Тема Процессы

Обработка ошибок

```
DWORD GetLastError();
```

Пример формирования текста описания ошибки

```
// получаем код ошибки
DWORD dwError = GetLastError();

TCHAR * pstr = NULL; // адрес массива символов
// получаем текстовое описание ошибки
BOOL fOk = FormatMessage(
FORMAT_MESSAGE_FROM_SYSTFM |
FORMAT_MESSAGE_ALLOCATC_BUFFER,
NULL, dwError, MAKELANGID(LANG_ENGLISH,
SUBLANG_ENGLISH_US).
(LPTSTR) &pstr, 0, NULL);

if (pstr != NULL) { SetDlgItemText(hwnd,
    IDC_ERRORTEXT, (PCTSTR) pstr);
LocalFree(pstr);
} else {
SetDlgItemText(hwnd, IDC ERRORTEXT, TEXT("Error
description not found "));
}
```

Класс приоритета процесса

Класс приоритета	идентификатор
Idle (простаивающий)	***IDLE_PRIORITY_CLASS***
Below normal (ниже обычного)	***BELOW_NORMAL_PRIORITY_CLASS***
Normal (обычный)	***NORMAL PRIORITY CLASS***
Above normal (выше обычного)	***ABOVE_NORMAL_PRIORITY_CLASS***
High (высокий)	***HIGH_PRIORITY_CLASS***
Realtime (реального времени)	***REALTIME_PRIORITY_CLASS***

Завершение процесса

1. Процесс можно завершить одним из четырех способов:

- ***возврат управления входной функцией первичного потока*** *(безопасный способ);*
- ***один из потоков процесса вызывает функцию ExitProcess*** *(менее безопасный способ);*
- ***поток другого процесса вызывает функцию TerminateProcess*** *(потенциально опасный способ);*
- ***все потоки процесса завершаютя по своей воле*** *(редко наблюдаемое явление).*

2. Возврат управления входной функцией первичного потока

Это единственный способ, позволяющий выполнить корректное освобождение всех ресурсов, принадлежавших процессу. При этом:

- вызываются деструкторы для C++-объектов;
- система освобождает память, которую занимал стек потока;
- система устанавливает код завершения процесса;
- счетчик пользователей объекта ядра "процесс" уменьшается на 1.

3. Вызов функции ExitProcess

Прототип функции имеет вид

```
VOID ExitProcess(UINT fuExitCode).
```

В приложении, написанном на C/C++, следует избегать вызова этой функции, так как библиотеке C/C++ скорее всего не удастся провести должную очистку. Это иллюстрирует код:

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

class CSomeObj {
public:
   CSomeObj()  { printf("Constructor\r\n"); }
   ~CSomeObj() { printf("Destructor\r\n"); }
};

CSomeObj g_GlobalObj;

void main () {
   CSomeObj LocalObj;
   ExitProcess(0);     // этого здесь не должно быть

   // в конце функции main компилятор автоматически
   // вставил код для вызова деструктора LocalObj,
   // но ExitProcess не дает его выполнить.
}
```

При выполнении программы Вы увидите:

```
Constructor
Constructor
```

и не увидите

```
Destructor
Destructor
```

Код создает два объекта: глобальный и локальный. Но Вы никогда не увидите строку *Destructor* C++ – объекты не разрушаются должным образом из-за того, что *ExitProcess* форсирует уничтожение процесса и библиотека C/C++ не получает шанса на очистку.

4. Вызов функции TerminateProcess

Прототип функции имеет вид

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

Пользуйтесь *TerminateProcess* лишь в том случае, когда иным способом завершить процесс не удается. Процесс не получает абсолютно никаких уведомлений. При этом теряются все данные, которые процесс не успел переписать из памяти на диск.

5. Что происходит при завершении процесса

1. Выполнение всех потоков в процессе прекращается.
2. Все User- и GDI-объекты, созданные процессом, уничтожаются.
3. Закрываются дескрипторы объектов ядра, открытые процессом.
4. Код завершения процесса меняется со значения STILL_ACTIVE на код, переданный в *ExitProcess* или *TerminateProcess*.
5. Объект ядра "процесс" переходит в свободное (signaled) состояние.
6. Счетчик объекта ядра "процесс" уменьшается на 1.

6. Получение кода завершения процесса

