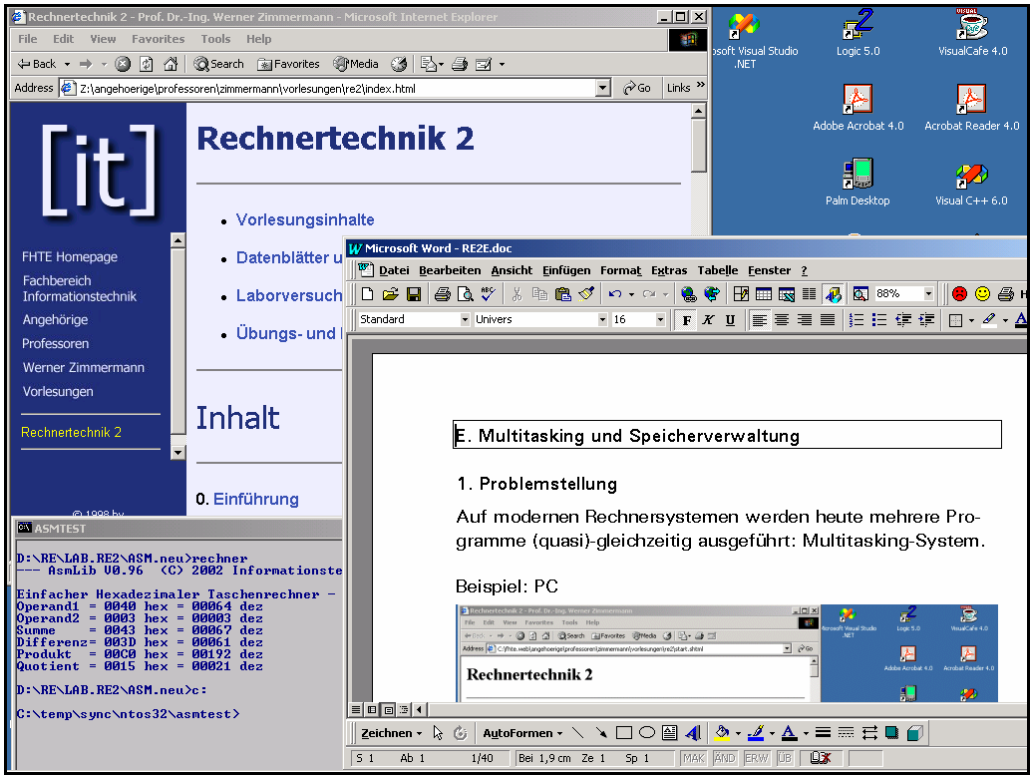
Multitasking und 80x86 Protected Mode unter Windows

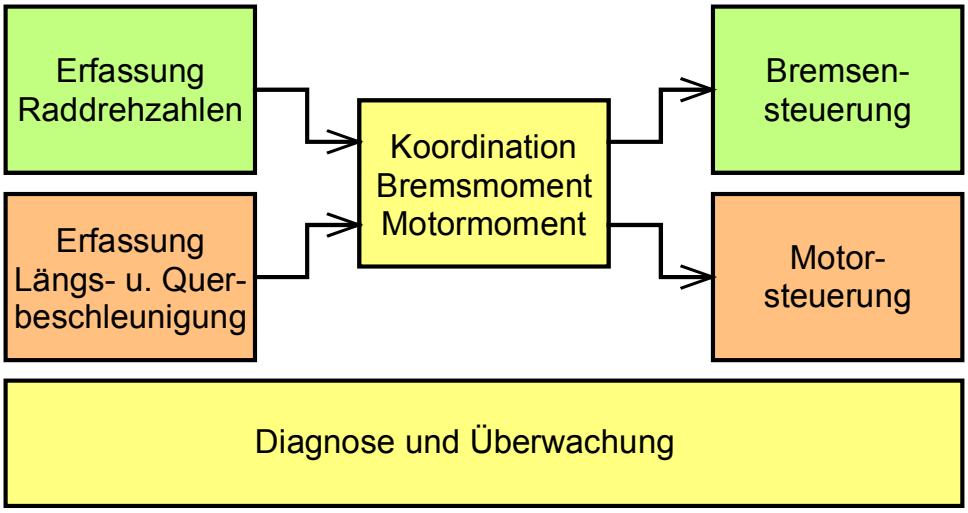
1. Problemstellung

Auf modernen Rechnersystemen werden heute mehrere Programme (quasi)-gleichzeitig ausgeführt: Multitasking-System.

Beispiel: PC-Anwendungsprogramme unter Windows



Beispiel: Embedded-Control-System ABS/ESP



Problematik

- **Rechenzeit** muss auf die (quasi)-gleichzeitig laufenden Programme aufgeteilt werden, da in Wirklichkeit nur eine einzige CPU vorhanden ist.
- **Speicherplatz** muss auf die (quasi)-gleichzeitig laufenden Programme aufgeteilt werden. Eventuell ist der Speicherplatzbedarf größer als der tatsächlich vorhandene Speicher.
- **Schutz** der Programme voreinander und Schutz des Betriebssystems vor den Programmen. Ein Programm sollte nicht unkontrolliert im Speicherbereich eines anderen Programms schreiben oder lesen können oder das andere Programm oder den gesamten Rechner blockieren können.

Diese 3 Aufgaben des Betriebssystems, können von diesem nur dann zuverlässig durchgeführt werden, wenn es durch Hardware-Mechanismen der CPU unterstützt wird. Beispiele:

- **Rechenzeit-Zuteilung (Scheduling)**: Sobald das Betriebssystem ein Anwendungsprogramm gestartet hat, kann das Anwendungsprogramm nur freiwillig ("**kooperativ**") **oder** durch einen Zeitgeber-Interrupt ("**präemptiv**") unterbrochen oder beendet und ein anderes Programm gestartet werden. Falls das Anwendungsprogramm aber das IF-Flag selbst sperren oder den Zeitgeber-Interrupt umprogrammieren kann, kann es die Rechenzeit-Zuteilung des Betriebssystems aushebeln. D.h. in einem sicheren Betriebssystem dürfen **Anwendungsprogramme keinen direkten Zugriff auf Interrupts und die Hardwarebausteine, die die Interrupts erzeugen, haben („privilegierte Befehle“)**. Die Überwachung muss durch die CPU-Hardware automatisch erfolgen
- **Speicherplatz-Zuteilung (Memory Management)**: Das Betriebssystem teilt jedem Anwendungsprogramm einen bestimmten **Speicherplatz** zu, **auf den nur dieses Anwendungsprogramm zugreifen darf.** Die Überwachung muss durch die CPU-Hardware automatisch erfolgen.

2. Lösungsansätze

Theoretische Möglichkeiten der Speicherverwaltung

- Ein einziger **gemeinsamer Adressraum für alle Programme**
  - Alle **Programme verwenden absolute physikalische Speicheradressen**. Die Adressen werden bereits **beim Übersetzen oder beim Laden** der Programme durch das Betriebssystem **festgelegt**.

⇒ Nachteil: **Sehr unflexibel**. Dynamisches Verschieben von Programmen/Daten praktisch nicht möglich. Bei Änderungen müssen gegebenenfalls die Adressen aller Programme neu angepasst werden.

- **Segmentierung**
  - Der Adressbereich wird in einzelne **Speichersegmente mit bedarfsabhängiger Segmentlänge** unterteilt.
  - **Programme arbeiten** intern **nur mit** relativen Adressen (**Offsetadressen**) bezogen auf den jeweiligen Segmentanfang (Basisadresse).
  - Auswahl des jeweils aktiven Segmentes erfolgt über ein CPU-Segment-Register. Die **Basisadresse** wird vom Betriebssystem **beim Laden** des Programms **festgelegt** und in eine Segment-Tabelle (oder das Segment-Register selbst) eingetragen.
  - Bei jedem Speicherzugriff addiert die Adresseinheit der CPU die Basisadresse und Offsetadresse zur tatsächlichen Speicheradresse
  - Die Segment-Tabelle kann **eventuell zusätzliche Segmenteigenschaften** (Länge, Zugriffsrechte usw.) enthalten, die von der CPU bei jedem Speicherzugriff überprüft werden.
  - **Segmente** können nicht nur für unterschiedliche Programme sondern innerhalb eines Programms **auch zur Trennung von Code und Daten** verwendet werden (logische Harvard-Architektur).

