

## Kurseinheit 12: Threads

1. Prinzip Threads
2. Windows Thread API
3. Typische Fallstricke mit Threads

## Übersicht KE 12

Lehrveranstaltung Ingenieur-Informatik – 2 SWS/2 Credits: EI1, EI+1, MKA1, MK+1, EI3nat3  
Lehrveranstaltung Programmierung 2 (Teil C) – 2 SWS/2 Credits: AI2

Unterrichtsdauer für diese Kurseinheit: 90 Minuten

Korrespondierende Kapitel aus *C-Programmierung – Eine Einführung*: **Dort nicht vorhanden**

Zusatzthemen: Keine

## 1. Prinzip Threads

3

### Definition

In der Informatik bezeichnet **Thread** [θɹɛd] (englisch thread, ‚Faden‘, ‚Strang‘) – auch **Aktivitätsträger** oder **leichtgewichtiger Prozess** genannt – einen Ausführungsstrang oder eine Ausführungsreihenfolge in der Abarbeitung eines Programms. Ein Thread ist Teil eines Prozesses.

Es wird zwischen zwei Arten von Threads unterschieden:

1. Threads im engeren Sinne, die sogenannten **Kernel-Threads**, laufen ab unter Steuerung durch das Betriebssystem.
2. Im Gegensatz dazu stehen die sogenannten User-Threads, die das Computerprogramm des Anwenders komplett selbst verwalten muss.

Quelle: wikipedia.de

Ein Programm kann mehrere solcher Ausführungsstränge (Einstieg ist meist eine Funktion, die sogenannte Threadfunktion) haben.

Bisher wurde dies noch nie berücksichtigt! Dann besteht das Programm sozusagen aus nur einem Thread, dem Hauptthread.

## 1. Prinzip Threads

4

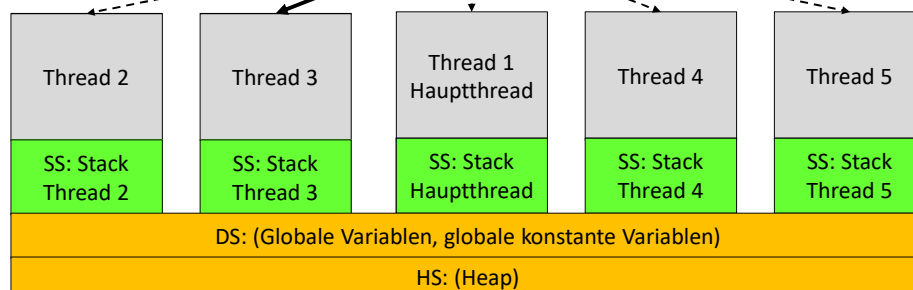
### Scheduler

Scheduler (dt. Disponent, Einplaner) ist Teil des Betriebssystems.

Scheduler schaltet in kurzen Zyklen zwischen Threads um (eine CPU).

Scheduler

„Quasi-Parallelität“ bei einer CPU  
Parallelität beim mehreren CPUs



Threads teilen sich das Datensegment und das Heapsegment! Jeder Thread hat einen eigenen Stack!

## 1. Prinzip Threads

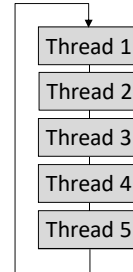
5

### Nutzen von Threads

Eine Anwendung soll exemplarisch die folgenden Aufgaben erledigen:

- User-Interface aktualisieren (Thread 1)
- Drucken (Thread 2)
- Bilddateien (Bilder) einlesen (Thread 3)
- Bilder bearbeiten (Glätten) (Thread 4)
- Bilder in Datenbank abspeichern (Thread 5)

Eine CPU:

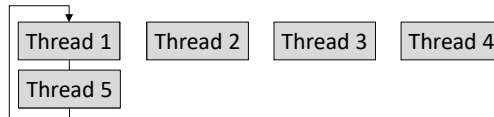


Bei nur einer CPU läuft das Programm sogar noch etwas langsamer, da intern noch auf den neuen Thread umgeschaltet wird. Dies wird „Kontextwechsel“ genannt.

