



## Nebenläufige Prozesse

Innerhalb eines in Ausführung befindlichen Programmes -das ist ein Prozess!- können weitere Programmabläufe gestartet werden. Solche sogenannten nebenläufigen Prozesse nennt man Threads!

### Prozesse und Threads unterscheiden sich!<sup>1</sup>

Verschiedene Prozesse haben voneinander getrennte virtuelle Adressräume. Damit ist gemeint, dass jeder Prozess "denkt", er hätte den kompletten Arbeitsspeicher eines Rechnersystems zur Verfügung stehen. Prozesse wissen normalerweise nichts voneinander! Auf einem Rechnersystem mit einem Prozessor, werden verschiedene Prozesse zeitlich hintereinander ausgeführt. Die Verwaltung hierzu übernimmt der sogenannte **Scheduler**.

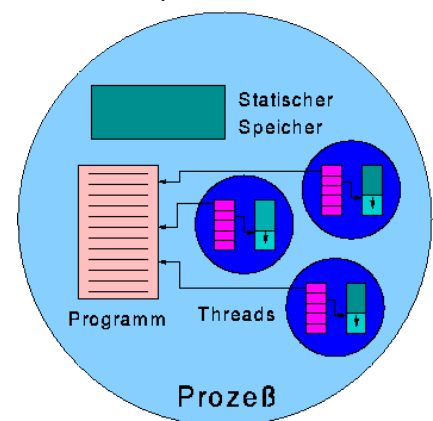
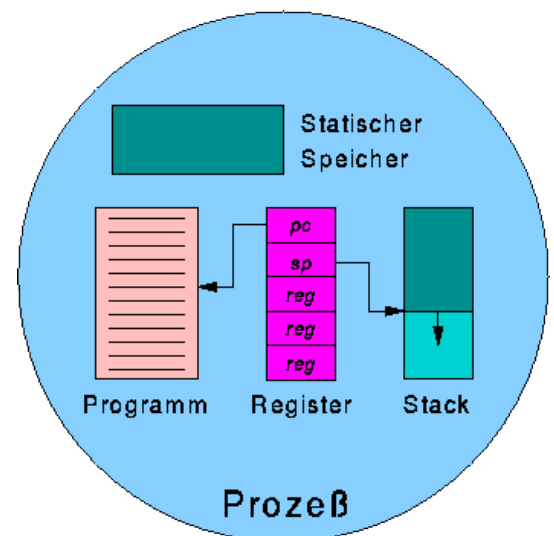
Das ist der Teil des Betriebssystems, der darüber wacht, dass jedes in Ausführung befindliche Programm für eine bestimmte Zeit den Prozessor und die weiter notwendigen Ressourcen zugeteilt erhält. Durch die Geschwindigkeit, mit der dies erledigt wird, entsteht der Eindruck, dass die Programme quasi zeitlich parallel ausgeführt werden. Erst hierdurch wird ein Betriebssystem multitaskingfähig.

Für die Ausführung von Prozessen, muss das Betriebssystem eine Menge an Verwaltungsarbeit übernehmen. Jedes Mal, wenn auf einen anderen Prozess umgeschaltet wird, müssen alle Informationen für diesen Prozess bereitgestellt werden. Jeder Prozess bekommt somit das "Gefühl" allein auf einer Maschine ausgeführt zu werden. Er "merkt" nicht, dass er durch den Scheduler zeitweilig in eine "Warteschleife" geschickt wird. Wie die Grafik oben<sup>2</sup> andeutet, wird für jeden Prozess das zugehörige ausführbare Programm im Arbeitsspeicher gehalten. Weiterhin gehören zu einem Prozess statischer Speicher, der alle globalen Variablen und nach dieser Skizze den Heap, den Speicher für dynamisch erzeugte Objekte beinhaltet und der Stapelspeicher, der alle lokalen Variablen aus den verschiedenen Funktionen aufnimmt. Zuletzt gehören noch die Registerinhalte der CPU zu einem Prozess.

