Abend 5
Erzeugen eines Threads mit der Windows-API

Ziel, Inhalt

- Wir erzeugen heute unter Windows einen zum main parallelen Arbeitsfaden (Thread) und beenden diesen auf sichere Art wieder

Erzeugen eines Thread mit der Windows-API

Was ist ein Thread

Ein Thread wird verwendet um mehrere Dinge in einem Programm parallel ablaufen zu lassen. Normalerweise startet unser Programm in einer *main*-Funktion und von dort aus können wir schrittweise verfolgen was unser Programm macht. Gewisse Aufrufe können aber manchmal sehr lange gehen, oder führen dazu, dass auf eine Eingabe des Benutzers gewartet wird. Als Beispiel möchte ich auch das Lesen von einer Schnittstelle, wie der seriellen Schnittstelle nennen. Solange keine Daten kommen bleibt der Aufruf hängen. Das Windows-Betriebssystem bietet dem Programmierer die Möglichkeit parallele Verarbeitungsstränge zu erzeugen. Das Programm scheint mehrere Dinge gleichzeitig zu tun, ein Umstand dem wir beim verwenden von Threads besondere Aufmerksamkeit widmen müssen.

Die Windows-API

Der Ausdruck API bezeichnet die Programmierschnittstelle, die den Applikationen zur Verfügung steht (**A**pplication **P**rogramming **I**nterface). Da die meisten von uns einen PC mit Windows Betriebssystem verwenden, betrachten wir die Windows-API ein wenig genauer.

Zugriff auf die Windows-Funktionen

Um Funktionen und Datentypen des Betriebssystem zu verwenden genügt es im allgemeinen die Datei „Windows.h“ mittels *#include* einzubeziehen. Danach stehen viele Datentypen und Funktionen zur Verfügung. Für einige Funktionen müssen aber zum Teil gewisse Libraries zusätzlich gelinkt werden. Bis auf eine kleine, „komische“ Ausnahme, ist die Windows-API eine reine C-Library und besitzt keine objektorientierte Schnittstelle.

Die Datentypen aus Windows.h

Da Windows von verschiedenen Programmiersprachen und Compilern aus programmiert werden kann, wurden alle Datentypen speziell definiert. In Basic ist es kaum bekannt, dass es in C++ einen *unsigned long* Datentypen gibt. Damit aber die Funktionen nur einmal definiert werden müssen, werden nur speziell definierte Datentypen verwendet. Diese werden dann für die jeweiligen Sprachen und Compiler definiert. Windows Datentypen sehen sich ähnlich. Sie sind meistens gross geschrieben. Auch werden bei Betriebssystem-Funktionen keine Referenzen verwendet. Hier einige Beispiele für Windows-Datentypen und deren Bedeutung.

Windows Datentyp	C++ Datentyp	Beschreibung
LONG	long	
DWORD	unsigned long	double word. Ein Word sind 16 bit, ein double Word demnach 32 bit
HANDLE	void*	Ein Handle wird verwendet um Betriebssystem-Objekt zu identifizieren
LPCTSTR	const char* oder const wchar_t*	Wird als Zeiger auf einen Nullterminierten C-String verwendet (auch Unicode)
LPVOID	void*	Das LP steht im Allgemeinen für Long-Pointer (32 Bit-Zeiger)

Betriebssystemobjekte

Das Betriebssystem kümmert sich ja um viele Dinge, wie zum Beispiel die Dateibehandlung oder die Bildschirmausgabe. Die Datenstrukturen die intern dafür gebraucht werden können wir auch als Betriebssystemobjekte betrachten. Ein Fenster auf dem Bildschirm ist ein solches Betriebssystem-Objekt. Auch eine Datei ist ein Betriebssystem-Objekt. Diese Betriebssystem-Objekte werden über Handles (Datentyp häufig HANDLE) identifiziert. Für die meisten Betriebssystem-Objekte gibt es „Create“-Funktionen. Für Ein- und Ausgabe-Objekte gibt es die *CreateFile*-Funktion. Diese Funktionen haben als Rückgabewert ein solches HANDLE. Wir lernen heute folgende Betriebssystem-Objekte kennen:

- Thread, Ein Handlungsstrang in einem Programm. Erzeugt mit *CreateThread*

- Event, Ein Objekt das ähnlich wie ein *bool* verwendet werden kann und dazu dient Signale zu geben, auf die gewartet werden kann. Erzeugt mit *CreateEvent*

Solche Handles werden nachdem sie nicht mehr gebraucht müssen mit der API-Funktion CloseHandle wieder zerstört und freigegeben werden.