⇒ Vorteil: **Sehr flexibel für den Anwendungsprogrammierer**

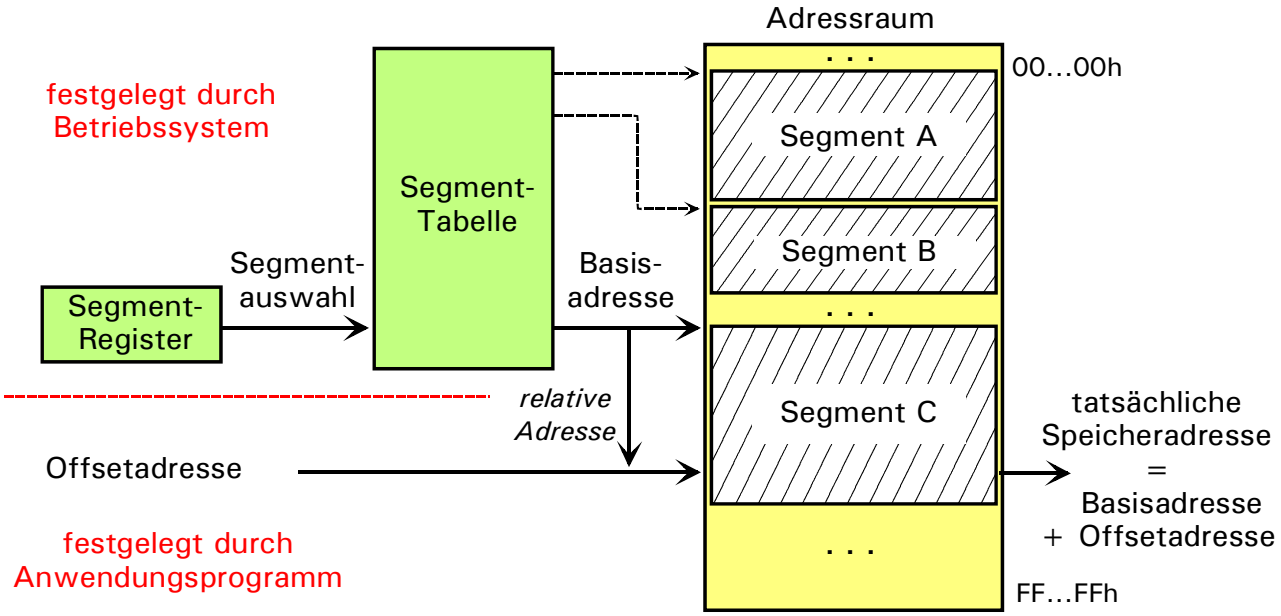
⇒ Nachteil: **Schwierige Speicherverwaltung für das Betriebssystem**. Durch unterschiedliche Segmentlängen zerstückel-

ter Adressraum. Dynamisches Ändern der Segmentlänge schwierig.

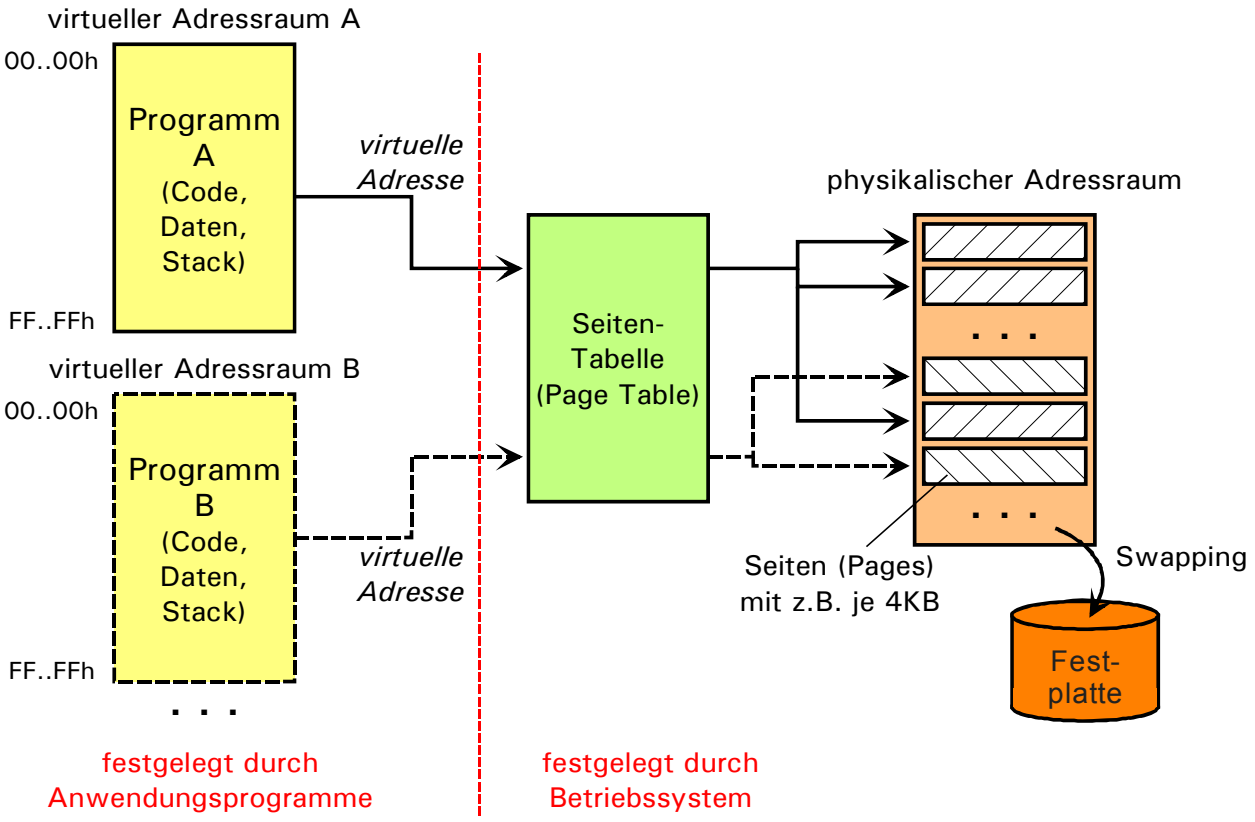
- **Paging**

- **Jedes Programm** arbeitet **mit** seinem **eigenen virtuellen Adressraum**. Die virtuellen Adressen werden **über** eine **Umsetzungstabelle** (**Seiten-Tabelle, Page Table**) **in** den **physikalischen Speicheradressraum abgebildet**.
- Damit die Umsetzungstabellen nicht zu gross werden, wird der Adressraum in viele kleine **Speicherbereiche konstanter Größe**, z.B. 4KB (**Seiten, Pages**) unterteilt. Die Adressumsetzung erfolgt auf Seiten-Ebene.
- Ein Programm kann zwar theoretisch den gesamten virtuellen Adressraum, z.B. 4GB verwenden, erhält aber nur soviel physikalischen Speicher, wie tatsächlich benötigt wird.
- Falls die Programme zusammen mehr virtuellen Speicher benötigen als physikalisch vorhanden ist, werden **Teile des virtuellen Speichers auf** die **Festplatte ausgelagert** (**Swapping**).
- In den Seitentabelle können eventuell zusätzliche Eigenschaften für die Speicherbereiche (Zugriffsreche usw.) festgelegt werden.
- Physikalische **Speicherbereiche**, die von mehreren Programmen **gemeinsam genutzt** werden sollen, z.B. Betriebssystemfunktionen, werden **über** die **Seitentabellen in mehrere Anwendungsprogramme eingeblendet** (**Shared Memory**)
- Die Seiten-Tabelle kann **zusätzliche Seiteneigenschaften** (Zugriffsrechte) enthalten, die von der CPU bei jedem Speicherzugriff überprüft werden.
- Vorteil: **Sehr flexibel für das Betriebssystem. Transparent für Anwendungsprogramme**, d.h. Anwendung muss sich nicht um die Speicherverwaltung kümmern.
- Nachteil: Relativ **umfangreiche Seiten-Tabellen**. Swapping ist zeitintensiv.