Bei mehreren CPUs läuft das Programm dann schneller.

Programme, die schon mehrere Threads anlegen, laufen dann auf einem Rechner mit mehreren CPUs schneller.

Vier CPUs:



## 1. Prinzip Threads

6

### Beispiel Task-Manager (Windows 10)

Name	PID	Threads	Betriebssyst...	Beschreibi
cmd.exe	8080	1		Windows-
dllhost.exe	9272	2		COM Surr
explorer.exe	8040	50		Windows-
firefox.exe	11048	30		Firefox

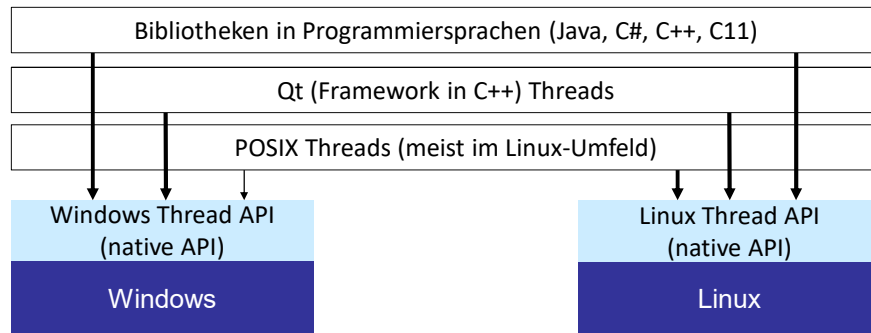
Im Task-Manager ist ersichtlich, dass die meisten Anwendungen aus mehreren Threads bestehen. Die bisher erstellten Programme mussten in der Liste während der Ausführung mit Threads = 1 dargestellt werden. Die Spalte Threads mittels rechter Maustaste im Kopf der Liste auszuwählen.

## 1. Prinzip Threads

7

### Programmierung von Threads

Threads sind Bestandteil des Betriebssystems und über ein Application Programming Interface (API) kann darauf zugegriffen werden.



Meist wird nicht die native API der Betriebssysteme verwendet, sondern eine andere Bibliothek, welche intern dann auf die native API zugreifen. In dieser LV soll die native Windows Thread API eingesetzt werden.

## 2. Windows Thread API

8

### Starten eines Threads

Um die Windows Thread API nutzen zu können, muss nur `<windows.h>` inkludiert werden. Darin sind auch verschiedene Typedefs enthalten. Unter <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread> ist die detaillierte Beschreibung von `CreateThread` zu finden.

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES    lpThreadAttributes,  
    SIZE_T                   dwStackSize,  
    LPTHREAD_START_ROUTINE   lpStartAddress,  
    __drv_aliasesMem LPVOID  lpParameter,  
    DWORD                     dwCreationFlags,  
    LPDWORD                  lpThreadId  
);
```

Besonders wichtig sind dabei:

**lpStartAddress:** Funktionszeiger auf eine Thread-Funktion. Deklaration einer Thread-Funktion:

```
DWORD WINAPI ThreadFunction(void *pParam);
```

**lpParameter:** void-Zeiger auf die Parameter

## 2. Windows Thread API

9

### Starten eines Threads – Handle als Rückgabe

Die Funktion `CreateThread` liefert ein Handle zurück. Dies ist nichts anderes als eine eindeutige Nummer für die interne Verwaltung im Betriebssystem (gekapselter Zeiger). Dieser Handle kann dann für weitere Funktionen verwendet werden.