```
BOOL GetExitCodeProcess( HANDLE hProcess, PDWORD
pdwExitCode);
```

Схема использования

```
DWORD dwExitCode;
TCHAR Message[255];
if(!GetExitCodeProcess(hProcess1, &dwExitCode)
{//Ошибка выполнения запроса
// Действия в случае ошибки вызова функции
}
else
     if (dwExitCode==STILL_ACTIVE)
          {// Процесс еще не завершился
          wsprintf(Message, "%s", "Процесс еще не
завершился");
          }
     else
          {// Процесс завершился
          wsprintf(Message, "Код завершения процесса
%d",dwExitCode);
          }
     MessageBox(hWnd1, Message, "Код завершения
процесса", MB_OK);
```

7. Дочерние процессы

Если Вы хотите создать новый процесс, заставить его выполнить какие-либо операции и дождаться их результатов, напишите примерно такой код

```
PROCESS_INFORMATION pi;
DWORD dwExitCode;

// порождаем дочерний процесс
BOOL fSuccess = CreateProcess(..., &pi};

if (fSuccess) {

// закрывайте описатель потока, как только
// необходимость в нем отпадает!
CloseHandle(pi.hThread);

// По условию задачи приостанавливаем выполнение
// родительского процесса пока не завершится
// дочерний процесс
WaitForSingleObject(pi.hProcess, INFINITE);

// дочерний процесс завершился; получаем код его
завершения
GetExitCodeProcess(pi.hProcess, &dwExitCode);

// закрывайте описатель процесса, как только
необходимость в нем отпадает!
CloseHandle(pi.hProcess);

}
```

8. Запуск обособленных дочерних процессов

Приведенный ниже фрагмент кода демонстрирует, как, создав процесс, сделать его обособленным. Это значит, что после создания и запуска нового процесса родительскому процессу нет необходимости с ним взаимодействовать.

```
PROCESS_INFORMATION pi;
BOOL fSuccess = CreateProcess( ... , &pi);
if (fSuccess) {
// закрывая дескрипторы мы разрешаем системе
// уничтожить обьекты ядра "процесс" и "поток"
// сразу после завершения дочернего процесса
CloseHandle(pi.hThread);
CloseHandle(pi.hProcess);
}
```

9. Перечисление процессов, выполняемых в системе

В процессе работы Windows ведет постоянно обновляемую базу данных Performance Data. В ней содержится информация, доступная через функции реестра вроде ***RegQueryValueEx****, для которой надо указать корневой раздел* ***HKEY_PERFORMANCE_DATA****.*

Чтобы упростить работу с этой базой данных, Microsoft создала для Windows NT набор функций под общим названием ***Performance Data Helper*** *(содержащийся в PDH.dll).*

В Windows 95 и Windows 98 такой базы данных нет. Вместо них предусмотрен набор функций, позволяющих перечислять процессы. Они включены в ***ToolHelp API*** *(библиотека toolhelp.dll)****.*** *Информацию можно найти в документации Platform SDK — ищите разделы по функциям* ***Process32First и Process32Next.***

Разработчики Windows NT для перечисления процессов создали свой набор функций под общим названием ***Process Status*** *(содержащийся в PSAPI.dll). Ищите в документации Platform SDK раздел по функции* ***EnumProcesses****.*

Функции ToolHelp включены в Windows 2000 и XP.

10. Псевдоописатели

Получение псевдоописателя не приводит к увеличению счетчика пользователей объекта.

Псевдоописатель процесса можно использовать при вызове функций, которым нужен описатель текущего процесса. Для его получения вызываем функцию

```
HANDLE GetCurrentProcess( );
```

Так, поток может запросить все показатели времен работы своего процесса, вызвав *GetProcessTimes*:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime,
ftUserTime;
GetProcessTimes(GetCurrentProcess(), &ftCreationTime,
&ftExitTime, &ftKernelTime, &ftUserTime);
```