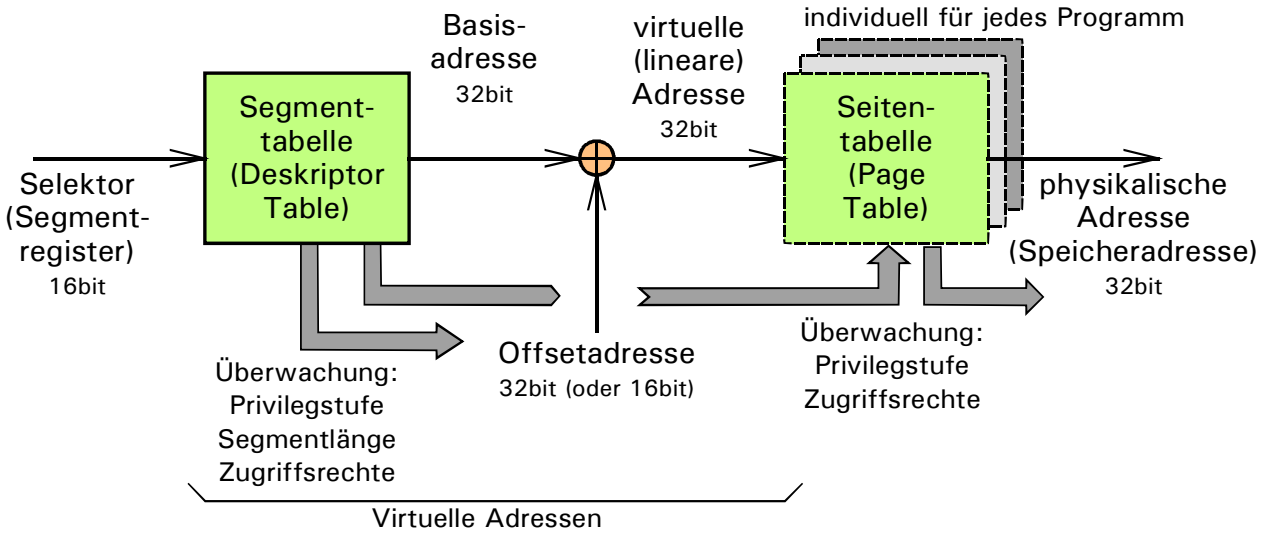
Segmentierung



Paging



Grundprinzipien 80x86 unter Windows und Linux

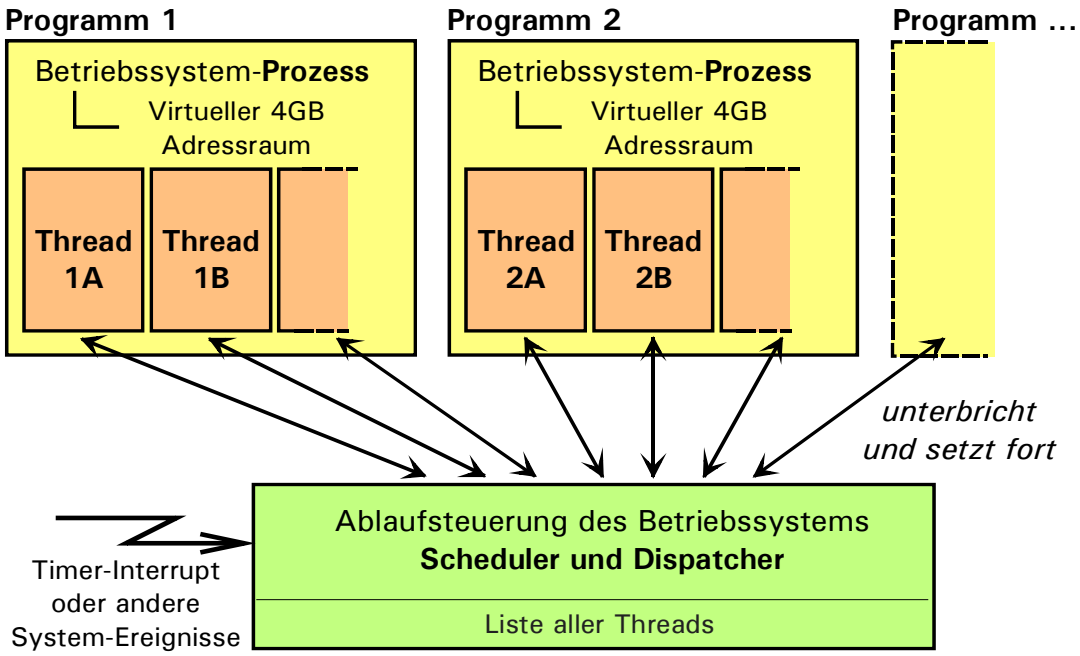


- **Paging mit Swapping**
- **Code, Daten und Stack eines Programms** befinden sich **in einem einzigen virtuellen 4GB-Adressraum** (Beginn bei der 32bit Basisadresse 0 und im Programm durch die 32bit-Offsetadresse in EIP, ESP usw. adressiert).
- **Jedes Programm** hat seinen **eigenen virtuellen 4GB-Adressraum**, d.h. seine eigenen Seitenumsetzungstabellen (**Seitentabellen = Page Tables**). Davon sind die oberen 2GB für das Betriebssystem (Kernel Mode sh. unten) reserviert.

- **Schutzmechanismen mit Privilegstufen** (über die Segment-Tabelle)
- Das Betriebssystem und die Anwenderprogramme erhalten zwei verschiedenene **Privilegstufen** (**Kernel Mode** und **User Mode**). Die **Zuordnung** der Privilegstufe erfolgt **über** die **Segment-Register und** die **Segment-Tabelle**.
- In den Seitentabellen wird **für jede Speicherseite** eingetragen, ob ein **Zugriff** nur **im Kernel Mode oder** auch **im User Mode** möglich ist und ob nur Lesen oder auch Schreiben zulässig ist (**Zugriffsschutz, Schreibschutz**).
- Bestimmte CPU-Befehle, z.B. STI/CLI, INT, IN/OUT, Laden einer neuen Seitentabelle usw., können nur im Kernel Mode ausgeführt werden (**Priviligierte Befehle, Befehlsschutz**).
- Bei **Verstoß gegen** die **Schutzmechanismen** wird ein Ausnahmefehler (**Exception Interrupt** 13_D oder 14_D) ausgelöst.