```
int main(void)
{
    HANDLE hThread1 = 0;

    hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    if (hThread1 != 0)
    {
        // Code
        CloseHandle(hThread);
    }

    return 0;
}
```

Hier werden keine Parameter an die Threadfunktion übergeben.

## 2. Windows Thread API

10

### Zwei Möglichkeiten der Implementierung einer Thread-Funktion

1. Thread-Funktion **terminiert** nach Abarbeitung des Codes

```
DWORD WINAPI ThreadFunction1(void* pParam)
{
    DWORD dwRet = 0;

    // Code

    return 0;
}
```

2. Thread-Funktion **terminiert nicht**

```
DWORD WINAPI ThreadFunction2(void* pParam)
{
    DWORD dwRet = 0;

    while(1)
    {
        // Code
    }
    return 0; // Never reached
}
```

## 2. Windows Thread API

11

### Parameterübergabe an Thread-Funktion

```
int main(void)
{
    struct Data myData = {4711, "Hello"};

    CreateThread(NULL, 0, ThreadFunction1, &myData, 0, NULL);
    return 0;
}
```

Impliziter Cast zu void\*  
(Typenloser Zeiger)

```
DWORD WINAPI ThreadFunction1(void *pParam)
{
    DWORD dwRet = 0;
    struct Data* pData = (struct Data*) pParam;
    // Code
    if (pData->iVal == 4711)
        return 0;
}
```

Auf typenlosen Zeiger  
kann nicht zugegriffen  
werden -> „Rückcast“

```
struct Data
{
    int iVal;
    char acArray[10];
}
```

## 2. Windows Thread API

12

### Synchronisation von Threads

Oft muss im Hauptprogramm oder in einer Thread-Funktion gewartet werden, bis ein oder mehrere Threads terminieren, da deren Ergebnisse für den folgenden Programmcode vorhanden sein müssen.

#### Warten auf einen Thread:

```
DWORD WaitForSingleObject(HANDLE hHandle,
                          DWORD dwMilliseconds );
```

#### Warten auf mehrere Threads:

```
DWORD WaitForMultipleObjects(DWORD nCount,
                             const HANDLE *lpHandles,
                             BOOL bWaitAll,
                             DWORD dwMilliseconds );
```

Details siehe:

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitformultipleobjects>

## 2. Windows Thread API

13

### Beispiel: Synchronisation von Threads

```
DWORD WINAPI ThreadFunction(void* pParam)
{
    struct Data* pData = (struct Data*) pParam;
    double dX = 0.;
    int iJ;
    int iI;

    for (iJ = 0; iJ < 100; iJ++)
    {
        for (iI = pData->iStart; iI < pData->iEnd; iI++)
        {
            dX = sqrt((double)iI);
        }
    }
    return 0;
}
```

„Rückcast“

Ggf. verkleinern

Laufzeit Thread - Workload

```
struct Data
{
    int iStart;
    int iEnd;
};
```

Thread-Funktion terminiert nach Abarbeitung (Workload)

## 2. Windows Thread API

14

### Beispiel: Synchronisation von Threads

Hinweis: In den folgenden Beispielen wird häufig nur eine Threadfunktion für mehrere Threads verwendet. Es kann natürlich auch immer eine andere Funktion verwendet werden.

```
DWORD WINAPI ThreadFunction(void* pParam)
{
    struct Data* pData = (struct Data*) pParam;
    double dX = 0.;
    int iJ;
    int iI;

    for (iJ = 0; iJ < 100; iJ++)
    {
        for (iI = pData->iStart; iI < pData->iEnd; iI++)
        {
            dX = sqrt((double)iI);
        }
    }
    return 0;
}
```

## 2. Windows Thread API

15

### Beispiel: Synchronisation von Threads – Hauptthread und Thread (hThread1)

```
#define COUNTCALC 10000000  
struct Data sData1 = {0, COUNTCALC};
```

```
iClockStart = clock();  
hThread1 = CreateThread(NULL, 0, ThreadFunction, &sData1, 0, NULL);
```

