Потоки

Содержание

Потоки

1 Создание потока. Функция CreateThread

2 Завершение потока

Что происходит при завершении потока

Возврат управления функцией потока

Поток самоуничтожается

Поток уничтожается

Получение кода завершения потока

3 Приостановка и возобновление работы потока

4 Пример приостановки процесса, путем приостановки всех его потоков

5 Переключение потоков

6 Определение периодов выполнения потока

7 Структуры FILETIME и SYSTEMTIME для даты и времени

8 Определение относительного времени

9 Пример создания и уничтожения потоков

10 Преобразование псевдоописателя в настоящий описатель

1 Создание потока. Функция CreateThread

```
HANDLE CreateThread(
PSECURITY_ATTRIBUTES psa, DWORD cbStack,
PTHREAD_START_ROUTINE pfnStartAddr, PVOID
pvParam, DWORD fdwCreate, PDWORD pdwThreadID);
```

Windows — операционная система с вытесняющей многозадачностью. Следовательно, новый поток и поток, вызвавший *CreateThread*, могут выполняться одновременно.

Нельзя делать какие-либо предположения о последовательности выполнения потоков в системе.

В связи с этим возможны проблемы.

Остерегайтесь, например, такого кода.

```
DWORD WINAPI FirstThread(PVOID pvParam)
{
// инициализируем переменную, которая содержится
в стеке
int x = 0; //переменная в стеке первичн. потока
DWORD dwThreadID;
// создаем новый поток
HANDLE hThread = CreateThread(NULL, 0,
SecondThread, (PVOID) &x, 0, &dwThreadId);

// Мы больше не ссылаемся на новый поток,
// поэтому закрываем свой описатель этого потока
CloseHandle(hThread);
// Завершаем работу первичного потока.
return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
// здесь выполняется какая-то
// длительная обработка
- - -
// Пытаемся обратиться к переменной
// в стеке первичного потока FirstThread
* ((int *) pvParam) = 5; // это может привести
// к ошибке – нарушение доступа, если
// первичный поток уже завершился
return(0);
}
```

2 Завершение потока

Поток можно завершить следующими способами:

1) функция потока возвращает управление (рекомендуемый способ);
2) поток самоуничтожается вызовом функции ***ExitThread*** (возможный, но не полностью безопасный способ);

3) один из потоков данного или стороннего процесса вызывает функцию ***TerminateThread*** (потенциально опасный способ);
4) завершается процесс, содержащий данный поток (тоже не безопасно и нежелательно).

Что происходит при завершении потока

- Освобождаются все описатели User-объектов, принадлежавших потоку. В Windows большинство объектов принадлежит процессу, поток владеет только окнами и ловушками (hooks). Когда поток, создавший такие объекты, завершается, система уничтожает их автоматически. Прочие объекты разрушаются, только когда завершается владевший ими процесс.
- Код завершения потока меняется со STILL_ACTIVE на код, возвращенный функцией потока, либо переданный в функцию *ExitThread* или *TerminateThread*.
- Объект ядра "поток" переводится в свободное состояние.
- Счетчик пользователей объекта ядра "поток" уменьшается на 1.
- Если данный поток является последним активным потоком в процессе, завершается и сам процесс.

Возврат управления функцией потока

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление.

Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших потоку. При этом:

- объекты C++, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система корректно освобождает память, которую занимал стек потока;
- система устанавливает код завершения потока в значение, которое возвратила функция потока;
- счетчик пользователей объекта ядра "поток" уменьшается на 1.

Поток самоуничтожается

```
VOID ExitThread(DWORD dwExitCode);
```

Поток уничтожается

```
BOOL TerminateThread( HANDLE hThread,
DWORD dwExitCode);
```

Получение кода завершения потока

```
BOOL GetExitCodeThread( HANDLE hThread,
            PDWORD pdwExitCode);
```

3 Приостановка и возобновление работы потока

```
DWORD SuspendThread(HANDLE hThread);

DWORD ResumeThread(HANDLE hThread);
```

Если вызов *ResumeThread* прошел успешно, она возвращает предыдущее значение счетчика простоев данного потока; в ином случае — 0xFFFFFFFF.