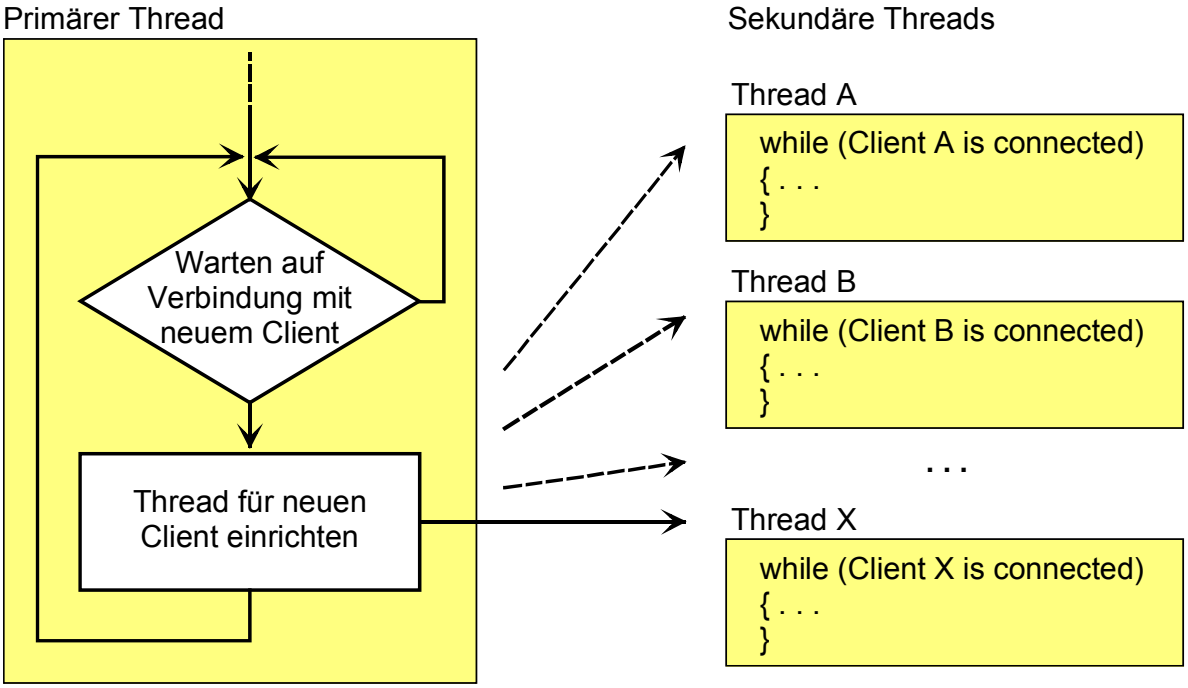
3. Multitasking unter Windows NT/2000/XP

3.1 Grundprinzip



- **Von einander** (weitgehend) **unabhängige Programme** werden vom Betriebssystem **als Prozesse** verwaltet. **Jedem Prozess** wird vom Betriebssystem ein **eigener virtueller Adressraum** (und eine Reihe weiterer Ressourcen, z.B. die Standard-Ein/Ausgaben) **zugeordnet**.
- Da jeder Prozess seinen eigenen Adressraum hat, ist ein **Datenaustausch zwischen** den **Prozessen** oder der Aufruf eines Unterprogramms in einem anderen Prozess nur über **aufwendige und langsame** Betriebssystem-Mechanismen wie Messages, Pipes, Shared Memory, Mailboxes/Mailslots, Sockets, Remote Procedure Call o.ä. möglich (**Inter-Prozess-Kommunikation**).
- Zur **Parallelisierung von Aufgaben innerhalb eines Programms** existieren **Threads**, d.h. vom Ablauf her zusammen--hängende Programmteile. Da alle Threads eines Programms innerhalb desselben virtuellen Adressraums ablaufen, kann der Datenaustausch bzw. Aufruf von Unterprogrammen mit globalen Variablen oder durch Aufrufparameter sehr einfach und schnell erfolgen.

Beispiel für Anwendung von Threads: Web-Server



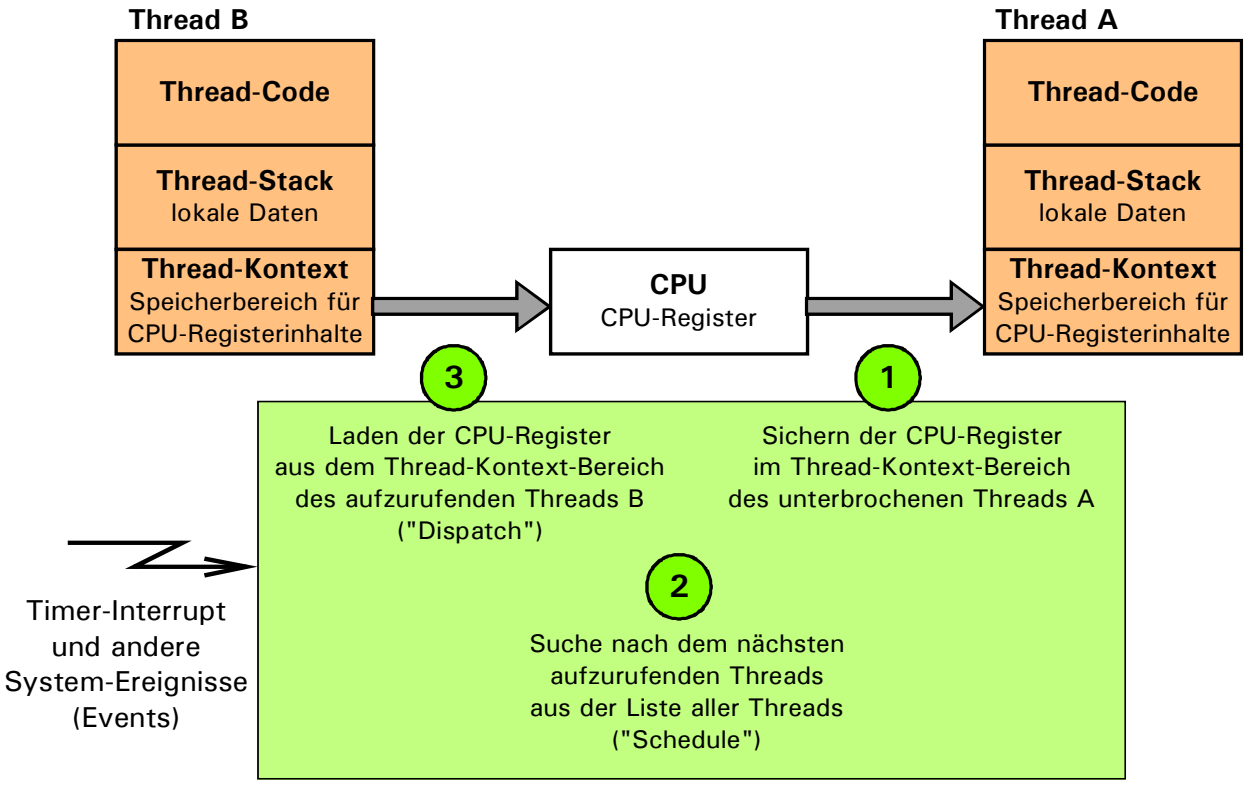
- Im primären Thread des Programms **wartet** der Server **auf** neu **eingehende Client-Verbindungen**.
- **Sobald** ein Client eine **Verbindung** aufnimmt, **erzeugt** der Server einen weiteren **Thread**, der dann die Verbindung mit dem Client abwickelt, bis der Client die Verbindung wieder beendet. Dieser Thread läuft dann parallel zum primären Thread und zu den anderen Threads, die ebenfalls Client-Verbindungen bedienen.
- Der **primäre Thread** dagegen **wartet sofort wieder auf neue** ankommende **Verbindung**en. Durch diese Struktur kann der Server gleichzeitig mehrere Clients bedienen.

Definition: Thread

Ein Thread ist ein **sequenziell ausgeführter Teil eines Programms, der aus einer oder mehreren Funktionen besteht**, die sich gegenseitig aufrufen, und der zeitlich quasi-parallel zu anderen Programmteilen ausgeführt wird.

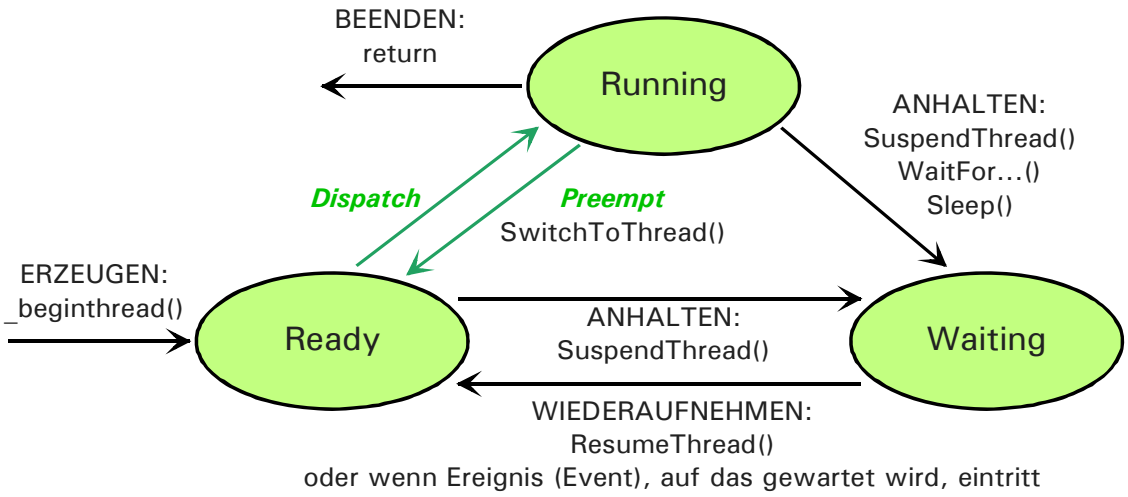
Solange nur eine CPU vorhanden ist, wird die **zeitliche Parallelität** dadurch **nachgebildet**, dass das **Betriebssystem** häufig **zwischen den einzelnen Threads hin- und herschaltet**.