```
if (hThread1 != 0)  
{  
    WaitForSingleObject(hThread1, INFINITE);  
    iClockEnd = clock();  
    dSeconds = ((double)iClockEnd - iClockStart) / CLOCKS_PER_SEC;  
    printf("\n\nLaufzeit ein Thread: %f\n\n", dSeconds);  
    CloseHandle(hThread1);  
}
```

Synchronisation: Hauptthread wartet auf hThread1

Synchronisation: Hauptthread wartet auf Thread (hThread1). Zeitmessung misst die Zeit: Thread starten bis Threadfunktion beendet ist (Thread terminiert).

## 2. Windows Thread API

16

### Beispiel: Synchronisation von Threads – Hauptthread und zwei Threads

```
sData1.iStart = 0;  
sData1.iEnd = COUNTCALC / 2;  
sData2.iStart = (COUNTCALC / 2) + 1;  
sData2.iEnd = COUNTCALC;
```

```
#define COUNTCALC 10000000
```

Über sData1 und sData2 teilen sich die beiden Threads den Workload.

```
iClockStart = clock();  
hThread1 = CreateThread(NULL, 0, ThreadFunction, &sData1, 0, NULL);  
hThread2 = CreateThread(NULL, 0, ThreadFunction, &sData2, 0, NULL);
```

```
if ((hThread1 != 0) && (hThread2 != 0))  
{  
    ahThread[0] = hThread1;  
    ahThread[1] = hThread2;  
    WaitForMultipleObjects(2, (const HANDLE*)&ahThread, TRUE, INFINITE);  
    iClockEnd = clock();  
    dSeconds = ((double)iClockEnd - iClockStart) / CLOCKS_PER_SEC;  
    printf("\n\nLaufzeit zwei Threads: %f\n\n", dSeconds);  
}
```

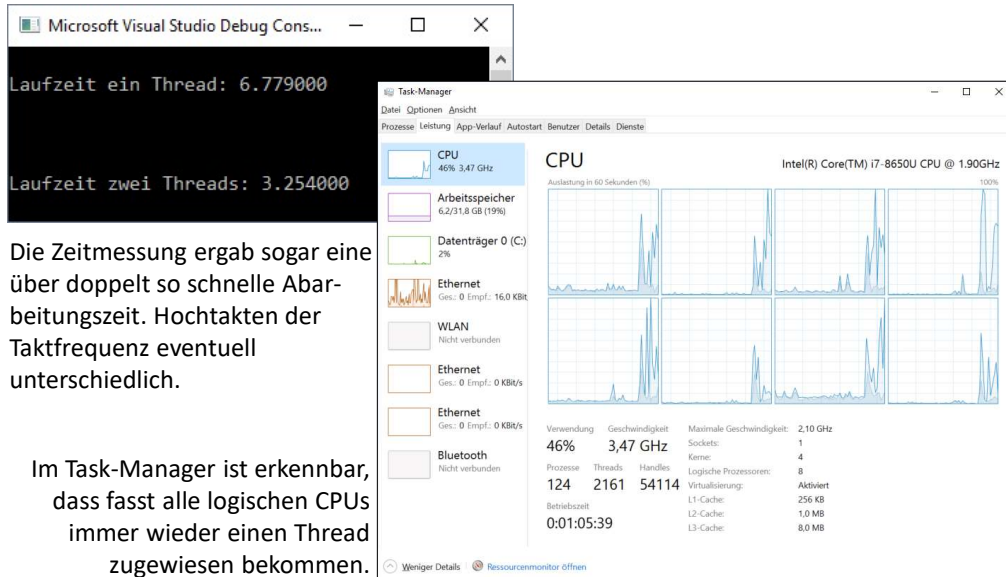
Synchronisation: Hauptthread wartet auf hThread1 und hThread2



## 2. Windows Thread API

17

### Beispiel: Synchronisation von Threads – Vergleich



Microsoft Visual Studio Debug Cons...

Laufzeit ein Thread: 6.779000

Laufzeit zwei Threads: 3.254000

Die Zeitmessung ergab sogar eine über doppelt so schnelle Abarbeitungszeit. Hochtakten der Taktfrequenz eventuell unterschiedlich.

Im Task-Manager ist erkennbar, dass fast alle logischen CPUs immer wieder einen Thread zugewiesen bekommen.

## 3. Typische Fallstricke mit Threads

18

### Fallstricke

Die Programmierung mit Threads setzt gewisses abstraktes Denkvermögen voraus. Hier sollen zwei typische Fallstricke kurz vorgestellt und an einem Beispiel erläutert werden:

1. Zugriff auf **globale** Variablen: Globale Variablen können von mehreren Threads verwendet werden. Die Zugriff auf globale Variablen ist nicht atomar. Dies führt meist zu falschen Ergebnissen, wenn auf eine globale Variable mehrfach ein Schreibzugriff erfolgt. Gleiches gilt für **modulglobale** Variablen!
2. Verwendung von **lokalen statischen** Variablen: Diese liegen ebenso im Datensegment und können – falls mehrere Threads gleichzeitig die Funktion aufrufen – zu falschen Ergebnissen führen. Dies kann auch geschehen, wenn ein Bibliotheksfunktion von mehreren Threads aufgerufen wird und diese intern lokale statische Variablen verwendet.

Auch sind in diesem Umfeld die Einstellungen unter Debug und Release sehr wichtig.

### 3. Typische Fallstricke mit Threads

19

#### 1. Zugriff auf globale Variablen

