Оглавление

ЛАБОРАТОРНАЯ РАБОТА ОБЪЕКТЫ (ПРОГРАММИРОВАНИЕ)

Теория

Без четкого понимания, что такое объекты не стать профессионалом в области разработки Windows программ. Объекты используются ОС (исполнительной системой и ядром) и приложениями для управления различными ресурсами: процессами, потоками, файлами и т.д.

Объектами управляет **диспетчер объектов** (**Object Manager**) исполнительной системы. Исполнительная система создает объект, когда она выделяет ресурсы процессу. Приложение не может получить прямой доступ к объекту, а только через соответствующий интерфейс. Кроме того, объект размещается в системной области и пользовательские программы не могут напрямую обращаться к памяти, в которой объект расположен.

Что такое объект

Как разработчик Windows приложений вы будите постоянно выполнять различные операции с объектами. Типы используемых объектов частично уже упоминались, но приведем некоторые из них еще раз:

- **Object Directory** – тип объекта, который включает набор других объектов. Исполнительная система использует этот тип для организации набора объектов.
- **Symbolic link** – ссылка на другой объект по его символическому имени. Этот тип используется для поддержки такой возможности.
- **Process –** тип объекта, который представляет процесс**.**

- **Thread** – тип объекта, который представляет поток.
- **Section** – тип объекта, который используется для реализации разделяемой памяти, используемой в адресных пространствах процессов.
- **File Port** – тип объекта, который представляет описатель файла, когда файл открыт для использования.
- **Access Token** – тип объекта, который использует Security Reference, Monitor Local Security Authority (LSA) для аутентификации пользователя и другие части системы защиты.
- **Event** – класс объектов, которые могут перехватывать определенное событие (event) в системе, таким образом, что другие части системы смогут выполнить определенные действия, когда они уверены, что событие произошло. Объект событие это основа для многих операций синхронизации.
- **Semaphore** – классические семафоры (Dijkstra).
- **Mutex** – тип объекта для синхронизации, используемый для взаимно исключающего доступа к критическим секциям.
- **Timer** – тип объекта, который используется для уведомления потока о том, что заданное время истекло.

Каждый объект содержит данные, определяемые типом, и стандартный *заголовок* (*title*) с которым работает **диспетчер объектов** (**Object Manager).** В заголовке содержится имя объекта, атрибуты безопасности и т.д.

Каждый объект – это просто блок памяти, выделенный в системной области памяти. Этот блок представляет структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа открытых описателей и д.р.) присутствует во всех объектах (заголовок), но большая часть специфична для объектов конкретного типа.

Приложение не может найти и модифицировать содержимое объектов. Для этого в Windows предусмотрен набор функций. Мы получаем доступ к

объектам только через эти функции. Когда вы вызываете функцию, создающую объект, она возвращает описатель, идентифицирующий созданный объект. Далее этот описатель может использовать любой поток вашего процесса, чтобы сообщить системе какой объект вас интересует.

Для надежности ОС Microsoft сделала так, чтобы значения описателей зависели от конкретного процесса.

Таблица описателей объектов

При инициализации процесса система создает в нем таблицу описателей объектов. Сведения о структуре этой таблицы плохо документированы, но квалифицированный программист должен понимать, как устроена таблица описателей объектов в процессе. Как видно из Таблица 1 Структура таблицы описателей объектов процесса, это просто массив структур данных. Каждая структура содержит указатель на какой-нибудь объект, маску доступа и некоторые флаги.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа	Флаги
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
...	...	...	...

Таблица 1 Структура таблицы описателей объектов процесса

Создание объекта

Когда процесс создается, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект, как диспетчер объектов выделяет для этого объекта блок памяти и инициализирует его. Дальше диспетчер объектов просматривает таблицу описателей процесса и ищет в ней первую свободную запись. Поскольку таблица еще пуста, то будет инициализирована структура с индексом 1. Указатель будет установлен на

внутренний адрес структуры объекта, маска доступа – на доступ без ограничений.