Siehe auch: [http://de.wikipedia.org/wiki/Prozess\\_\(Informatik\)](http://de.wikipedia.org/wiki/Prozess_(Informatik))

Threads sind ebenfalls Programmabläufe, die in Ausführung gebracht werden, die aber mit einem Prozess in Verbindung stehen. Ein Beispiel sei das Starten eines Druckvorganges in einer Textverarbeitung. Die Daten werden aufbereitet und an den Drucker gesendet, währenddessen eine Weiterarbeit in der Textverarbeitung möglich ist. Oder: Ein Server nimmt Clientverbindungen an und die Kommunikation mit dem Client wird in einen Thread ausgelagert, der es dem Server ermöglicht, auf Verbindungswünsche weiterer Clients zu reagieren.

Jeder Thread hat seinen eigenen Stapelspeicher für die Ablage von lokalen Variablen und seinen eigenen Registersatz. Ein wich-



1 Eine englischsprachige Beschreibung: <https://computing.llnwd.net/tutorials/pthreads/>

2 <http://www.risc.uni-linz.ac.at/people/schreine/papers/rt++-linuxmag1/main.html>



tiger Unterschied ist jedoch, dass Threads sich mit dem Prozess, durch den sie gestartet wurden, dessen statische und globale Variablen teilen, wie in der vorigen Skizze<sup>3</sup> angedeutet ist.

Ergänzende einführende Informationen unter:

[http://de.wikipedia.org/wiki/Thread\\_\(Informatik\)](http://de.wikipedia.org/wiki/Thread_(Informatik))

Für die Erzeugung von Threads können bei Microsoft Windows Betriebssystemen verschiedene Methoden verwendet werden. Eine Variante der Threaderzeugung sind die sogenannten CRT<sup>4</sup>-Threads.

## CRT-Threads

In der C-Runtime Library sind mittels der Headerdatei `process.h` Threadfunktionen verfügbar. Ebenfalls wichtig hierbei ist, dass der Linker die geeigneten Bibliotheken verwendet. Zumindest unter Visual Studio 2005 sind diese mittlerweile alle als Multithreadingfähig. Bei älteren Visual Studio Versionen musste dem Linker in den Linkeroptionen der Schalter `/MT` bzw. `/MTd` für die Debug-Version eines Programms angegeben werden, damit die erforderliche Bibliothek verwendet wurde.

Da die Materie sehr komplex ist, soll anhand eines stark vereinfachten Demoprogramms das Grundprinzip dargestellt werden<sup>5</sup>. Weitergehende Thematiken wie Suspendieren eines Threads oder die Kommunikation zwischen Threads sollen hier ausgeblendet bleiben.

In der `main()`-Funktion werden zwei Threads jeweils einmal mit der Anweisung `_beginthread()` gestartet.

Diese Funktion hat drei Parameter:

- Der Name der zu startenden Funktion
- Die Größe des Stapelspeichers; der Wert 0 gibt den Standard vor.
- eventuelle Parameter für die Threadfunktion; ist keiner erforderlich, wird `NULL` angegeben

Die erste Funktion erhält keine Parameter; die zweite erhält den `int`-Wert 4711 als Parameter. Aufgrund der erforderlichen Datentypen ist eine Typumwandlung (`cast`) auf einen `void`-Zeiger erforderlich.

Beide Threads enthalten eine Endlosschleife. Erst mit dem Ende der `main()`-Funktion, also nach Drücken einer Taste, enden beide Threads.

```
#include <iostream>
#include <conio.h>
#include <process.h>

using namespace std;

void Thread1( void* noPar )
{
    while ( 1 )
    {
        cout << _threadid << ":In Thread " << endl;
    }
}

void Thread2( void* threadnr )
{
    while ( 1 )
    {
        cout << _threadid << ":Anderer Thread "
            << (int)threadnr << endl;
    }
}

int main( void )
{
    int threadnr = 4711;

    _beginthread( Thread1, 0, NULL );
    _beginthread( Thread2, 0, (void *)threadnr );

    while( !_kbhit() )
        cout << "In der Main-Funktion..." << endl;

    printf("Fertich...");

    return 0;
}
```

<sup>3</sup> <http://www.risc.uni-linz.ac.at/people/schreine/papers/rt++-linuxmag1/main.html>

<sup>4</sup> C-Runtime Bibliothek

<sup>5</sup> Eine kurze aber prägnante Einführung (in Englisch) bei <http://www.adrianxw.dk/SoftwareSite/Threads/Threads1.html>



## Absprachen sind notwendig!

Das zuvor vorgestellte Rumpfprogramm führt die beiden Threads aus: Neben ist ein typischer Screenshot dargestellt.

