Thread-Synchronisation (IV)

```
#include <windows.h>   //genauer: winbase.h
```

Beendigung eines Threads auch durch einen anderen Thread prinzipiell möglich:

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

Parameter:

hThread: Thread-Kennung (Handle)

dwExitCode: Exit-Code des Threads
(Abfrage mit **GetExitCodeThread()**)

Rückgabewert: TRUE falls erfolgreich, FALSE sonst

Kommentar des Microsoft Developer Network (MSDN):

"*TerminateThread is a dangerous function that should only be used in the most extreme cases*" (Hinterläßt offene Critical Sections etc.)

Windows-Multithreading

Grundsätzliches zu Prozessen und Threads:

- Jedem Prozeß steht virtuell der ganze Rechner und der gesamte verfügbare Adreßraum zur Verfügung;

  so bleiben seine Daten privat (unsichtbar) und geschützt.

  (Bsp.: „Schreibprogramme“ Präsentation, Editor, Email, Textverarbeitung: ähnlich, aber gegeneinander geschützt)
- Private virtuelle Adreßräume laufender Prozesse werden durch das Betriebssystem mit einer Speicherverwaltungs-Hardware (*Memory Management Unit, MMU*) realisiert.

  Die MMU wird zur Prozeßumschaltung umprogrammiert.

  Dieser Mehraufwand entfällt bei Threads-Umschaltung:
- Threads sind parallel ablaufende Aktivitäten in einem Prozeß, d.h. im selben Adreßraum, mit denselben globalen Variablen.

Grundsätzliches zu Prozessen und Threads (Forts.):

- Jeder Prozeß, der neu erzeugt wird, enthält implizit mindestens einen Thread (*primary thread*, z.B.: **`main()`**):

  „Der Prozeß besitzt die Ressourcen, der Thread führt den Code aus.“

- Als Teil der sog. Prozeßumgebung nutzen Threads gemeinsam die verfügbaren Ressourcen des Prozesses:

  Kein Schutz der Daten gegen Einsicht und Veränderung!

- Jeder Thread besitzt seinen eigenen Kontrollfluß und seine eigene Aufrufhierarchie von Unterprogrammen.

  Dazu gehört nicht nur der aktuelle Inhalt des Programmzählers und des Stack-Zeigers, sondern auch ein eigener Stack. Bei jeder Thread-Umschaltung müssen alle Prozessorregister gesichert werden.

Windows-Multithreading

- Steigerung des Nutzens von Threads (vgl. „1/10-sec-Regel“) durch Prioritäten in der Zuteilungsstrategie (engl. *scheduling*)

  z.B.: Nach Eingabe eines Reiseziels, Auslagerung der Routen-Berechnung auf einen Thread mit (ggf. temporär) erhöhter Priorität (~ Rechenzeitzuteilung) zur Beschleunigung der Interaktion

- POSIX-Threads unterstützen Thread-Prioritäten:

  Bibliotheken mit minimal 32 Stufen, je nach Implementierung.

- Windows bietet 32 Prioritätsstufen in der Rechenzeitzuteilung;

  Niedrigste Stufe (0) ist nur für BS-interne Aktivitäten reserviert (insb. Kennzeichnung freigegebenen Speichers).

  Vergabe der Stufen 1…31 nicht direkt, sondern über 6 Prozeß-Prioritätsklassen (*priority classes*) mit je 7 Thread-Prioritätswerten (*thread priority values*).

  Stufen 16…31 können den Windows-Betrieb behindern (z.B. Start des Task-Managers bei Blockierungen). Grundsätzlich:

  Meist sind nur kurzzeitige Prioritätsänderungen sinnvoll.