Umschalten zwischen Threads („Context-Switch“)



Zustände eines Threads

Das Auslösen der Zustandsübergänge erfolgt durch das Betriebssystem bzw. durch den Aufruf von C- und WIN32-API-Funktionen im Anwendungsprogramm.



- Wenn ein Programm gestartet wird, besteht es zunächst aus einem einzigen Thread (**primärer Thread**). Durch Aufruf der Funktion _beginthread() kann es selbst weitere (**sekundäre**) **Threads** erzeugen. Neue Threads werden vom Betriebssystem in eine Liste aller vorhandenen Threads aufgenommen und in den **Zustand 'Ready'** versetzt.

- Jedem Thread ist neben seinem Programmcode ein eigener Stack sowie ein Speicherbereich zugeordnet, in dem die Registerzustände des Threads gespeichert werden können (**Thread Context**).
- Der Thread, dessen Programmcode gerade von der CPU ausgeführt wird, befindet sich im **Zustand 'Running'**.
- Ausgelöst durch einen Zeitgeber-Interrupt oder ein anderes Ereignis unterbricht das Betriebssystem den gerade ausgeführten Thread ("**Preempt**") und überprüft, welcher Thread jetzt ausgeführt werden soll ("**Schedule**"). Wenn ein Thread unterbrochen wird, werden die CPU-Register im Kontext-Speicherbereich des Threads gesichert. Falls ein anderer Thread ausgeführt werden soll, werden die gesicherten Registerinhalte dieses Threads aus dessen Kontext-Speicherbereich in die CPU-Register geladen (**"Dispatch"**, **Context Switch**). Dabei wird auch EIP geladen, so dass dieser Thread an der Stelle fortgesetzt wird, an der er früher unterbrochen wurde.

  Hinweis: Unter Windows gibt es keine Möglichkeit, einen Thread direkt unter Umgehung des Schedulers zu starten.
- Ein Thread kann sich selbst oder einen anderen Thread auch anhalten (**Zustand 'Waiting'**). Threads im Zustand 'Waiting' werden beim Scheduling nicht berücksichtigt. Ein angehaltener Thread kann entweder durch einen anderen Thread (mit ResumeThread()) oder automatisch durch Eintreffen eines Ereignisses ("**Event**"), z.B. Ablauf einer Wartezeit, wieder in den Zustand 'Ready' versetzt werden.

Betriebssysteme, bei denen Threads zwangsweise unterbrochen werden, bezeichnet man als Systeme mit '**präemptivem Multitasking**' (Beispiel: Windows NT/2000/XP). Systeme, bei denen ein Context-Wechsel nur dann erfolgt, wenn ein Thread freiwillig den Scheduler aufruft, bezeichnet man als Systeme mit '**kooperativem Multitasking**' (Beispiel: Windows 3.1).

Beispiel: ***Thread1.cpp***

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <process.h>
. . .
HANDLE hThreadA, hThreadB;
int iA = 0, iB = 0;

void threadA(int& i)    //*********** Sekundärer Thread A **************
{   printf("----- Thread A gestartet -----\n");
    SuspendThread(hThreadA);        // Thread A anhalten                3a

    while(1)                        // Endlosschleife
    {   i++;                        // Zaehler A inkrementieren
    }
}
void threadB(int& i)    //*********** Sekundärer Thread B **************
{   printf("----- Thread B gestartet -----\n");
    ResumeThread(hThreadA);         // Thread A fortsetzen              3b

    while(1)                        // Endlosschleife
    {    i++;                       // Zaehler B inkrementieren
    }
}
void main(void)         //****** Hauptprogramm (primärer Thread) ******
{                                   // Thread A und B erzeugen          1
    hThreadA =
         (HANDLE) _beginthread((void (*)(void *)) threadA, 0, (void *) &iA);
    hThreadB =
         (HANDLE) _beginthread((void (*)(void *)) threadB, 0, (void *) &iB);

    printf("----- Hauptprogramm -----\n");
    SwitchToThread();               // Scheduler aufrufen               2

    while(!kbhit())                 // Schleife bis Taste gedrueckt wird
    {
         printf("Thread A: %8Xh   Thread B: %8Xh\n", iA, iB);
         Sleep(1000);               // 1000ms Wartezeit                 4
    }
}
```

Im Hauptprogramm werden zwei sekundäre Threads A und B erzeugt. In diesen beiden Threads wird in einer Endlosschleife

jeweils ein Zähler inkrementiert. Der Zählerstand wird vom primären Thread (Hauptprogramm) jede Sekunde ausgegeben:

Ausgabe Thread1.cpp

```
----- Hauptprogramm -----
----- Thread A gestartet -----
----- Thread B gestartet -----
Thread A:   67BDB5h   Thread B:   2B83FDh
Thread A:  1C10FBCh   Thread B:  184659Dh
Thread A:  2F85883h   Thread B:  2DA6A9Dh
Thread A:  42832DBh   Thread B:  446CF2Eh
Thread A:  58A1D64h   Thread B:  58E2D5Bh
Thread A:  6C9328Bh   Thread B:  6E906F9h
Thread A:  8262C78h   Thread B:  82AE163h
Thread A:  97422E4h   Thread B:  9848083h
Thread A:  AADE723h   Thread B:  ADD4058h
. . .
```

Wie man an den Zählerständen sieht, zählen beide Threads ungefähr gleich schnell, d.h. das Betriebssystem lässt beide Threads ungefähr gleich oft und gleich lang laufen.

1 `_beginthread(void *threadFunction, int stackSize, void *p)`

Startet die Funktion `threadFunction()` als sekundären Thread. Die Stackgröße kann vorgegeben werden, `stackSize=0` verwendet einen Defaultwert. `_beginthread()` liefert eine 32bit Kennnummer für den Thread, ein sogenanntes Handle, zurück.

Die Thread-Funktion wird vom Betriebssystem mit einem einzelnen Pointerparameter *p aufgerufen : `void threadFunction (void *p)`

Über den Pointer *p kann ein einzelner 32bit-Wert oder ein Pointer auf einen beliebigen Parameter oder eine Parameter-Struktur übergeben werden. Um innerhalb der Thread-Funktion nicht ständig Typumwandlungen vornehmen zu müssen, ist der Parameter der Funktion threadA bzw. threadB hier als Referenzparameter auf einen Integerwert `int& i` definiert. Beim Erzeugen des Theads werden explitzite Typumwandlungen (Type Cast) für die Thread-Funktion und den Parameter verwendet:

```
_beginthread( (void(*)(void *)) threadA, 0, (void *) &iA )
```

Im Gegensatz zum Beispiel, bei dem jeder Thread eine andere Thread-Funktion hat, kann auch für mehrere Threads dieselbe Thread-Funktion verwendet werden. Die Threads und ihre jeweiligen lokalen Variablen sind trotzdem unabhängig voneinander und können über *p mit unterschiedlichen Parametern versorgt werden.

2 SwitchToThread()

Da mit _beginthread() zwar Threads erzeugt und in den Zustand 'Ready' versetzt, aber nicht unbedingt sofort gestartet werden, ruft das Hauptprogramm mit SwitchToThread() explizit den Scheduler auf und versetzt sich selbst damit vom Zustand 'Running' in den Zustand 'Ready'.

3 SuspendThread(thread_handle) - ResumeThread(thread_handle)

Da die beiden sekundären Threads dieselbe Priorität aufweisen (siehe Abschnitt 3.2) wird der zuerst erzeugte Thread A auch vor dem Thread B gestartet. Der Thread A versetzt sich selbst mit SuspendThread() in den Zustand 'Waiting', wodurch wiederum der Scheduler aktiviert und von diesem dann Thread B gestartet wird. Dieser aktiviert dann mit ResumeThread() auch wieder Thread A. Danach laufen die beiden Threads A und B quasi-parallel und inkrementieren ihre jeweiligen Zählvariablen.

Mit SuspendThread() kann ein Thread sich selbst oder jeden anderen Thread, dessen Handle als Parameter angegeben wird, in den Zustand 'Waiting' versetzen. Falls der Handle des eigenen Threads nicht bekannt ist (z.B. beim primären Thread), kann er über GetCurrentThread() ermittelt werden.

4 Sleep(wartezeit_in_msec)

Das Hauptprogramm (primärer Thread) gibt in einer Schleife die beiden Zählerstände auf dem Bildschirm aus. Damit die Bildschirmanzeige sich nicht zu schnell ändert, pausiert das Hauptprogramm für ca. 1sec. Sleep() ruft den Scheduler auf und versetzt den aufrufenden Thread in den Zustand 'Waiting'. Wenn die Wartezeit abgelaufen ist, wird der Thread automatisch wieder in den Zustand 'Ready' versetzt.

Hinweis: Die angegebene Zeit ist eine Mindestzeit, die Auflösung beträgt ca. 10ms. Die tatsächliche Wartezeit kann auch länger sein.

3.2 **Scheduling-Algorithmen** (Zuweisung der CPU)

Für unterschiedlichste Anwendungsfälle existiert eine Vielzahl von Strategien, aus der Liste aller Threads im Zustand 'Ready' denjenigen Thread auszuwählen, der tatsächlich als nächster ausgeführt wird. Die beiden wichtigsten Verfahren für präemptives Multitasking sind:

- **Round Robin** (Zeitscheibenverfahren)
  Jeder Thread darf für eine feste Zeit ("Zeitscheibe") laufen, bevor der Scheduler wieder aufgerufen wird, z.B. für 20ms.
- **Priority Based** (Prioritätsgesteuerertes Verfahren)
  Jeder Thread erhält eine Priorität. Es wird jeweils derjenige Thread vom Scheduler aktiviert, der die höchste Priorität hat. Die Prioritäten der Threads können entweder beim Programmentwurf bzw. -start fest vorgegeben werden (**statische Priorität**) oder während des Programmablaufs vom Betriebssystem oder vom Anwendungsprogramm verändert werden (**dynamische Priorität**). Wenn mehrere Threads mit gleicher Priorität existieren, muss zwischen diesen Threads nach einem anderen Verfahren, z.B. Round Robin, entschieden werden.

Der Scheduling-Algorithmus von Windows ist nicht in vollem Umfang dokumentiert, Microsoft nimmt an dieser Stelle bei jeder Version Veränderungen vor. Bekannt ist, dass

- Windows 31 **Prioritätsstufen** verwendet, wobei die Stufen 1 (niedrigste Priorität) ... 15 für Anwenderprogramme vorgesehen sind, während die Stufen 16 ... 31 (höchste Priorität) für Betriebssystem-Threads verwendet werden.
- Der Windows-Scheduler sucht grundsätzlich unter allen Threads, die sich im Zustand ‚Ready' bzw. ‚Running' befinden, den Thread, der die momentan höchste Priorität hat und aktiviert diesen (**Priority Based**).

- Wenn es mehrere Threads mit derselben (momentan höchsten) Prioritätsstufe gibt, wird jeder dieser Threads für eine feste Zeitdauer (typ. 30ms) aktiviert (**Round Robin**).

  Um dem Anwender das Gefühl einer schnelleren Reaktion des Rechners zu geben, wird diese Zeit für denjenigen Thread verdoppelt, dessen Bildschirmfenster gerade aktiv, d.h. angeklickt ist.

- Um sicherzustellen, dass alle Threads irgendwann tatsächlich aktiviert werden, verändert Windows die **Prioritäten** eines Threads **dynamisch**. Je länger ein Thread trotz des Zustandes ‚Ready' beim Scheduling nicht zum Zug kommt, desto weiter erhöht das Betriebssystem die Priorität dieses Threads. Entsprechend wird die Priorität eines Threads, der auf ein bestimmtes Ereignis wartet oder eines Threads, der gerade eine Tastatur- oder Mauseingabe entgegengenommen hat, dynamisch erhöht, sobald dieses Ereignis eingetroffen ist („**Priority Boost**").

Aufgrund der dynamischen Veränderung der Prioritäten und der Zeitscheiben sowie der Möglichkeit, dass der Betriebssystemkern oder vom Administrator installierbare Gerätetreiber für eine undefinierte Zeitdauer alle Interrupts sperren, kann Windows nicht zuverlässig garantieren, wann ein bestimmter Thread aktiviert wird. **Windows ist** daher **kein „hartes" Echtzeit-(Real Time)-Betriebssystem**. Von einem Echtzeit-Betriebssystem, wie es z.B. in Kfz-Steuergeräten eingesetzt wird, wird verlangt, dass das Scheduling so erfolgt, dass,

- präzise vorhergesagt werden kann, wie lange es höchstens dauert, bis nach einem Interrupt oder einem anderen Ereignis die dafür zuständige Routine aufgerufen wird (**Latency**),
- präzise garantiert werden kann, dass ein Thread bis zu einem bestimmten Zeitpunkt spätestens aktiviert wird und bis zu einem bestimmten Zeitpunkt („**Deadline**") seine Aufgaben erledigt hat (**„R"Echtzeitigkeit**).

  Bsp.: Bei der Steuerung einer elektrohydraulischen Kfz-Bremse muss unter allen Umständen garantiert werden, dass innerhalb einer definierten Zeit nach dem Betätigen des Bremspedals (=Eintreffen des Bremssignals am Steuergerät) durch Betätigen der Bremsventile tatsächlich die Bremse aktiviert wird.

Beispiel: ***Thread2.cpp***

```
. . .
//***** Thread-Funktionen threadA() und threadB() wie in Thread1.cpp *****
. . .
void main(void)
{   hThreadA = (HANDLE) _beginthread((void (*)(void *)) threadA, ... );
    SetThreadPriority(hThreadA, THREAD_PRIORITY_HIGHEST);                  1a

    hThreadB = (HANDLE) _beginthread((void (*)(void *)) threadB, ... );
    SetThreadPriority(hThreadB, THREAD_PRIORITY_LOWEST);                   1b

    printf("----- Hauptprogramm -----\n");
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);         1b
    . . .
}
```

1 SetThreadPriority(thread_handle, priorität)

Mit diesem Befehl kann ein Anwenderprogramm die Priorität eines Threads verändert. Der Defaultwert für die Priorität ist THREAD_PRIORITY_NORMAL. Mit THREAD_PRIORITY_HIGHEST wird hier für Thread A ein höherer, mit THREAD_PRIORITY_LOWEST wird für Thread B ein niedrigerer Wert eingestellt (in der WIN32-API-Dokumentation sind noch eine Reihe von weiteren Zwischenprioritäten angeben, ausserdem lässt sich der absolute Wert der Defaultprioritätsstufe verändern). Durch die höhere Prioritätsstufe läuft Thread A wesentlich öfter als B und der Zählerstand A ändert sich damit viel schneller als der von B:

Ausgabe Thread2.cpp