Es fällt sofort ins Auge, dass die Ausführung des eines Threads zeitweise durch die Ausführung des anderen Threads unterbrochen wird, wodurch das „Durcheinander“ in der Ausgabe entsteht.

Der Scheduler des Betriebssystems teilt ja jedem Thread eine bestimmte Zeitspanne für die Ausführung zu, dann wird er unterbrochen und der nächste Thread kommt an die Reihe.

Also muss ein Möglichkeit gesucht werden, wie solch ein Durcheinander verhindert werden kann.

Eine Website<sup>6</sup> bemüht hierzu ein Beispiel: Eine Gruppe Schuljungen strandet auf einer Insel. Bei Besprechungen reden alle wild durcheinander. Einer hat nun die Idee, dass immer nur derjenige sprechen darf, der eine bestimmte Seemuschel in der Hand hält. Sobald er mit Reden fertig ist, reicht er die Seemuschel an den nächsten Redewilligen weiter.

Und solch ein Verfahren eignet sich auch, um das Durcheinander bei der Threadausführung zu verhindern!

Hierzu verwendet man ein so genanntes **CRITICAL\_SECTION**-Objekt. Durch die Einbindung von `windows.h` wird der Datentyp verfügbar. Die verwendete Objektvariable soll `Section` heißen. Die sich dahinter befindliche Struktur muss vor der Benutzung initialisiert werden. Dies erfolgt durch folgenden Aufruf: `InitializeCriticalSection( &Section );`

Die Funktion erhält hierzu die Adresse der zu verwendenden Objektvariablen. Wird das Objekt nicht mehr benötigt, bzw. am Ende der `main()`-Funktion wird die Objektvariable zerstört. Hierzu verwendet man die Funktion `DeleteCriticalSection( &Section );`

Die Objekt-Variable muss nun für alle Threads erreichbar sein. Deshalb ist die Variable global zu deklarieren.

Durch zwei Funktionen wird nun die „Seemuschel“ ergriffen, „geredet“ und freigegeben:

`EnterCriticalSection( &Section )` und `LeaveCriticalSection( &Section )`.

An allen Stellen, an denen nun eine Koordination erforderlich ist, wird der „kritische“ Bereich betreten und anschließend wieder verlassen.

```
C:\WINDOWS\system32\cmd.exe
21482392:In Thread
:Anderer Thread 4711
2148:Anderer Thread 4711
2392:In Thread
2148:Anderer Thread 4711
2148:Anderer Thread 4711
2392:In Thread
2148:Anderer Thread 4711
2148:Anderer Thread 4711
2392:In Thread
2148:Anderer Thread 4711
2148:Anderer Thread 2392:In Thread 4711

2148:Anderer Thread 4711
2392:In Thread
2148:Anderer Thread 4711
2148:Anderer Thread 4711
2392:In Thread
2148:Anderer Thread 4711
2148:Anderer Thread 4711
2392:In Thread
2148:Anderer Thread 4711
```

```
...
#include <process.h>
#include <windows.h>
using namespace std;

CRITICAL_SECTION Section;

void Thread1( void* noPar )
{
    while ( 1 )
    {
        EnterCriticalSection(&Section);
        cout << _threadid << ":In Thread " << endl;
        LeaveCriticalSection(&Section);
        Sleep(100);
    }
}
...
int main( void )
{
    InitializeCriticalSection(&Section);
    int threadnr = 4711;

    _beginthread( Thread1, 0, NULL );
    _beginthread( Thread2, 0, (void *)threadnr );
    ...
    DeleteCriticalSection(&Section);
    return 0;
}
```



## Warten auf Godot

Threads werden ja häufig eingesetzt, um im "Hintergrund" eine zeitaufwändige<sup>7</sup> oder eigenständig ablaufende<sup>8</sup> Aufgabe auszuführen. Ansonsten wäre das Benutzerinterface für diese Zeit nicht zugänglich.

### Im Threading-Modell von Windows endet ein noch in Ausführung befindlicher Thread aber mit dem Prozess, der ihn ins Leben gerufen hat!