```
hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);


if ((hThread1 != 0) && (hThread2 != 0))
{
    ahThread[0] = hThread1;
    ahThread[1] = hThread2;
    WaitForMultipleObjects(2, (const HANDLE*)&ahThread, TRUE, INFINITE);
    printf("\n\nWert iGlobal: %i\n\n", iGlobal);
}
```

```
DWORD WINAPI ThreadFunction(void* pParam)
{
    int iJ;

    for (iJ = 0; iJ < 1000000; iJ++)
    {
        iGlobal++;
    }


    return 0;
}
```

#### Target Release



Microsoft Visual Studio De... — □ ×  
Wert iGlobal: 2000000

#### Target Debug



Microsoft Visual Studio De... — □ ×  
Wert iGlobal: 1027324

### 3. Typische Fallstricke mit Threads

20

#### 1. Zugriff auf globale Variablen

##### Target Debug



Microsoft Visual Studio De... — □ ×  
Wert iGlobal: 1027324

Optimization disabled  
(Properties -> Configuration Properties  
-> C/C++ -> Optimization -> „Disabled  
(/Od)“)


Damit in Multithreading-Anwendungen alle Threads immer den aktuellen Wert einer globalen Variablen haben, unterliegt diese einem Load-Modify-Store Zyklus (mov, add, mov). Der Befehl iGlobal++ ist somit auf Maschinencodeebene (Assembler) nicht atomar!

iGlobal++;

##### Assemblercode

```
mov eax,dword ptr [iGlobal (011EA138h)]
add eax,1
mov dword ptr [iGlobal (011EA138h)],eax
```

##### Target Release



Microsoft Visual Studio De... — □ ×  
Wert iGlobal: 2000000

Optimization auf „Maximum  
Optimization (Favor Speed) (/O2)“  
eingestellt.

Schleife wird komplett getilgt!

```
for (iJ = 0; iJ < 1000000; iJ++)
{
    iGlobal++;
}
```

##### Assemblercode

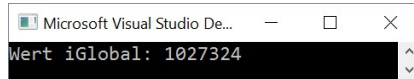
```
add dword ptr _iGlobal, 1000000
```

### 3. Typische Fallstricke mit Threads

21

#### 1. Zugriff auf globale Variablen – Beispiel nicht atomarer Befehl

##### Target Debug



Threads teilen sich das Datensegment und damit die globalen Variablen.