```
----- Thread A gestartet -----
----- Hauptprogramm -----
Thread A:         0h   Thread B:         0h
----- Thread B gestartet -----
Thread A:   2846689h   Thread B:         0h
Thread A:   5257CB6h   Thread B:         0h
Thread A:   F59FB0Fh   Thread B:    1B2181h
Thread A: 11E6B1A4h   Thread B:    7549E1h
Thread A: 1457E100h   Thread B:    7549E1h
Thread A: 1F35F65Fh   Thread B:    8BB862h
Thread A: 21B4BD01h   Thread B:    8BB862h
Thread A: 2435A109h   Thread B:    E5AECCh
Thread A: 269111BAh   Thread B:    E5AECCh
Thread A: 28F5C5D1h   Thread B:    E5AECCh
Thread A: 2B925BE2h   Thread B:    E5AECCh
. . .
```

Wären die Prioritäten rein statisch, würde Thread B niemals aktiviert, da Thread A die höhere Priorität hat und sich selbst niemals durch einen Scheduler-Aufruf deaktiviert. Erst durch die dynamische Prioritätsanhebung für Threads, die längere Zeit nicht gelaufen sind, kommt auch Thread B gelegentlich zum Zug.

Der Scheduler arbeitet ausschliesslich auf der Ebene von Threads und nicht auf der Ebene von Prozessen. Ein Programm (Prozess), das viele Threads verwendet, bekommt daher durch das Round Robin Verfahren ‚automatisch' mehr Rechenzeit als ein Programm, das nur wenige Threads verwendet (sofern alle Threads, wie bei den meisten Anwendungen üblich, mit der Default-Priorität arbeiten).

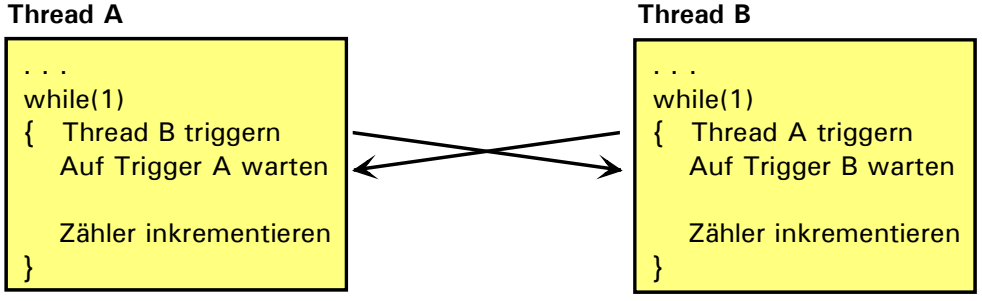
Anwenderprogramme sollten übrigens, wenn nicht absolut notwendig, die Prioritätsstufe auf dem Defaultwert belassen, weil sonst die subjektive Reaktionsgeschwindigkeit des Rechners aus Sicht des Benutzers leidet. Wenn Sie das Beispielprogramm Thread2.cpp laufen lassen, werden Sie selbst auf Rechnern mit schneller CPU feststellen, dass die Fenster der anderen Programme deutlich „träger" reagieren, z.B. wenn Sie ein Fenster auf dem Bildschirm verschieben.

3.3 Thread-Synchronisation

Normalerweise fasst man in einem Thread Programmfunktionen zusammen, die weitgehend unabhängig von den Funktionen in anderen Threads ablaufen können. In vielen Fällen ist es aber notwendig, den Ablauf der Threads zu synchronisieren:

a) „Triggern" eines Threads durch einen anderen Thread

Im Beispiel Thread1.cpp wurden die Zähler in den beiden Threads nicht exakt gleichmässig inkrementiert, weil der Scheduler den beiden Threads nicht exakt dieselbe Rechenzeit zugeteilt hat. Wenn man will, dass die beiden Zähler streng synchron zählen, können sich die beiden Threads mit Hilfe von Events (Unix-Bezeichnung: Signal) gegenseitig triggern:



Beispiel:

Thread3.cpp

```
. . .
HANDLE hEventA,  hEventB;
void threadA(int& i)
{   printf("----- Thread A gestartet -----\n");
    ...
    while(1)
    {   i++;
        SetEvent(hEventB);                      // Event B triggern      3
        WaitForSingleObject(hEventA, INFINITE);// ... und auf Event A warten
    }                                                                     2
}
void threadB(int& i)
{   printf("----- Thread A gestartet -----\n");
    ...
    while(1)
    {   SetEvent(hEventA);                      // Event A triggern
        WaitForSingleObject(hEventB, INFINITE);// ... und auf Event B warten
// alternativ: SignalObjectAndWait(hEventA, hEventB, INFINITE, FALSE); 4
        i++;
    }
}
void main(void)
{   hEventA = CreateEvent(NULL, FALSE, FALSE, NULL);                      1
    hEventB = CreateEvent(NULL, FALSE, FALSE, NULL);  // Events erzeugen

    hThreadA = (HANDLE) _beginthread((void (*)(void *)) threadA, ...);
    hThreadB = (HANDLE) _beginthread((void (*)(void *)) threadB, ...);

    printf("----- Hauptprogramm -----\n");
    SwitchToThread();

    while(!kbhit())
    {    . . .
    }
}
```

Ausgabe Thread3.cpp

```
----- Hauptprogramm -----
----- Thread A gestartet -----
----- Thread B gestartet -----
Thread A:      348h   Thread B:      348h
Thread A:    1066Eh   Thread B:    1066Eh
Thread A:    20B67h   Thread B:    20B68h
Thread A:    307B9h   Thread B:    307BAh
Thread A:    40D0Ch   Thread B:    40D0Ch
. . .
```

1 `CreateEvent(NULL, manualReset, initialStateSet, name)`

Erzeugt ein Event, das über einen 32bit-Wert (Handle) identifiziert werden kann. Bei `manualReset=TRUE` muss das Event, wenn es getriggert wurde, mit dem Befehl `ResetEvent()` manuell zurückgesetzt werden, ansonsten erfolgt das Rücksetzen automatisch, sobald der „Trigger ausgelöst“ wurde. Mit `initialStateSet=TRUE` kann es beim Erzeugen sofort „getriggert“ werden. `name` (Textstring) des Event, darf `NULL` sein.

2 `WaitForSingleObject(eventHandle, timeout_in_msec)`

Versetzt einen Thread in den Zustand ‚Waiting‘, bis ein Event getriggert wird oder bis die Wartezeit `timeout_in_msec` abgelaufen ist (`INFINITE` für unendlich lange Wartezeit). Gibt `WAIT_OBJECT_0` zurück, falls das Event getriggert wurde, und `WAIT_TIMEOUT`, falls die Wartezeit ohne Trigger abgelaufen ist. Mit der Funktion `WaitForMultipleObjects()` kann auch auf mehrere Events gleichzeitig gewartet werden.

3 `SetEvent(eventHandle)`

triggert ein Event. Ein ‚ManualReset‘-Event muss anschliessend, in der Regel in dem Thread, der auf das Event gewartet hat, mit `ResetEvent()` zurückgesetzt werden.

4 `SignalObjectAndWait(hEventS, hEventW, timeout_in_msec, FALSE)`

fasst die häufig benötigte Sequenz `SetEvent(hEventS)` und `WaitForSingleObject(hEventW,...)` zusammen. Dabei ist `hEventS` das Event, das getriggert wird, `hEventW` das Event, auf das anschliessend gewartet wird.

b) **Zeitgesteuertes Triggern von Events bzw. Aufrufen von Threads**

Häufig ist das Ausführen einer Funktion zu einem definierten Zeitpunkt, nach einer definierten zeitlichen Verzögerung oder in periodischen Zeitintervallen notwendig:

1 `CreateWaitableTimer(NULL, manualReset, name)`

2 `SetWaitableTimer(hTimer, &dueTime, period, NULL, NULL, FALSE)`

Erzeugt und startet ein periodisch getriggertes Event mit der Periodendauer `period` (in Millisekunden, bei 0 einmaliges Triggern). Der Zeitpunkt des erstmaligen Triggerns kann über `dueTime` absolut (pos.Wert) oder relativ (negat. Wert) zum aktuellen Zeitpunkt angegeben werden.

Beispiel:

Thread4.cpp