11. Функция дублирования описателя

```
BOOL DuplicateHandle(
    HANDLE hSourceProcess, HANDLE hSource,
    HANDLE hTargetProcess, PHANDLE phTarget,
    DWORD fdwAccess,
    BOOL bInheritHandle,
    DWORD fdwOptions);
```

Функция ***DuphcateHandle*** позволяет преобразовать псевдоописатель процесса в настоящий описатель. Вот как это сделать:

```
HANDLE hProcess;

DuplicateHandle(
   GetCurrentProcess(),// Описатель исходного процесса
   GetCurrentProcess(),// Описатель для дублирования
   GetCurrentProcess(),// Описатель процесса назначения
   &hProcess, //Место для нового, настоящего описателя
   0,
   FALSE,  // Новый описатель процесса ненаследуемый
   DUPLICATE_SAME_ACCESS); // Новому описателю процесса
присваиваются те же атрибуты защиты, что и исходному
псевдоописателю
```

12. Время выполнения процесса

Время существования процесса это время от момента запуска процесса до момента его завершения. Это время можно определить следующим образом:

```
// В начале работы процесса
// получаем стартовое время
DWORD dwStartTime = GetTickCount();

// здесь выполняются потоки процесса

// Перед завершением процесса вычитаем
// стартовое время из текущего
DWORD dwElapsedTime = GetTickCount() - dwStartTime;
```

Получаем таким образом время существования процесса в миллисекундах.

Существует функция, позволяющая получить более детальную информацию о процессе:

```
BOOL GetProcessTimes(
HANDLE hProcess,
PFILETIHE pftCreationTime, PFILETIME pftExitTime,
PFILETIME pftKernelTime,
PFILETIME pftUserTime);
```

GetProcessTimes возвращает четыре временных показателя, которые представлены в следующей таблице:

Показатель времени	Описание
Creation time *(время создания)*	**Абсолютная величина**, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания первого потока процесса
Exit time *(время завершения)*	**Абсолютная величина**, выраженная в интервалах по 100нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента завершения последнего потока процесса. *Если процесс не завершился, этот показатель имеет неопределенное значение*
Kernel time *(время выполнения в*	**Относительная величина**, выраженная в интервалах по 100 нс. Сообщает время, затраченное

режиме ядра)	всеми потоками процесса на выполнение кода операционной системы в режиме ядра
User time *(время выполнения в режиме пользователя)*	**Относительная величина**, выраженная в интервалах по 100 нс. Сообщает время, затраченное всеми потоками процесса на выполнение кода приложения в режиме пользователя.

GetProcessTimes возвращает временные параметры, суммированные по всем потокам (даже уже завершенным) в указанном процессе. Например, время выполнения ядра будет суммой периодов времени, затраченного всеми **потоками процесса** на выполнение кода **операционной системы.**

GetThreadTimes имеет аналогичные параметры и возвращает данные по использованию процессорного времени для одного потока.

Пример. Определение времени, затраченного процессом для выполнения участка программы:

```
//Функция объединения двух 32-разрядных чисел в одно 64-разрядное
_int64 FileTimeToQuadWord(PFILETIME pft)
{
return(Int64ShllMod32(pft->dwHighDateTime, 32) |
pft->dwLowDateTime);
}

void PerformLongOperation ()
{

FILETIME ftKernelTimeStart, ftKernelTimeEnd;
FILETIME ftUserTimeStart, ftUserTimeEnd;
FILETIME ftDummy;

_int64 qwKernelTimeElapsed, qwUserTimeElapsed,
qwTotalTimeElapsed;

// получаем начальные показатели времени
GetProcessTimes(GetCurrentProcess(), &ftDummy,
&ftDummy, &ftKernelTimeStart, &ftUserTimeStart);

// здесь выполняем оцениваемый участок программы

- - -
```

```
// получаем конечные показатели времени
GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
&ftKernelTimeEnd, &ftUserTimeEnd);

// получаем значении времени, затраченного на
// выполнение в режиме ядра и пользователя,
// преобразуя начальные и конечные показатели
// времени из FILETIME в учетверенные слова,
// а затем вычитая начальные показатели из конечных
qwKernelTimeElapsed =
                FileTimeToQuadWord(&ftKernelTimeEnd)-
    FileT imeToQuadWord(&ftKernelTimeStart);

qwUserTimeElapsed = FileTimeToQuadWord(&ftUserTimeEnd)
- FileTimeToQuadWord(&ftUserTimeStart);

// получаем общее время, складывая время
// выполнения ядра и User
qwTotalTimeElapsed = qwKernelTimeElapsed +
qwUserTimeElapsed;

// общее время хранится в qwTotalTimeElapsed
}
```