Windows-Multithreading

- (Thread-) **Prioritätswert** und (Prozeß-) **Prioritätsklasse** bilden zusammen die **Basispriorität** (*base priority level*) des Threads.
- Prozesse werden mit **`NORMAL_PRIORITY_CLASS`**, Threads mit **`THREAD_PRIORITY_NORMAL`** gestartet (Voreinstellung)
- Ein Thread kann nur seine Relativpriorität wechseln (d.h. seinen Prioritätswert innerhalb derselben Prioritätsklasse).

process priority classes

thread priority values

	…_PRIORITY_CLASS					
THREAD_PRIORITY	**IDLE**	**BELOW_NORMAL**	**NORMAL**	**ABOVE_NORMAL**	**HIGH**	**REALTIME**
…_IDLE	1	1	**1**	1	1	16
…_LOWEST	2	4	**6**	8	11	22
…_BELOW_NORMAL	3	5	**7**	9	12	23
…_NORMAL	4	6	**8**	10	13	24
…_ABOVE_NORMAL	5	7	**9**	11	14	25
…_HIGHEST	6	8	**10**	12	15	26
…_TIME_CRITICAL	15	15	**15**	15	15	31

Windows-Multithreading

Setzen und Abfragen der Priorität eines Threads unter Windows:

```
#include <windows.h> //genauer: winbase.h

BOOL SetThreadPriority (HANDLE handle, int Prio);
int  GetThreadPriority (HANDLE handle);
```

Parameter:

handle: Thread-Handle (Rückgabewert von **_beginthread()**)

Prio: Gewünschte Priorität (**THREAD_PRIORITY_…**)

Rückgabewerte:

`SetThreadPriority()`: ungleich null (**TRUE**) bei Erfolg, sonst 0

`GetThreadPriority()`: aktuelle Prioritätsstufe (-15 bis +15)

Windows-Multithreading

Abfragen und Setzen der Prioritätsklasse eines Prozesses:

```
#include <windows.h> //genauer: winbase.h

HANDLE GetCurrentProcess (void);
DWORD  GetPriorityClass (HANDLE hProc);
BOOL SetPriorityClass (HANDLE hProc,DWORD dwPrioClass);
```

Parameter:

hProc: Prozeß-Handle (Rückgabewert von **GetCurrentProcess()**)

dwPrioClass: Gewünschte Prioritätsklasse (**…_PRIORITY_CLASS**)

Rückgabewerte:

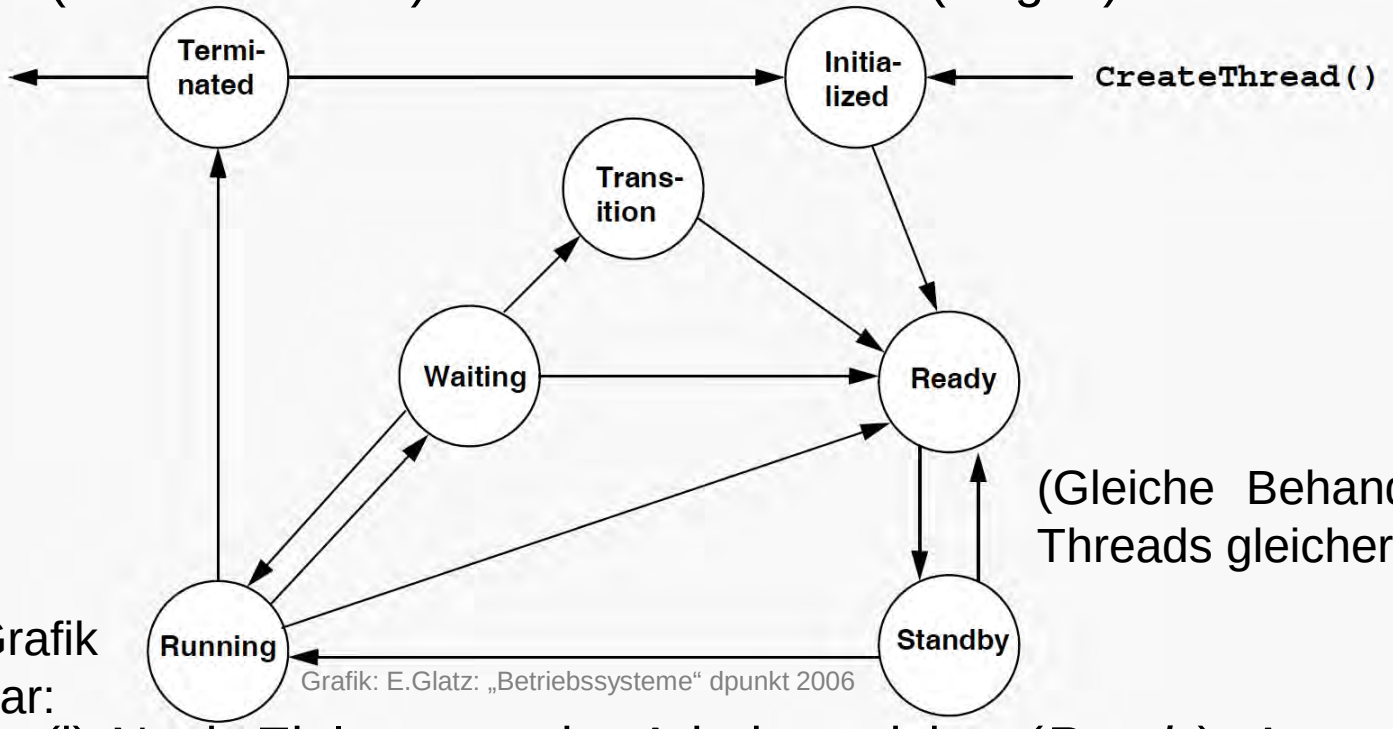
`GetCurrentProcess()`: (Pseudo-) Handle des aktuellen Prozesses
(MSDN: Aufruf statt Variable sichert zukünftig Kompatibilität)