Приостановка на заданное время

```
VOID Sleep(DWORD dwMilliseconds);
```

4 Пример приостановки процесса, путем приостановки всех его потоков

```
VOID SuspendProcess(DWORD dwProcessID, BOOL
fSuspend)
{
// получаем список потоков в системе
HANDLE hSnapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD,
dwProcessID);
if (hSnapshot != INVALID_HANDLE_VALUE) {
// просматриваем список потоков
THREADENTRY32 te;
ZeroMemory(&te, sizeof(THREADENTRY32));
te.dwSize = { sizeof(te) };
BOOL fOk = Thread32First(hSnapshot, &te);
for (; fOk; fOk = Thread32Next(hSnapshot, &te))
{
// относится ли данный поток к нужному процессу
if (te.th320wnerProcessID == dwProcessID)
    {
    // получаем описатель потока по
    идентификатору
    HANDLE hThread =
```

```
    OpenThread(THREAD_SUSPEND_RESUME, FALSE,
    te.th32ThreadID);
    if (hThread != NULL){
        // приостанавливаем или
        // возобновляем поток
        if (fSuspend)
            SuspendThread(hThread);
        else
            ResumeThread(hThread);
        }
    CloseHandle(hThread);
    }
}
CloseHandle(hSnapsnot);
}
}
```

Для перечисления списка потоков здесь используются ToolHelp функции (см. Рихтер, глава 4).

Для получения дескриптора потока нужного процесса, вызывем *OpenThread*.

```
HANDLE OpenThread( DWORD dwDesiredAccess, BOOL
bInheritHandle, DWORD dwThreadID);
```

Это функция появилась в Windows 2000. Она находит объект ядра "поток" по идентификатору, указанному в dwThreadID

5 Переключение потоков

Функция *SwitchToThread*позволяет подключить к процессору другой поток (если он есть):

```
BOOL SwitchToThread();
```

6 Определение периодов выполнения потока

Иногда нужно знать, сколько времени затрачивает поток на выполнение той или иной операции.

Простейший вариант:

```
// получаем стартовое время
DWORD dwStartTime = GetTickCount();

// здесь выполняем какой-нибудь сложный алгоритм

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

Подход дает длительность интервала.

Более подробную информацию можно получить с помощью

```
BOOL GetThreadTimes( HANDLE hThread,
PFILETIME pftCreationTime,
PFILETIME pftExitTime,
PFILETIME pftKernelTime,
PFILETIME pftUserTime);
```

GetThreadTimes возвращает четыре временных параметра:

Показатель времени	Описание
Время создания (creation time)	Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания потока
Время завершении (exit time)	Абсолютная величина, выраженная в интервалах по 100нс Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента завершения потока. Если поток все еще выполняется, этот показатель имеет неопределенное значение
Время выполнения ядра (kernel time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное этим потоком на выполнение кода операционной системы
Время выполнения User (User time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время,

	затраченное потоком на выполнение кода приложения.

Пример определения с помощью *GetThreadTimes* времени, необходимого для выполнения некоторого алгоритма:

```
//впомогательная функция для преобразования
//двух 32-разрядных элементов FILETIME
//в одно 64-разрядное значение
_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
GetThreadTimes(GetCurrentThread(), &ftDummy,
&ftDummy, &ftKernelTimeStart, &ftUserTimeStart);

// здесь выполняем сложный алгоритм

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

// получаем значении времени, затраченного на
// выполнение в режиме ядра и User,
// преобразуя начальные и конечные показатели
// времени из FILETIME
// в учетверенные слова, а затем вычитая
// начальные показатели из конечных
```

```
qwKernelTimeElapsed =
FileTimeToQuadWord(&ftKernelTimeEnd) -
FileTimeToQuadWord(&ftKernelTimeStart);

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

// получаем общее время, складывая время
// выполнения в режимах ядра и пользователя
qwTotalTimeElapsed = qwKernelTimeElapsed +
qwUserTimeElapsed;

// общее время хранится в qwTotalTimeElapsed
}
```

7 Структуры FILETIME и SYSTEMTIME для даты и времени

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

BOOL FileTimeToSystemTime(
  CONST FILETIME *lpFileTime,
  LPSYSTEMTIME lpSystemTime
);

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

VOID GetSystemTime(
  LPSYSTEMTIME lpSystemTime);

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

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

8 Определение относительного времени

Для определения относительного времени
поступаем следующим образом:

1) Преобразуем структуру **SYSTEMTIME**
   в структуру **FILETIME**.
2) Копируем структуру **FILETIME**
   в структуру **ULARGE_INTEGER**.
3) Применяем операции 64-разрядной арифметики к
   ULARGE_INTEGER значениям.