Einen Thread erzeugen

Die Thread-Funktion

Um einen Thread zu erzeugen müssen wir dem Betriebssystem eigentlich nur mitteilen wo sich die Eintrittsfunktion befindet, die auf dem neu erzeugten Handlungsstrang, also Thread aufgerufen wird. Stelle dir diese Funktion wie eine Art zweite *main*-Funktion vor. Damit das Betriebssystem eine Funktion als weitere *main*-Funktion aufrufen kann muss diese Funktion folgende Form haben:

```
DWORD WINAPI ThreadFunktion(LPVOID param);
```

Dabei ist DWORD der Rückgabewert. WINAPI bezeichnet die Aufrufkonvention, das heisst die Art wie der Compiler Argumente und Rückgabewerte behandelt. Für uns genügt es zu wissen, dass wir einfach WINAPI hinschreiben können und dass die Funktion dadurch nicht eine Methode von einer Klasse sein kann, ausser es ist eine statische Methode! Die Funktion kann einen beliebigen Namen haben. Es ist auf jeden Fall der Name, den wir der CreateThread-API übergeben müssen. Das Argument ist ein Zeiger auf „irgendetwas“. Wir können über dieses Argument der Thread-Funktion ein Argument übergeben.

Die CreateThread-API

Die CreateThread-API hat wie viele Funktionen einen Haufen Parameter, wobei wir viele davon nicht brauchen, oder mit default (Standard)-Werten belegen können.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
  SIZE_T dwStackSize,                       // stack size
  LPTHREAD_START_ROUTINE lpStartAddress,    // thread function
  LPVOID lpParameter,                       // thread argument
  DWORD dwCreationFlags,                    // creation option
  LPDWORD lpThreadId                       // thread identifier
);
```

Das Windows-Betriebssystem kennt Benutzer-Rechte und dergleichen. Das erste Argument würde ermöglichen einen Thread als ein bestimmter Benutzer oder mit bestimmten Benutzerrechten auszuführen. Für unsere Zwecke ignorieren wir das einfach und geben eine 0 als Argument an. **Jeder Thread hat einen eigenen Stack, auf dem lokale Variablen erzeugt werden!** Wir werden sehen, dass wir Funktionen von verschiedenen Threads aus aufrufen können. Wenn wir in einer solchen Funktion eine Variable erzeugen und diese Funktion gleichzeitig auf einem anderen Thread aufrufen existiert die Variable zweimal unabhängig voneinander. Was das bedeutet oder nützt sehen wir später. Auf jeden Fall können wir mit dem zweiten Argument, die grösse des Stack für den Thread bestimmen. Wenn

wir hier eine 0 einsetzen wird eine default-Grösse gewählt.
Der dritte Parameter ist nun die Thread-Funktion, die ausgeführt werden soll.
Der lpParameter ist wie ein *void**. Dieser Zeiger wird vom Betriebssystem der Funktion als Argument übergeben. Meistens ist das sehr nützlich!
Übder den zweitletzten Parameter, den Creation-Flag Parameter ist es möglich zu steuern wie der Thread erzeugt wird. Es sit möglich den Thread zu erzeugen aber nicht sofort zu starten.
Der letzte Parameter ist ein Zeiger auf eine Variable vom Datentypen DWORD. Das Betriebssystem setzt diesen Wert auf die ThreadId. Die ThreadId ist ähnlich wie das HANDLE des Threads zur Identifikation des Threads manchmal nötig. Hier also ein Beispiel:

```
#include <windows.h>

DWORD WINAPI ThreadFunktion(LPVOID param)
{
    // tue nichts
    return 0;
}

int main()
{
    DWORD threadId = 0;