Вот некоторые функции, создающие объекты ядра:

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreationFlags,
    PDWORD pdwThreadId);
HANDLE CreateFile(
    PCTSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMade,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDistribution,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
DWORD dwMaximumSizeHigh,
DWORD dwMaximumSizeLow,
    PCTSTR pszFileName);
HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszFileName);
```

Все функции, создающие объекты, возвращают описатели, которые привязаны к конкретному процессу и могут быть использованы в любом потоке данного процесса. Значение описателя представляет собой индекс в таблице описателей процесса, и таким образом идентифицирует место, где хранится объект. В Windows 2000 это значение определяет не индекс, а байтовое смещение нужной записи от начала таблицы.

Всякий раз, при вызове функции, которая в качестве аргумента использует описатель объекта, вы передаете ей значение, возвращаемое одной из ***Create<Class>*** функций. При этом функция смотрит в таблицу описателей объектов вашего процесса и считывает адрес нужного объекта.

Если вы передаете неверный описатель, то функция завершается с ошибкой и ***GetLastError*** возвращает ERROR_INVALID_HANDLE. Это может быть связано с тем, что описатели представляют собой индексы в таблице объектов процесса и недействительны в других процессах.

Если объект создать не удалось, то обычно возвращается NULL (0). Это может произойти при нехватке памяти или проблем с защитой. Но некоторые функции возвращают в таком случае не NULL, а INVALID_HANDLE_VALUE (-1). Например, если ***CreateFile*** не сможет открыть указанный файл, то она вернет INVALID_HANDLE_VALUE. **Поэтому будьте очень внимательны при проверке значений, возвращаемых функциями, создающими объекты.**

```
HANDLE hMutex = CreateMutex(…);
If ( hMutex == INVALID_HANDLE_VALUE )
{
// этот код не будет выполнен никогда
// при ошибке CreateMutex возвращает NULL
}
HANDLE hFile = CreateFile(…);
If (hFile == NULL)
{
```

```
// этот код не будет выполнен никогда
// при ошибке CreateFile возвращает INVALID_HANDLE_VALUE
}
```

Закрытие объекта

Независимо от того, как именно вы создали объект, по окончании работы с ним его нужно закрыть вызвав ***CloseHandle***:

BOOL ***CloseHandle***(HANDLE hObj);

Эта функция сначала проверяет таблицу описателей вызывающего процесса, чтобы убедиться, идентифицирует ли переданный ей описатель объект, к которому этот процесс действительно имеет доступ. Если передан правильный индекс, то система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик открытых описателей; как только счетчик будет равен 0, объект будет удален из памяти.

Если описатель неверен, то происходит одно из двух:

- в нормальном режиме ***CloseHandle*** возвращает FALSE, а GetLastError – ERROR_INVALID_HANDLE;
- в режиме отладки система просто уведомляет об ошибке.

Перед самым возвратом управления ***CloseHandle*** удаляет соответствующую запись из таблицы описателей. После этого данный описатель недействителен в вашем процессе, и использовать его нельзя. После вызова ***CloseHandle*** вы больше не получите доступ к этому объекту ядра; но если счетчик числа открытых описателей не обнулен, то объект останется в памяти. Это означает, что объект используется другим процессом или процессами. Когда и они завершат с ним работу, вызвав ***CloseHandle***, он будет уничтожен.

Учет объектов

Объекты принадлежат системе, а не процессу. Например, если ваш процесс создает объект, а затем завершается, объект может быть не уничтожен. Если

созданный вами объект используется другим процессом, то система запретит его разрушение до тех пор, пока от него не откажется и тот процесс.

Системе известно, сколько процессов использует конкретный объект, так как в заголовке каждого объекта имеется *счетчик открытых описателей*. В момент создания объекта счетчику присваивается 1. Когда к существующему объекту обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, все его счетчики автоматически уменьшаются на 1. Как только счетчик какого-либо объекта обнуляется, этот объект уничтожает.

Защита объектов

Вызовы функций ***Create<Class>*** или ***Open<Class>*** использовались для запроса системных ресурсов, для того, чтобы объект был размещен, а описатель возвращен вызывающему потоку. Механизм защиты **диспетчера объектов** требует, чтобы при вызове этих функций были заданы *права доступ*. Существует набор общих прав доступа для всех объектов (чтение и запись), также есть типозависимые права (например, отложить доступ к вновь созданному процессу). При вызове ***Open<Class>*** **диспетчер объектов** проверяет права доступа, используя SECURITY_DESCRIPTOR и если возможно, предоставляет желаемый доступ.

Механизм безопасности основан на возможности аутентифицировать пользователя, который запустил программу. Процесс WINLOGIN аутентифицирует пользователей, когда они регистрируются в системе. Когда **диспетчера объектов** создает объект, он создает дескриптор безопасности, который включает идентификатор владельца, дискретный и системный списки контроля доступа. Дискретный список контроля доступа это список процессов и их прав для объекта, а системный список контроля доступа это список процессов и их прав, которые должны быть записаны в журнал безопасности при использовании для последующего аудита.

Если доступ разрешен, то Security Reference Monitor возвращает точный набор прав доступа, которые гарантируются вызывающему процессу и

диспетчера объектов сохраняет эти права как часть описателя объекта в таблице описателей объектов процесса. Когда поток процесса использует описатель, его права доступа к данному объекту сравниваются с гарантированным доступом, перед тем как к объекту будет предоставлен запрашиваемый доступ.

В лабораторной работе 2 мы уже использовали ***CreateProcess*** для создания нового процесса.

