закрываем объект ядра
        CloseHandle(hMapping);
};
```

Прочие механизмы (сокеты, pipe)

Кроме разделяемой памяти, в Windows есть и другие способы передачи информации между процессами, например, каналы, поименованные каналы и сокеты. Все они имеют сходный принцип и представляют собой своеобразный канал или соединение, «трубу», соединяющую процессы. Программа, имея один конец такого соединения, может читать и\или писать в него данные, обмениваясь таким образом информацией с программой на другом конце.

Каналы используются для пересылки данных в одном направлении между дочерним и родительским процессами или между двумя дочерними процессами. Операции чтения\записи в канал похожи на подобные операции при работе с файлами.

Поименованные каналы используются для двустороннего обмена данными между процессом-сервером и одним или несколькими процессами-клиентами. Как и анонимные каналы, они используют файлоподобный интерфейс, но, в отличие от первых, пригодны также для обмена данными по сети.

Сокет – это абстрактный объект для обозначения одного из концов сетевого соединения, в том числе и через Internet. Сокеты Windows бывают двух типов: сокеты дейтаграмм и сокеты потоков. Интерфейс Windows Sockets (WinSock) основан на BSD-версии сокетов, но в нем имеются также расширения, специфические для Windows.

Сообщения в Windows (оконные сообщения)

Говоря о Windows нельзя не упомянуть о таких понятиях как windows (окна), messages (сообщения), message queue (очередь сообщений) и т.д.

Window – это (прямоугольная) область экрана в которой приложение отображает информацию (если оно видимо, конечно) и получает информацию от пользователя. Окна принадлежат потокам. Поток, создавший окно считается владельцем этого окна. Поток может быть владельцем нескольких окон.

Окна управляются сообщениями. Все события, происходящие с окном, сопровождаются посылкой ему сообщений: создание и уничтожение окна, ввод с клавиатуры, перемещение мыши, перерисовка и перемещение окна и т.д. Сообщения окну могут посылаться как самой системой, так и пользовательскими приложениями. Каждому окну приписана функция, называемая оконной процедурой (*window procedure*), которая и вызывается при обработке сообщения.

Сообщения можно посылать не только окну, но и самому потоку. Каждый поток, владеющий окном, имеет очередь сообщений. Как правило, поток, владеющий окнами, только тем и занимается, что обрабатывает сообщения, посылаемые его окнам.

Если описатели объектов ядра процессо-зависимы, то описатели окон уникальны в пределах Deskop. Поэтому одному процессу не составляет никакого труда получить и использовать описатель окна принадлежащему потоку другого процесса.

Посылка же сообщений из одного приложения другому есть не что иное, как один из способов межпроцессного общения.

Пример 12. Программа находит окно с заголовком “Калькулятор” и закрывает его, посылая сообщение WM_CLOSE.

```
#include <windows.h>
int main(int argc, char* argv[])
{
        HWND hwnd = FindWindow( NULL , “Калькулятор”);
        if (NULL != hwnd) PostMessage(hwnd, WM_CLOSE, 0, 0 );
        return 0;
}
```

Тема оконных сообщений в Windows весьма обширна и требует для рассмотрения, по крайней мере, отдельной статьи.

Заключение

В этой статье коротко были рассмотрены основные моменты многозадачной работы в Windows: создание и завершение процессов, синхронизация и межпроцессное общение.

Статья ни в коем случае не претендует на полноту и предназначена дать лишь общие сведения или указать направление для поиска нужной информации в огромной документации. Чтобы узнать более подробно о приведенных в статье объектах и функциях, читайте первоисточники:

- Microsoft Platform SDK
- Jeffrey Richter. Programming Applications for Microsoft® Windows. ISBN 1-57231-996-8
- Соломон, Руссинович. Внутреннее устройство MS Windows 2000. ISBN 5-7502-0136-8

Статья написана специально для http://www.uinc.ru

Автор: TN