    HANDLE threadHandle = CreateThread(0, // keine Security
                                       0, // default-Stack
                                       ThreadFunktion,
                                       0, // kein Parameter
                                       0, // normal erzeugen
                                       &threadId // threadId
                                       )
    return 0;
}
```

Dieses Beispiel wird nichts sinnvolles machen, es zeigt einfach ein Beispiel für die CreateThread-API Funktion.

Synchronisierung

Die Synchronisierung zwischen Threads ist dann nötig, wenn ein Thread etwas von einem anderen Thread braucht, oder sonst irgendwie mit einem Thread Kontakt hat. Die erste Art der Synchronisierung ist das Warten des Hauptthread auf die Beendigung des zweiten. Wir wollen sicherstellen, dass vor dem Beenden der main-Funktion alle ezeugten Threads auch beendet wurden. Andernfalls werden diese einfach beendet, ohne dass sie möglicherweise nötige Aufräumarbeiten erledigen.

Warten auf Signale

Es gibt die Möglichkeit mit einer Betriebssystem-Funktion auf gewisse Dinge zu warten, ohne dass dabei Rechenzeit in irgendwelchen Warteschleifen vergeudet wird. Warten kann man auf bestimmte HANDLE's die signalisiert sein können. Dazu gehören auch HANDLE's von Threads. Diese HANDLE's werden signalisiert, wenn der zugehörige Thread beendet wurde! Eine API-Funktion zum Warten auf Signale heisst WaitForSingleObject:

```
DWORD WaitForSingleObject(HANDLE handle, DWORD timeout);
```

Das erste Argument ist das HANDLE auf das gewartet werden soll, das zweite Argument gibt an wieviele Millisekunden gewartet werden soll. Es gibt aber eine spezielle Konstante INFINITE mit der man ewig warten kann. Der Rückgabewert gibt an wieso die Wartefunktion beendet wurde. Einerseits könnte das HANDLE signalisiert sein, andererseits könnte das time-out abgelaufen sein. Es gibt dafür einige vordefinierte Konstanten.

Warten auf Thread-Beendigung

```
#include <windows.h>

DWORD WINAPI ThreadFunktion(LPVOID param)
{
    // tue nichts
    return 0;
}

int main()
{
    DWORD threadId = 0;

    HANDLE threadHandle = CreateThread(0, // keine Security
                                       0, // default-Stack
                                       ThreadFunktion,
                                       0, // kein Parameter
                                       0, // normal erzeugen
                                       &threadId // threadId
                                       )
    DWORD wait = WaitForSingleObject(threadHandle, 10000);
    if(WAIT_TIMEOUT == wait)
    {
        // Das Time-out ist eingetreten
    }
    else if(WAIT_OBJECT_0 == wait)
    {
        // Das Handle wurde signalisiert
        // Der Thread wurde also beendet
    }

    return 0;
}
```

Bei diesem Beispiel warten wir 10000 ms, also 10 Sekunden. Falls dieses Timeout eintritt ist die erste Bedingung erfüllt, falls aber der Thread beendet wird ist der Rückgabewert der Funktion WAIT_OBJECT_0 und die zweite Bedingung ist erfüllt.

Signale mittels Events

Um einem Thread mitzuteilen, dass er etwas tun muss oder beenden soll, könnte man eine globale Variable definieren, die vom Hauptthread gesetzt wird und vom zweiten Thread ausgelesen wird. Der zweite Thread müsste dann "ab und zu" nachsehen, ob die Variable verändert wurde. ABER VORSICHT! Diese Art eine Variable von verschiedenen Threads zu lesen und zu schreiben ist unsicher, denn man kann nicht sichergehen ob die Variable genau dann gelesen wird, während sie vom anderen Thread geschrieben wird. Globale Variablen sind im allgemeinen nicht "Thread-Safe", nicht sicher für Threads.
Es gibt eine viel sauberere Lösung unter Windows, die auch weniger Rechenzeit verbraucht als ständig eine Variable zu prüfen. Man spricht von sogenannten Events. Solche Events sind ganz einfache Signale, die entweder gesetzt oder nicht gesetzt sind. Das wichtigste dabei ist, dass man mit WaitForSingleObject darauf warten kann, dass ein solches Event gesetzt wird. Hier die nötigen Funktionen:

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES Security,
                   BOOL ManualReset,
                   BOOL InitialState,
                   LPCTSTR Name);

BOOL SetEvent(HANDLE Event);

BOOL ResetEvent(HANDLE Event);
```