```
BOOL CreateProcess(
        …
        LPSECURITY_ATTRIBUTES lpProcessAttributes,
        // атрибуты безопасности процесса ( указатель )
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        // атрибуты безопасности потока ( указатель )
        …
            );
```

lpProcessAttributes и lpThreadAttributes это указатели на структуру данных SECURITY_ATTRIBUTES.

```
typedef struct_SECURITY_ATRIBUTES { //sa
        DWORD nLength;
        LPVOID lpSecurityDescriptor;
        BOOL bInheritHandle;
                        } SECURITY_ATTRIBUTES;
```

lpSecurityDescriptor это указатель на структуру SECURITY_DESCRIPTOR, она не документирована, но ее поля можно прочитать или установить используя Win32 API (см. в MSDN SECURITY_DESCRIPTOR). Функции Win32 API ***GetSecurityDescriptorControl***, ***SetSecurityDescriptorDacl*** и другие позволяют установить значения полей в SECURITY_DESCRIPTOR. Таким образом, вызывающий поток может специфицировать желаемые права доступа, которые будут назначены объекту или использовать аутентифицированный доступ для существующего объекта.

Значение по умолчанию для атрибутов безопасности (в вызове ***CreateProcess***) NULL, это означает, что структура SECURITY_ATTRIBUTES при вызове не передается. **Диспетчер процессов** (**Process Manager**) и **диспетчер объектов** (**Object Manager**) интерпретируют NULL следующим образом: процесс ребенок (или поток) будет использовать SECURITY_DESCRIPTOR для существующего объекта или системное значение по умолчанию, если создается новый объект.

Если вы хотите, то можете ограничить доступ ребенка к его собственному объекту-процессу. Для этого необходимо создать SECURITY_ATTRIBUTE и выполнить вызов, чтобы список контроля доступа не давал возможности ребенку ссылаться на объект. Это необычно, когда вы ограничиваете доступ ребенка к его собственному объекту процессу. Один и тот же механизм атрибутов безопасности используется для всех объектов исполнительной системы файлов, секций и т.д.

Итак, объекты можно защищать, используя дескриптор защиты (security descriptor), который описывает, кто создал объект и кто имеет права доступа к нему.

Почти все функции, создающие объекты, принимают указатель на структуру SECURITY_ATTRIBUTES. Рассмотрим еще один пример.

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszFileName);
```