```
typedef union _ULARGE_INTEGER {
struct {
        DWORD LowPart;
        DWORD HighPart;
   };
ULONGLONG QuadPart;
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

```
typedef union _LARGE_INTEGER {
struct {
        DWORD LowPart;
        LONG  HighPart;
   };
LONGLONG QuadPart;
} LARGE_INTEGER, *PLARGE_INTEGER;
```

9 Пример создания и уничтожения потоков

```
- - -
hThrd = CreateThread(
NULL,  // атрибуты безопасности
0,     // размер стека по умолчанию
(LPTHREAD_START_ROUTINE) ThreadFunc,//ф-я потока
(LPVOID)hwndChildWnd, // параметр ф-ии потока
CREATE_SUSPENDED, // флаг – создать «спящий»
&IDThread);       // для идентификатора потока
if (hThrd == NULL)// вывод сообщения и выход
    ErrorExit("CreateThread Failed!");
AddThreadToList(hThrd);//оп-тель потока в список
dwCount++;// плюс 1 к счетчику потоков
// установка приоритета потока ниже
// приоритета первичного потока,
// отвечающего за пользовательский ввод.
if (!SetThreadPriority(hThrd,
      THREAD_PRIORITY_BELOW_NORMAL))
      ErrorExit("SetThreadPriority failed!");
if ((ResumeThread(hThrd)) == -1)
     ErrorExit("ResumeThread failed!");
return 0L;
- - -


DWORD ThreadFunc(HWND hwnd)
{
LONG lKillMe = 0L;
while (TRUE){
   lKillMe = GetWindowLong(hwnd, GWL_USERDATA);
   if (fKillAll || lKillMe) break;
   // . . .
   // Здесь выполнение задачи потока.
   // . . .
   }
// Действия перед завершением потока
return 0;
```

```
}
```

10 Преобразование псевдоописателя в настоящий описатель

Иногда бывает нужно выяснить “настоящий”, а не псевдоописатель потока. В приводимом фрагменте кода родительский поток передает дочернему свой описатель. Но он передает псевдо –, а не настоящий описатель. Начиная выполнение, дочерний поток передает этот псевдоописатель функции GetThreadTimes, и она вследствие этого возвращает временные показатели своего – а вовсе не родительского потока. Все дело в том, что значение псевдоописателя в любом потоке одно и то же – 0xFFFFFFFE, и всегда указывает на текущий поток:

```
DWORD WINAPI ParentThread(PVOID pvParam)
{
HANDLE hThreadParent = GetCurrentThread();
CreateThread(NULL, 0, ChildThread, (PVOID)
hThreadParent, 0, NULL);
// далее следует какой-то код
}
DWORD WINAPI ChildThread(PVOID pvParam)
{
HANDLE hThreadParent = (HANDLE) pvParam;
FILETIME ftCreationTime, ftExitTime,
ftKernelTime, ftUserTime;
GetThreadTimes(hThreadParent, &ftCreationTime,
&ftExitTime, &ftKernelTime, &ftUserTime);
// далее следует какой-го код.
}
```

Чтобы исправить приведенный выше фрагмент кода, превратим псевдоописатель в настоящий через функцию *DuplicateHandle*:

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

Скорректируем с ее помощью фрагмент кода так:

```
DWORD WINAPI ParentThread(PVOID pvParam)
{
HANDLE hThreadParent;
DuplicateHandle(
// описатель исходного процесса
```

```
GetCurrentProcess(),
// псевдоописатель родительского потока;
GetCurrentThread(),
// описатель целевого процесса
GetCurrentProcess(),
// адрес места для нового настоящего описателя,
// идентифицирующего родительский поток;
&hThreadParent,
0, // игнорируется из-за DUPLICATE_SAME_ACCESS
FALSE, // новый описатель потока ненаследуемый,
DUPLICATE_SAME_ACCESS // новому описателю
// потока присваиваются те же атрибуты защиты,
// что и псевдоописателю
);
CreateThread(NULL, 0, ChildThread, (PVOID)
hThreadParent, 0, NULL) ;
// далее следует какой-то код
}

DWORD WINAPI ChildThread(PVOID pvParam)
{
HANDLE hThreadParent = (HANDLE) pvParam;
FILETIME ftCreationTime, ftExitTime,
ftKernelTime, ftUserTime;
GetThreadTimes(hThreadParent, &ftCreationTime,
&ftExitTime, &ftKernelTime, &ftUserTime);
CloseHandle(hThreadParent);
// далее следует какой-то код..
}
```

Теперь родительский поток преобразует свой псевдоописатель в настоящий описатель, однозначно определяющий родительский поток, и передает его в ***CreateThread***