Es gibt nun aber eine Möglichkeit, das Terminieren des Prozesses zu verzögern, bis auch der letzte gewünschte Thread seine Aufgabe erledigt hat:

```
WaitForMultipleObjects (2,          //Die Anzahl zu überwachender Threads
                        hThreads,   //Ein Array mit den Threadnummern
                        TRUE,        //Es wird auf alle Threads gewartet
                        INFINITE)    //Es wird unendlich lange gewartet
```

Wenn ein Thread mit `_beginthread()` gestartet wird, liefert diese Funktion eine Nummer, die das Betriebssystem vergibt; ein sogenanntes **HANDLE**. Über ein Handle ist jeder Thread zugreifbar. Die Funktion kann nun in dem Array mit den **HANDLE**-Objekten prüfen, ob die zugehörigen Threads ihre Aufgabe beendet haben.

Zu Beginn wird das Array für die Handles angelegt (hier: `hThreads[2]`).

Den Arrayelementen wird bei Start eines Threads mit `_beginthread()` dessen Rückgabewert zugewiesen. Der Datentyp wird mit einem type-cast auf **HANDLE** angepasst.

`WaitForMultipleObjects()` wartet nun auf alle im Array eingetragenen Threads, bis diese ihre jeweilige Aufgabe erledigt haben und terminieren.

```
...
#include <process.h>
#include <windows.h>
using namespace std;

CRITICAL_SECTION Section;

void Thread1( void* noPar )
{
    while ( 1 )
    {
        EnterCriticalSection(&Section);
        cout << _threadid << ":In Thread " << endl;
        LeaveCriticalSection(&Section);
        Sleep(100);
    }
}
...
int main( void )
{
    HANDLE hThreads[2]; //Zwei Threads
    InitializeCriticalSection(&Section);
    int threadnr = 4711;

    hThreads[0] = (HANDLE)_beginthread(
        Thread1,
        0,
        NULL
    );

    hThreads[1] = (HANDLE)_beginthread(
        Thread2,
        0,
        (void *)threadnr
    );
    ...
    WaitForMultipleObjects(2, hThreads, TRUE, INFINITE);

    DeleteCriticalSection(&Section);
    return 0;
}
```

<sup>7</sup> z.B. das Drucken eines Bildes

<sup>8</sup> wie z.B. sich sich bewegende Elemente in einem Spielprogramm  
© Uwe Homm Version vom 1. Mai 2009



## Parameter für den Thread

In den vorigen Beispielen wurde an den einen Thread bereits ein `int`-Wert als Parameter übergeben. Der Datentyp für den Parameter der Funktion `_beginthread()` lautet `void*`; also ein untypisierter Zeiger. Der zu übergebende Wert muss in diesen Typ „gecastet“ werden (können).

Das funktioniert aber nicht mit `float`- oder `double`-Werten oder gar mit Strukturen oder Klassen. Hier gibt es einen Compiler-Fehler, dass die Typumwandlung nicht möglich sei.

Abhilfe schafft hier nur die Übergabe eines Zeigers auf den gewünschten Typ. Es muss also ein **Zeiger auf eine Objekt-Variable** vom gewünschten Typ (hier: `Caesar`) deklariert und mit `new` initialisiert werden.

In der Thread-Funktion wird nun ebenfalls eine Hilfsvariable von diesem Typ erzeugt und der übergebene Wert „gecastet“.

Innerhalb der Thread-Funktion kann man nun per Pfeil-Operator (`->`) auf Methoden und Attribute einer Klasse oder auf Strukturelemente zugreifen.

```
...
#include <process.h>
#include <windows.h>
#include "Caesar.h"
using namespace std;

CRITICAL_SECTION Section;

void Thread2( void* nr )
{
    Caesar n = (Caesar*)nr;
    while ( 1 )
    {
        EnterCriticalSection(&Section);
        cout << "In Thread (" << i << " )"
             << _threadid
             << " Parameter: "
             << n->show() << " " << n->value << endl;
        LeaveCriticalSection(&Section);
        Sleep(100);
    }
}
...
int main( void )
{
    InitializeCriticalSection(&Section);
    Caesar* c = new Caesar(10);

    _beginthread( Thread1, 0, NULL );
    _beginthread( Thread2, 0, (void *)c );
    ...
    DeleteCriticalSection(&Section);
    return 0;
}
```