Чтобы ограничить доступ к созданному объекту, создадим дескриптор защиты и инициализируем структуру SECURITY_ATTRIBUTES следующим образом:

```
SECURITY_ATTRIBUTES sa;
```

```
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = pSD; // адрес инициализированной SD
sa.bInheritHandle = FALSE;
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE,
&sa,
    PAGE_READWRITE, 0, 1024, “MyFileMapping”);
```

Для получения доступа к существующему объекту, необходимо указать какие операции вы собираетесь проводить с ним. Например, если мы хотим считать данные из существующей проекции файла, то надо вызвать функцию ***OpenFileMapping*** так:

```
HANDLE hFileMapping=OpenFileMapping(FILE_MAP_READ, FALSE,
                    “MyFileMapping”);
```

Передавая первым параметром FILE_MAP_READ, мы просим предоставить доступ к проекции файла для чтения. Функция ***OpenFileMapping*** прежде чем вернуть описатель, проверит тип защиты объекта. Если доступ разрешен, то функция возвратит действительный описатель, а если в доступе отказано, то NULL, а вызов ***GetLastError*** вернет код ошибки ERROR_ACCESS_DENIED (5).

Работа с объектами

Когда в лабораторной работе 2 ваш поток использовал ***CreateProcess*** и ***CreateThread*** для создания другого процесса или потока, он передавал имя **диспетчеру объектов** (Object Manager). При этом возвращались описатель (HANDLE) и идентификатор. В случае ***CreateProcess*** описатели нового процесса и нового потока, а также их идентификаторы заносятся в структуру данных PROCESS_INFORMATION. Функция ***CreateThread*** также возвращает описатель созданного потока и его идентификатор. После того как пользовательский поток получил описатель, он может передавать его исполнительной системе как параметр для других запросов. Например, после того как один поток создал другой поток, он передает описатель обратно исполнительной системе, закрывая описатель потока.

ChildThreadHandle = ***CreateThread***(…);

…

CloseHandle(ChildThreadHandle);

Когда **диспетчеру объектов** обрабатывает первый вызов, который ссылается на объект, он:

- добавляет имя объекта к набору используемых имен объектов;
- создает объект исполнительной системы с телом и заголовком;
- инициализирует поля заголовка;
- передает объект другим компонентам исполнительной системы, для заполнения тела типозависимой информацией.

Исполнительная система может определить, что такой объект уже существует, так как другой процесс (или другой поток) его уже создал. В этом случае *счетчик открытых описателей* в заголовке объекта увеличивается на 1, указывая, что теперь два описателя ссылаются на этот объект. Операции “открыть” приводят к увеличению значения счетчика на 1, а “закрыть” к уменьшению значения счетчика на 1. Когда значение счетчика становится равным 0, это означает, что больше не существует пользовательских описателей, которые ссылаются на этот объект. В этом случае **диспетчеру объектов** удаляет имя объекта из пространства имен.

Кроме того, компоненты исполнительной системы могут также ссылаться на объект, но для этого им нет необходимости использовать описатель (они могут использовать адрес объекта, так как они функционируют в режиме ядра). Поэтому заголовок объекта содержит также *счетчик ссылок* для хранения числа всех ссылок на объект (см. Рис. 1 Счетчики описателей и ссылок).