Mit CreateEvent wird ein solches Event-Objekt erzeugt. Das erste Argument haben mit Security zu tun. Das zweite Argument bestimmt ob ein Event mit ResetEvent zurückgesetzt werden muss. Wird hier FALSE verwendet, wird das Event automatisch in den nicht gesetzten Zustand zurückgesetzt, sobald ein einzelner Thread mit WaitForSingleObject (oder einer ähnlichen Funktion) erfolgreich auf das Event gewartet hat. Ein zweiter Thread, der auch auf das Event wartet, wird nicht geweckt. Das dritte Argument gibt an, ob das Event mit gesetztem Zustand erzeugt wird. Mit dem vierten Argument sind wir in der Lage dem Event einen Namen zu geben. Wir sehen im folgenden Beispiel, wie man ein solches Event einsetzen kann.

Beispiel mit Ausgabe auf zwei verschiedenen Threads

```
#include <windows.h>
#include <string>
#include <iostream>

using namespace std;

// struktur mit Daten, die dem Thread übergeben werden
struct ThreadData
{
    HANDLE stopEvent;
    string     data;
    ThreadData() : stopEvent(0)
    {
    }
};

DWORD WINAPI ThreadFunc(LPVOID param)
{
    ThreadData* threadData = (ThreadData*)param;

    while(1)
    {
        DWORD wait = 0;
        // wir warten hier darauf, dass das Event gesetzt wird
        // Falls das innerhalb von 500ms nicht geschieht, haben
        // wir ein Timeout, sonst wird der Thread beendet
        wait = WaitForSingleObject(threadData->stopEvent, 500);
        if(WAIT_TIMEOUT == wait)
        {
            cout << threadData->data << endl;
        }
        else
        {
            break; // Schleife verlassen und somit den
        }          // Thread beenden
    }

    return 69;
}

// main auf der nächste Seite
```

```
int main()
{
    ThreadData* threadData = new ThreadData;
    threadData->data = "Hallo Thread";
    // Hier wird das Event erzeugt
    threadData->stopEvent = CreateEvent(0,
                                        FALSE,
                                        FALSE,
                                        "StopEvent");


    DWORD threadId = 0;
    // und hier ezeugen wir den Thread
    HANDLE h = CreateThread(0, // keine Security
                            0, // normale Stackgrösse
                            ThreadFunc,
                            (void*)threadData,
                            0,
                            &threadId);

    for(int i = 0; i < 8; ++i)
    {
        cout << "Main Thread" << endl;
        Sleep(1000);
    }

    // Zeit messen um zu sehen wie lange es
    // dauert bis der Thread beendet wird
    DWORD startWait = GetTickCount();
    // Signal setzen. damit der Thread merkt,
    // dass er beenden soll
    SetEvent(threadData->stopEvent);

    DWORD wait = WaitForSingleObject(h, INFINITE);
    DWORD diff = GetTickCount() - startWait;

    cout << "Wartezeit : " << diff << " ms" << endl;

    // da der Thread nun sicher beendet ist, können wir
    // die Daten sicher löschen
    delete threadData;

    return 0;
}
```

Wir haben hier aber eines der häufigsten Probleme beim Programmieren mit Threads. Es kann geschehen, dass beide Threads gleichzeitig etwas auf *cout* ausgeben wollen. Dadurch kann der Text unterbrochen werden und komisch aussehen. Wir lernen noch einen Weg kennen, wie man Threads voneinander schützt und den gleichzeitigen Zugriff auf Resourcen verhindert.
Du findest den Code zum Download auf
http://www.devmentor.ch/teaching/additional/00I/Semester5/Abend5/main.cpp.