```
int iGlobal = 0;
```

##### Thread 1:

```
iGlobal++;
```

##### Assemblercode

```
mov eax,dword ptr [iGlobal (011EA138h)]
add eax,1
mov dword ptr [iGlobal (011EA138h)],eax
```

##### Thread 2:

```
iGlobal++;
```

##### Assemblercode

```
mov eax,dword ptr [iGlobal (011EA138h)]
add eax,1
mov dword ptr [iGlobal (011EA138h)],eax
```

Beide Threads haben die 0 am Anfang in das eax-Register geladen, Wert wird dort inkrementiert und wieder weggeschrieben. Es steht dann eine 1 statt einer 2 in der Variablen. Diese Problematik taucht dann häufig während der Laufzeit auf. Um dieses Problem zu umgehen, müsste vom Programm verhindert werden, dass dieses Verhalten verhindert wird. In C11 gibt es das Schlüsselwort `_Atomic` – was aber vom Microsoft C-Compiler (noch) nicht unterstützt wird. Abhilfe: Mutexe

### 3. Typische Fallstricke mit Threads

22

#### 1. Zugriff auf globale Variablen – Mutexe

```
HANDLE hMutex = 0;
```

```
int main(void)
```

```
{
    hMutex = CreateMutex(NULL, FALSE, NULL);
    //more Code
    if (hMutex != 0)
    {
        CloseHandle(hMutex);
    }
    return 0;
}
```

Der gleichzeitige Zugriff auf globale Variablen (und alle anderen Objekte, die sich Threads teilen) kann mit **Mutexen** verhindert werden.

Mit `WaitForSingleObject` wartet die Thread-Funktion, bis `hMutex` wieder frei ist und belegt diesen dann. Wird die Belegung nicht mehr benötigt, wird diese mit `ReleaseMutex` wieder freigegeben.

```
DWORD WINAPI ThreadFunction(void* pParam)
{
    int iJ;

    for (iJ = 0; iJ < 1000000; iJ++)
    {
        WaitForSingleObject(hMutex, INFINITE);
        iGlobal++;
        ReleaseMutex(hMutex);
    }

    return 0;
}
```

**Critical Section**

### 3. Typische Fallstricke mit Threads

23

#### 2. Zugriff auf statische lokale Variablen

```
DWORD WINAPI ThreadFunction(void* pParam)
{
    char* pcToken;
    DWORD dwThreadId;

    //ANSI-conform strtok
    pcToken = strtok((char*)pParam, ",");

    while (pcToken != NULL)
    {
        dwThreadId = GetCurrentThreadId();
        printf("ThreadID: %i String: %s\n", dwThreadId, pcToken);
        pcToken = strtok(NULL, ",");
    }

    return 0;
}
```

Die ANSI-konforme Implementierung von strtok verwendet einen lokalen statischen char-Pointer -> liegt im Datensegment

### 3. Typische Fallstricke mit Threads

24

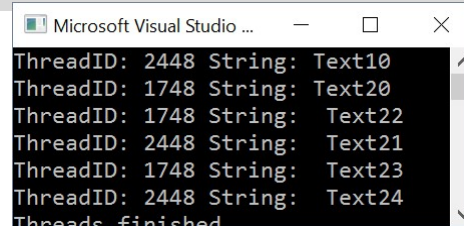
#### 2. Zugriff auf statische lokale Variablen

```
char acText1[] = "Text10; Text11; Text12; Text13; Text14";
char acText2[] = "Text20; Text21; Text22; Text23; Text24";

hThread1 = CreateThread(NULL, 0, ThreadFunction, acText1, 0, NULL);
hThread2 = CreateThread(NULL, 0, ThreadFunction, acText2, 0, NULL);

if ((hThread1 != 0) && (hThread2 != 0))
{
    ahThread[0] = hThread1;
    ahThread[1] = hThread2;
    WaitForMultipleObjects(2, (const HANDLE*)&ahThread, TRUE, INFINITE);
}
```

Thread mit der ID 1748 überschreibt den statischen Zeiger. Thread mit der ID 2448 arbeitet nun mit dem falschen Array!  
Microsoft hat dies in der eigenen Implementierung von strtok behoben, daher wird in diesem Beispiel eine Standard-konforme Implementierung von strtok verwendet.



Microsoft Visual Studio ...