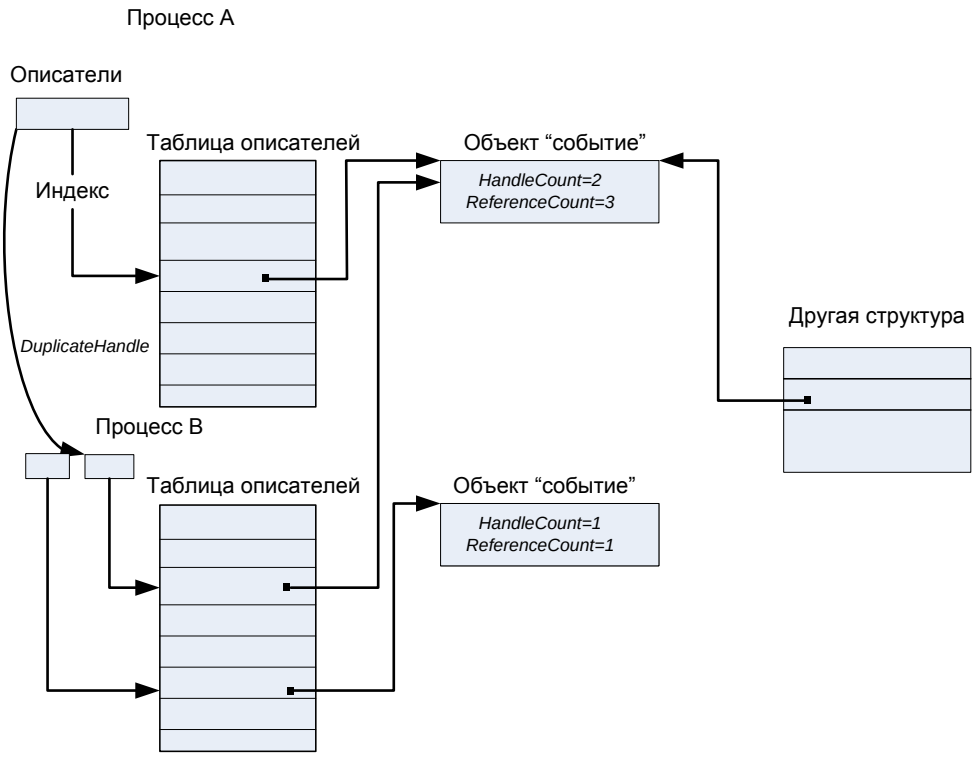


Рис. 1 Счетчики описателей и ссылок

Операции “открыть” в пользовательском режиме или в режиме ядра приводят к увеличению значения счетчика на 1, а “закрыть” к уменьшению значения счетчика на 1. Когда значение счетчика становится равным 0, это означает, что объект не используется никакими программными компонентами (пользовательскими или ядра) и можно освободить память, которую он занимал.

Какие отношения существуют между объектом ядра, описателем объекта и его идентификатором? Как показано на Рис. 2 Описатели и таблица описателей создается исполнительной системой и ядром и хранится в области памяти ядра, поэтому пользовательский поток не может получить к нему прямой доступ. Описатель это 32 битная ссылка на объект, определенная для процесса. Это смещение в таблице описателей объектов

процесса. Когда исполнительной системе нужно создать описатель объекта, она сначала ищет в таблице описателей процесса свободную строку. Затем заносит в первое поле адрес объекта в области ядра, во второе разрешения на доступ и т.д. Таким образом, возвращаемый описатель это индекс в таблице описателей объектов процесса, расположенной в области ядра. Описатель доступен только потокам, которые существуют в том же адресном пространстве.