`GetPriorityClass()`: Prioritätsklasse (bei Versagen: 0)

`SetPriorityClass()`: ungleich null (**TRUE**) bei Erfolg, sonst 0

Windows-Multithreading

- Windows-Scheduling im Round-Robin-Verfahren für Threads (oder Prozesse) in unterschiedlichen (insg. 7) Zuständen:



Grafik: E.Glatz: „Betriebssysteme“ dpunkt 2006

(Gleiche Behandlung für Threads gleicher Priorität)

In der Grafik erkennbar:

(i) Nach Einlagerung im Arbeitsspeicher (*Ready*), Auswahl und Einreihung (*Standby*) zur Ausführung (*Running*)

(ii) Rückführung von *Running* und *Standby* in *Ready*, falls „höherpriore“ Threads (aus *Waiting*) in *Ready* wechseln.

Anmerkungen zur Windows-Rechenzeitzuweisung:

- Die Priorisierung „hochpriorer“ Prozesse / Threads kann zur **Prioritätsumkehrung** (*priority inversion*) führen:

  Auslagerung eines Threads im kritischen Abschnitt (CS) kann bewirken, daß der unterbrechende Thread gerade auf die reservierte Ressource warten muß (Deadlock)

  Währenddessen werden evtl. Prozesse mittlerer Priorität vorgelassen, wodurch die Blockierung unerkannt bleibt.

  Windows-Taktik: In zufälligen Zeitabständen Priorisierung niederpriorer Threads, damit sie CS verlassen können.

- Multiprozessor-Scheduling ist nur z.T. eine Lösung:

  Zur optimalen Cache-Nutzung: Versuch, jeden Thread einer bestimmten CPU (*ideal processor*) zuzuordnen.

  ⇨ CPU-Wechsel erst nach Bedienung aller Wartenden

Windows-Multithreading

Grundsätzliches zu Prozessen und Threads (Forts.):

Threads eignen sich besser als Prozesse, um innerhalb einer Applikation parallele Aktivitäten durchzuführen, insb.