```
ThreadID: 2448 String: Text10
ThreadID: 1748 String: Text20
ThreadID: 1748 String: Text22
ThreadID: 2448 String: Text21
ThreadID: 1748 String: Text23
ThreadID: 2448 String: Text24
Threads finished
```

### 3. Typische Fallstricke mit Threads

25

#### 2. Zugriff auf statische lokale Variablen – strtok\_s

Im vorliegenden Fall schafft hier Abhilfe die Verwendung der sicheren Funktion strtok\_s (strtok\_r unter GCC – „r“ für reentrant).

```
char* strtok(char* pcStr, const char* pccDelimiters);
```

```
char* strtok_s(char* pcStr, const char* pccDelimiters, char** ppcContext);
```

Während strtok einen lokalen static char-Zeiger (pcRemember in KE 11) verwendet, wird bei strtok\_s der Kontext (Zeiger auf Zeiger auf char) übergeben.

Da strtok\_s keinen lokalen static char-Zeiger mehr benötigt, ist diese Funktion "thread safe".

**Threadsicherheit** ist eine Eigenschaft von Softwarekomponenten und hat eine wichtige Bedeutung in der Softwareentwicklung. Sie besagt, dass eine Komponente gleichzeitig von verschiedenen Programmbereichen mehrfach ausgeführt werden kann, ohne dass diese sich gegenseitig behindern.

Quelle: wikipedia.org

### 3. Typische Fallstricke mit Threads

26

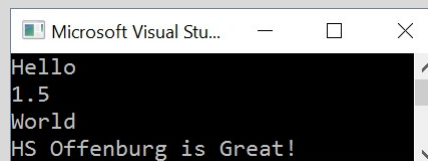
#### 2. Zugriff auf statische lokale Variablen – Vergleich strtok <-> strtok\_s

```
char acStringWithTokens[] = "Hello;1.5;World;HS Offenburg is Great!";  
char* pcToken;
```

```
pcToken = strtok(acStringWithTokens, ";");
```

```
while (pcToken != NULL)  
{  
    printf("%s\n", pcToken);  
    pcToken = strtok(NULL, ";");  
}
```

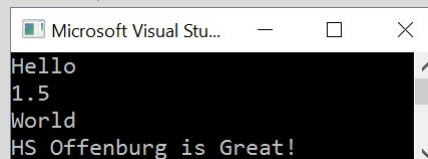
Beispiel aus KE 11



```
char acStringWithTokens[] = "Hello;1.5;World;HS Offenburg is Great!";  
char* pcToken;  
char* pcContext = NULL;
```

```
pcToken = strtok_s(acStringWithTokens, ";", &pcContext);
```

```
while (pcToken != NULL)  
{  
    printf("%s\n", pcToken);  
    pcToken = strtok_s(NULL, ";", &pcContext);  
}
```



## Behandelte Schlüsselwörter in KE 12

Schlüsselwörter C89:

<del>auto</del> ✓	do ✓	<del>goto</del> ✓	signed ✓	unsigned ✓
break ✓	double ✓	if ✓	sizeof ✓	void ✓✓
case ✓	else ✓	int ✓	static ✓✓	volatile ✓
char ✓	enum ✓	long ✓	struct ✓	while ✓
const ✓✓	extern ✓✓	register ✓	switch ✓	
<del>continue</del> ✓	float ✓	return ✓	typedef ✓	
default ✓	for ✓	short ✓	union ✓	

Schlüsselwörter ab C99:

_Bool ✓	_Complex ✓	_Imaginary ✓	inline ✓	restrict ✓
---------	------------	--------------	----------	------------

Schlüsselwörter ab C11:

_Alignas	_Alignof ✓	_Atomic ✓	_Generic	_Noreturn
_Static_assert	_Thread_local			