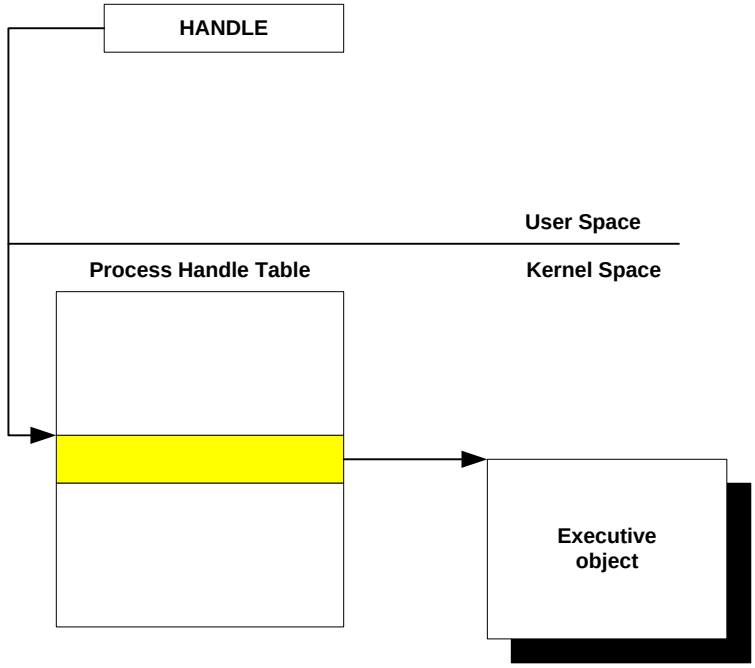


Рис. 2 Описатели и таблица описателей

Если необходимо идентифицировать объект в разных адресных пространствах, то вместо описателя можно использовать идентификатор.

Объект WaitableTimer

В этой лабораторной работе надо будет использовать *объект ожидаемый таймер* (*WaitableTimer*) для управления поведением потока. В лабораторной работе 2 предлагалось использовать ***GetSystemTime*** и разделяемую переменную runFlag для определения, когда остановить набор потоков. Один поток читал системное время и определял, не пора ли завершить работу, и если нет, то он блокировал себя на определенное время, а затем просыпался и

снова проверял. Попробуем сделать лучше, используя *ожидаемый таймер*. Этот тип объекта умеет периодически сообщать вашему процессу, что заданный интервал времени истек. Точность 100 ns.

```
HANDLE CreateWaitableTimer(
        LPSECURITY_ATTRIBUTES lpTimerAttributes,
        // указатель на атрибуты безопасности
        BOOL bManualReset, // флаг ручного перезапуска
        LPCTSTR lpTimerName // указатель на имя объекта таймера
                );
```

Атрибуты безопасности используются такие же, как в ***CreateProcess*** и ***CreateThread***. Задавая lpTimerName можно создать именованный объект или получить описатель уже существующего объекта (открыть). Если объект с таким именем уже существует, то ***GetLastError*** вернет ERROR_ALREADY_EXISTS. Параметр bManualReset контролирует число потоков, которые получат сигнал от *ожидаемого таймера*, когда он его пошлет. Если он TRUE, то все потоки, которые ждут таймер, получат уведомление, в противном случае только один. Функция возвращает описатель таймера.

Поток может получить описатель существующего таймера, используя ***OpenWaitableTimer***.

```
HANDLE OpenWaitableTimer(
        DWORD dwDesiredAccesss, // флаг доступа
        BOOL bInheritFlag, // флаг наследования
        LPCTSTR lpTimerName // указатель на имя объекта таймера
                );
```

Используя dwDesiredAccess, процесс может задать то, как он собирается использовать описатель таймера. Вы можете только получать уведомление от таймера, устанавливать период между посылаемыми уведомлениями или иметь полный доступ ко всем функциям таймера.

После того, как таймер создан, он должен быть установлен, перед тем как начнет выдавать уведомления. Для этого используется функция ***SetWaitableTimer***.

```
BOOL SetWaitableTimer(
        HANDLE hTimer, // описатель таймера
        Const LARGE_INTEGER *pDueTime,
        // когда таймер перейдет в состояние signaled
        LONG lPeriod, // периодический временной интервал
        PTIMERAPCROUTINE pfnCompletionRoutine,
        // указатель на процедуру (APC)
        LPVOID lpArgToCompletionRoutine,
        // данные, передаваемые в процедуру (APC)
        BOOL fResume // флаг состояния
                );
```

htimer это описатель, который возвращают функции ***CreateWaitableTimer*** и ***OpenWaitableTimer*** (при его создании или открытии).

pdueTime - это число в 64 битовом формате FILETIME.

```
typedef struct_FILETIME { //ft
        DWORD dwLowDateTime;
        // это младшие 32 бита времени в системном формате
        DWORD dwHighDateTime;
        // это старшие 32 бита
                } FILETIME;
```

Вы можете задать pDueTime как относительное или абсолютное время, отличить относительное время от абсолютного можно установив 64 битовое время отрицательным. lperiod задает время в милисекундах между уведомлениями. Если задано 0 значение, то таймер пошлет уведомление только один раз.