- um Mehrfachnutzung von Ressourcen zu ermöglichen

  z.B. Textverarbeitung: Text editieren / umbrechen / Rechtschreibung prüfen / durchsuchen / speichern
- um mehrere Anfragen an ein Programm zu bedienen

  z.B. Webserver („Thread-pro-Anfrage-Situation“):

  Daten-Abruf, -Suche, -Sortierung, -Übermittlung

Bei Multithreading rechnerintern zu unterscheiden:

- (UL-) **User-Level-Thread**:

  vom Programmierer entworfener Thread
- (KL-) **Kernel-Level-Thread**:

  auf der CPU tatsächlich ausgeführter Thread

Windows-Multithreading

Kategorisierung nach „Thread-Kardinalitäten“

(d.h.: Zuordnungsverhältnis UL : KL)

- **1:1**: Multithreading findet auf Systemebene statt.
- **m:1**: Alle UL-Threads auf Benutzerebene zu einem KL-Thread zusammengefaßt (meist mit Sw-Bibliotheken)
  - für Betriebssystem: keine Thread-Applikation
  - Rechenzeit-Konkurrenz unter Threads nur eines Prozesses
  - Umschaltung schneller als bei KL-Threads
  - BS-Aufrufe aus einem Thread können alle anderen blockieren
  - Nutzung nur eines Prozessors
- **m:n**: (m > n) Hybridlösung
  - Möglichkeit zur optimierten Rechenzeit-Zuteilung unter konkurrierenden Threads einer Applikation

  (Alle Kardinalitäten sind in der POSIX-Norm zulässig.)

vgl. kooperatives Multitasking

Windows-Multithreading

Mit Win 2000/XP eingeführt: sog. **Fibers** („Fasern“):

Ein Fiber

- hat eigenen **Stack**, eigene **Startadresse** und eigene **Daten**
- ist ein reiner **User-Level-Thread** (UL-Thread) – d.h.:
  Die Prozessorzuteilung muß in d. Applikation realisiert werden.

Die Applikation, selbst ein Thread, muß erst selbst in einen Fiber konvertiert werden, damit sie einen (zweiten) Fiber einrichten kann:

```
ConvertThreadToFiber (lpParameter);
CreateFiber (dwStackSize, lpStartAddress, lpParameter);
```

Parameter:

`lpParameter`: 32-Bit-Adressen (`LPVOID`) auf untypisierte Daten

`dwStackSize`: Startwert Stackgröße (**`0`**: Standard ⇨ autom. Anpassung)

`lpStartAddress`: Fiber-Startadresse (`LPVOID`)

Rückgabewerte: jeweils 32-Bit-Adressen (`LPVOID`): Fiber-Adresse

Windows-Multithreading

Prozessor-Umschaltung von Fiber zu Fiber:

```
SwitchToFiber (lpFiber);
```

Terminierung eines anderen Fibers (oder des Threads selbst):

```
DeleteFiber (lpFiber);
```

Parameter:

lpFiber: Fiber-Adresse (32-Bit, untypisiert: **LPVOID**)

Rückgabewerte: keine (**void**)

Abfragen:

```
GetFiberData();   //fragt eigenen Parameter ab
GetCurrentFiber();//Adresse gerade ausgefuehrten Fibers
```

Rückgabewerte: untypisierte 32-Bit-Adressen (LPVOID)

Beispiel:

```
/*… main() …*/
LPVOID lpFiber[2]; /*(...)*/
lpFiber[0] = ConvertThreadToFiber(NULL); //werde Fiber!
lpFiber[1] = CreateFiber(0, FiberFunc, (void *)data);
SwitchToFiber (lpFiber[1]);
/*… (weiter mit veraenderter Variable data) …*/
```

Anmerkungen:

- Fibers sind für das BS unsichtbar (und schneller)
- Grundsätzlich zwei Prozessorzuteilungsstrategien:

  Beim *Master-slave-scheduling* übernimmt ein *Master-Fiber* die Steuerung der Fiber-Umschaltung

  Beim *Peer-to-peer-scheduling* gibt jeder Fiber selbständig die Kontrolle an einen anderen Fiber weiter