СТРУКТУРЫ ДЛЯ РАБОТЫ ДАННЫМИ ВРЕМЕНИ И ДАТЫ

```
typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME, *PFILETIME;

typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;

typedef union _ULARGE_INTEGER {
struct {
      DWORD LowPart;  // 32 - разрядное поле
      DWORD HighPart; // 32 - разрядное поле
  };
ULONGLONG QuadPart; // 64-разрядное поле
} ULARGE_INTEGER, *PULARGE_INTEGER;

typedef union _LARGE_INTEGER {
struct {
      DWORD LowPart; // 32 - разрядное поле
      LONG  HighPart;  // 32 - разрядное поле
  };
LONGLONG QuadPart; // 64-разрядное поле
} LARGE_INTEGER, *PLARGE_INTEGER;
```

Определение относительного времени (длительность интервала)

Для определения **относительного времени** (разность двух отсчетов времени) поступаем следующим образом:

1. Преобразуем структуры ***SYSTEMTIME*** в структуры ***FILETIME***;
2. Копируем значения из полей dwLowDateTime и dwHighDateTime структур ***FILETIME*** в поля LowPart и HighPart объединений ***ULARGE_INTEGER***;
3. Применяем операции 64-разрядной арифметики к значениям полей QuadPart объединений ***ULARGE_INTEGER***;
4. Выполняем обратное копирование

Некоторые из функций преобразования формата времени

```
VOID GetSystemTime(
  LPSYSTEMTIME lpSystemTime);

BOOL FileTimeToLocalFileTime(
  CONST FILETIME *lpFileTime,
  LPFILETIME lpLocalFileTime
);

BOOL LocalFileTimeToFileTime(
  CONST FILETIME *lpLocalFileTime,
  LPFILETIME lpFileTime
);
```

BOOL **FileTimeToSystemTime**(

CONST FILETIME *lpFileTime,

LPSYSTEMTIME lpSystemTime

);

Функции работы с данными времени и даты

Функция	Описание
CompareFileTime	Compares two 64-bit file times.
DosDateTimeToFileTime	Converts MS-DOS date and time values to a 64-bit file time.
FileTimeToDosDateTime	Converts a 64-bit file time to MS-DOS date and time values.
FileTimeToLocalFileTime	Converts a UTC file time to a local file time.
FileTimeToSystemTime	Converts a 64-bit file time to system time format.
GetFileTime	Retrieves the date and time that a file was created, last accessed, and last modified.
GetLocalTime	Retrieves the current local date and time.
GetSystemTime	Retrieves the current system date and time in UTC format.
GetSystemTimeAdjustment	Determines whether the system is applying periodic time adjustments to its time-of-day clock.
GetSystemTimeAsFileTime	Retrieves the current system date and time in UTC format.
GetTickCount	Retrieves the number of milliseconds that have elapsed since the system was started.
GetTimeZoneInformation	Retrieves the current time-zone parameters.
LocalFileTimeToFileTime	Converts a local file time to a file time based on UTC.
SetFileTime	Sets the date and time that a file was created, last accessed, or last modified.
SetLocalTime	Sets the current local time and date.
SetSystemTime	Sets the current system time and date.
SetSystemTimeAdjustment	Enables or disables periodic time adjustments to the system's time-of-day clock.
SetTimeZoneInformation	Sets the current time-zone parameters.
SystemTimeToFileTime	Converts a system time to a file time.
SystemTimeToTzSpecificLocalTime	Converts a UTC time to a specified time zone's corresponding local time.