Следующие два параметра pfnCompletionRoutine и lpArgToCompletionRoutine здесь не рассматриваются. Флаг fResume необходим для особого режима

использования компьютера, в случае если компьютер завис, то если значение флага TRUE, при получении уведомления он будет перезагружен.

Теперь рассмотрим использование функции ***WaitForSingleObject***. Когда поток вызывает эту функцию, он будет заблокирован, пока не придет уведомление от указанного объекта. Поток может создать таймер, установить его и затем ждать от него уведомления, вызвав ***WaitForSingleObject.***

DWORD ***WaitForSignleObject***(

HANDLE hHandle; // описатель ожидаемого объекта

DWORD dwMilliseconds; // тайм-аут в милисекудах

);

У нас hHandle это описатель таймера. dwMilliseconds определяет максимальный промежуток времени, который поток желает ждать пока истечет время. Вы можете использовать ***GetLastError***, чтобы посмотреть, что возвращает функция:

- WAIT_OBJECT_O – если послано уведомление;
- или WAIT_TIMEOUT – истек тайм-аут.

Вы можете также задать INFINITE для ожидания уведомления без тайм-аута.

Скелет кода с использованием таймера.

```
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include <stdio.h>

#define _SECOND 10000000


void main( void )
{
	HANDLE wTimer;
```

```
    __int64 endTime;
    LARGE_INTEGER quitTime;
    SYSTEMTIME now;

    wTimer = CreateWaitableTimer(NULL,FALSE,NULL);
    endTime = -5 * _SECOND;
    quitTime.LowPart = (DWORD) (endTime & 0xFFFFFFFF);
    quitTime.HighPart = (LONG) (endTime >> 32);
    SetWaitableTimer(wTimer,&quitTime,0,NULL,NULL,FALSE);
    GetSystemTime(&now);
    printf("System Time %d %d %d\n",now.wHour,now.wMinute,now.wSecond);
    WaitForSingleObject(wTimer,INFINITE);
    printf("Waitable Timer sent signal\n");
    GetSystemTime(&now);
    printf("System Time %d %d %d\n",now.wHour,now.wMinute,now.wSecond);
    CloseHandle( wTimer );
}
```

Чему нужно научиться

Необходимо научиться работать с объектами.

Задания

Уровень 1 (A)

Написать программу, которая использует ожидаемый таймер (WaitableTimer), чтобы остановить себя через К секунд после старта, где К параметр командной строки.

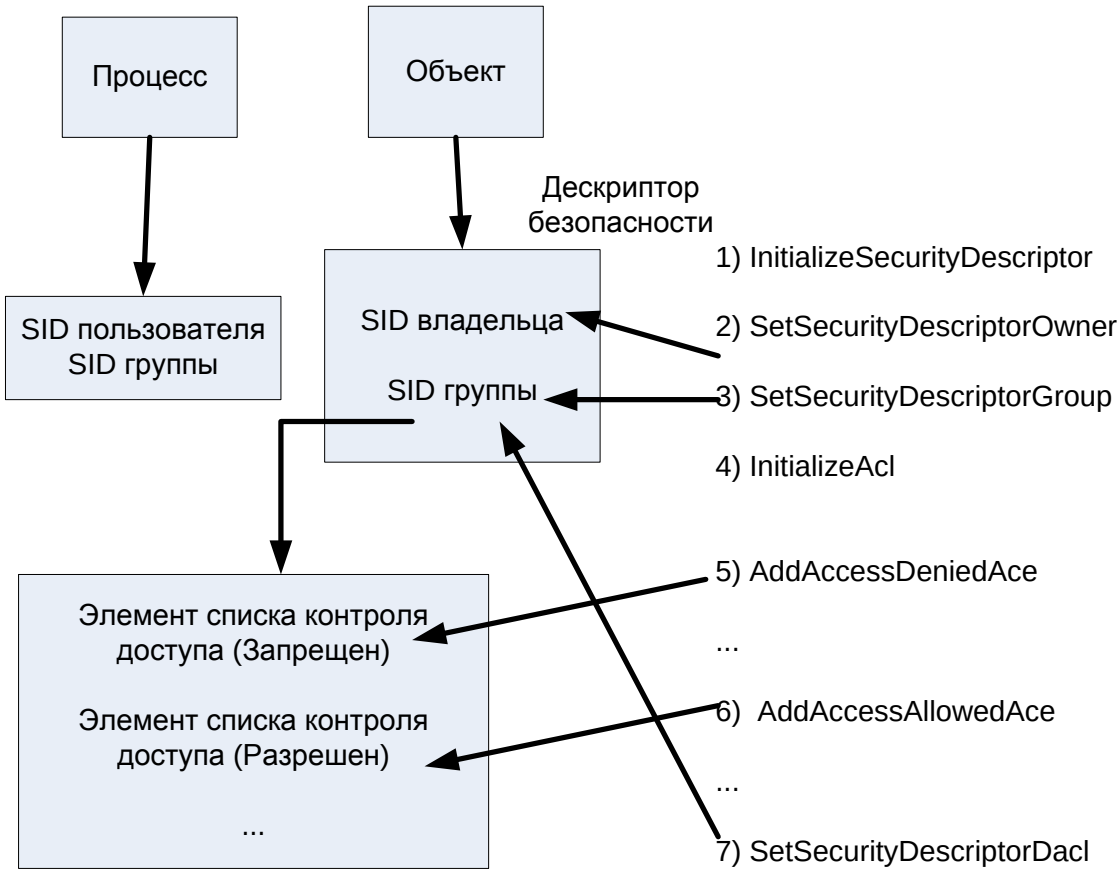
Уровень 2 (А)

Модифицировать программу первого уровня, чтобы она создавала N фоновых процессов, каждый запускал бы программу, которая заканчивалась бы в случайное время по своему собственному усмотрению (N – параметр командной строки). А после того, как истекут К секунд (К – параметр командной строки), она уничтожала процессы, которые не закончились сами. Если все фоновые процессы закончились, то главная программа сама выполняется К секунд, а потом заканчивается.

Уровень 3 (А)

Поработайте с правами доступа. Убедитесь в том, что проверка прав доступа осуществляется только при открытии описателя, а не при всяком его использовании. После того, как процесс успешно открыл описатель, предоставленные ему права доступа не отзываются подсистемой защиты, даже если изменился ACL объекта.

Используйте функции: ***GetSecurityInfo, SetSecurityInfo, GetNamedSecurityInfo, SetNamedSecurityInfo, LookupAccountSid, GetEffectiveRightsFromAcl, AddAccessAllowedAce, AddAccessDeniedAce*** и т.д.



Скелет кода фоновой программы

Хотим создать процесс, который может закончиться или не закончиться по своему собственному усмотрению.

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

int main(int argc, char *argv[])
{
	printf(“Client %s beginning to run\n”,argv[1]);
	while(TRUE)
	{
		if(getc(stdin)==’y’) break;
		getc(stdin);
	}
	return 0;
```

}

Для выполнения задания вам может понадобиться ***TerminateProcess***:

BOOL ***TerminateProcess***(

HANDLE hProcess, // описатель процесса

UINT uExitCode // код выхода для процесса

);

Один процесс может закончить другой процесс, но он должен иметь на это соответствующий допуск (PROCESS_TERMINATE) для этого описателя.