```
. . .
HANDLE hTimerA,  hTimerB;

void threadA(int& i)                // Thread A wird alle 400 ms aufgerufen
{   printf("----- Thread A gestartet -----\n");
    while(1)
    {   i++;
        WaitForSingleObject(hTimerA, INFINITE);      // Auf Timer A warten
    }
}
void threadB(int& i)                // Thread B wird alle 200 ms aufgerufen
{   printf("----- Thread B gestartet -----\n");
    while(1)
    {    i++;
         WaitForSingleObject(hTimerB, INFINITE);     // Auf Timer B warten
    }
}
void main(void)
{   LARGE_INTEGER dueTime = { -10000, 0 };

    hTimerA  = CreateWaitableTimer(NULL, FALSE, NULL);                  1
    hTimerB  = CreateWaitableTimer(NULL, FALSE, NULL); // Timer erzeugen

    hThreadA = (HANDLE) _beginthread((void (*)(void *)) threadA, ...);
    hThreadB = (HANDLE) _beginthread((void (*)(void *)) threadB, ...);

    printf("----- Hauptprogramm -----\n");
    SwitchToThread();

    SetWaitableTimer(hTimerA, &dueTime, 400, NULL, NULL, FALSE);        2
    SetWaitableTimer(hTimerB, &dueTime, 200, NULL, NULL, FALSE); // u.starten

    while(!kbhit())
    {    . . .
    }
}
```

Ausgabe Thread4.cpp

```
----- Hauptprogramm -----
----- Thread A gestartet -----
----- Thread B gestartet -----
Thread A:         1   Thread B:         1
Thread A:         3   Thread B:         5
Thread A:         5   Thread B:        10
Thread A:         8   Thread B:        15
Thread A:        10   Thread B:        20
. . .
```

c) Gegenseitiger Ausschluss (Mutual Exclusion): Zugriff auf eine gemeinsam genutzte Ressource aus verschiedenen Threads

Wenn mehrere Threads eine gemeinsame Ressource verwenden, z.B. auf den Bildschirm schreiben, ist eine Synchronisation zwingend notwendig. Beispiel:

Gewünschte Bildschirmausgabe:

Ausgabe Thread5b.cpp
(mit Synchronisation)

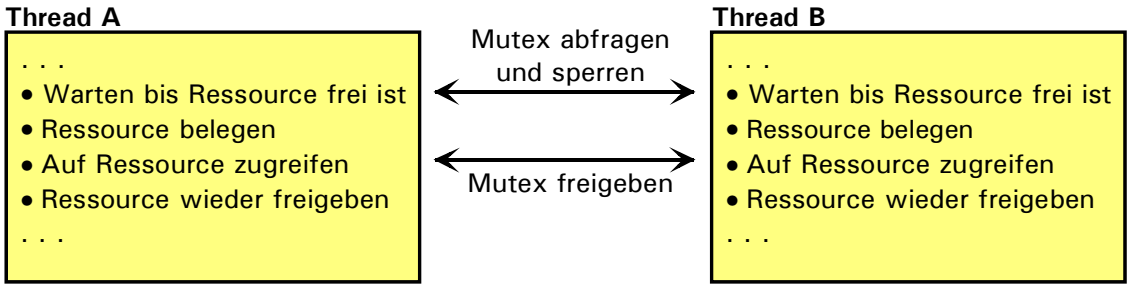
```
----- Hauptprogramm -----
----- Thread A gestartet -----
----- Thread B gestartet -----
 0  1  2  3  4  5  6  7  8  9 10 11 12
 0  1  2  3  4  5  6  7  8  9 10 11 12
 a  b  c  d  e  f  g  h  i  j  k  l  m
 a  b  c  d  e  f  g  h  i  j  k  l  m
 0  1  2  3  4  5  6  7  8  9 10 11 12
 0  1  2  3  4  5  6  7  8  9 10 11 12
 a  b  c  d  e  f  g  h  i  j  k  l  m
 a  b  c  d  e  f  g  h  i  j  k  l  m
 . . .
```

Tatsächliche Bildschirmausgabe ohne Synchronisation:
Weil das Betriebssystem zu beliebigen Zeiten zwischen den Threads umschaltet, gerät die Bildschirmausgabe durcheinander.

Ausgabe Thread5a.cpp
(ohne Synchronisation)

```
----- Hauptprogramm -----
----- Thread A gestartet -----
----- Thread B gestartet -----
 0  1  2  3  4  5  6  7  8  9 10 11 12
 0  1  2  3  4  5  6  7  8  9 10 11 12
 0  1  a  b  c  d  e  f  g  h  i  j  k  l  m
 a  b  c  d  e  f  g  h  i  j  k  l  m
 a  b  c  d  e  f  g  2  3  4  5  6  7  8  9 10 11 12
 0  1  2  3  4  5  6  7  8  9 10 11 12
 . . .
```

Unter Windows kann die Synchronisation in solchen Fällen durch einen **Mutex** (Mutual Exclusion Object) erfolgen, dies ist eine vereinfachte Ausführung der bekannten **Semaphoren** (sh. Vorlesung Betriebssysteme).



Beispiel: ***Thread5x.cpp***

```
. . .              // Programm ähnlich wie Thread1.cpp mit Ausnahme der
                   // dargestellten Unterschiede

HANDLE hMutex;

void threadA(int& i)
{   . . .
    while(1)
    {   WaitForSingleObject(hMutex, INFINITE); // Warten, bis frei      2a
        for (int i=0; i<13; i++)
            printf("%2u ", i);
        printf("\n");
        ReleaseMutex(hMutex);                  // Wieder freigeben      3a
    }
}
void threadB(int& i)
{   . . .
    while(1)
    {   WaitForSingleObject(hMutex, INFINITE); // Warten, bis frei      2b
        for (char c='a'; c<'n'; c++)
            printf("%2c ", c);
        printf("\n");
        ReleaseMutex(hMutex);                  // Wieder freigeben      3b
    }
}
void main(void)
{   hMutex = CreateMutex(NULL, NULL, NULL);    // Mutex erzeugen        1
  . . .
}
```

1 `CreateMutex(NULL, NULL, name)`

Erzeugt ein Mutex-Objekt. Das Mutex-Objekt wird durch einen 32bit-Wert (Handle) identifiziert. `name` (Textstring) des Event, darf `NULL` sein.

2 `WaitForSingleObject(mutexHandle, timeout_in_msec)`

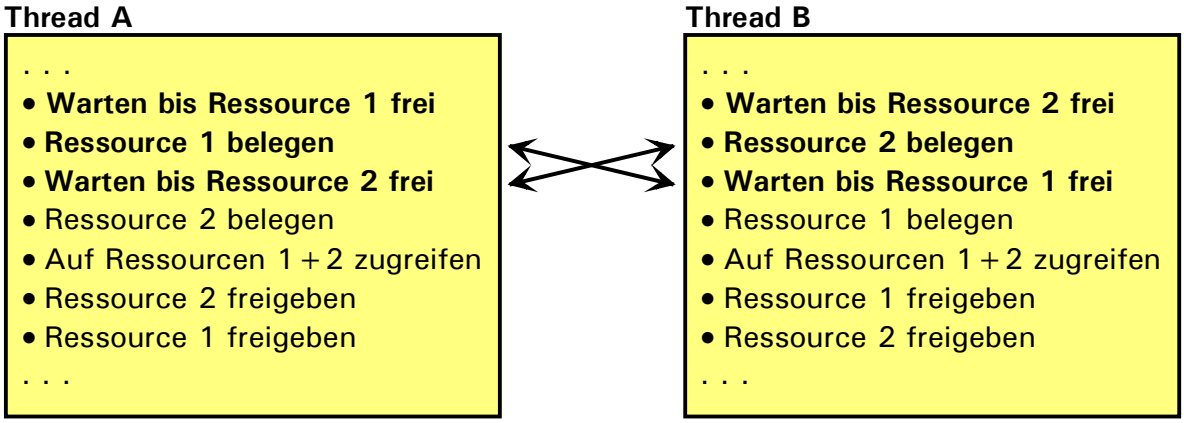
Fragt ein Mutex-Objekt ab und wartet, falls es gerade nicht frei ist. Die maximale Wartezeit kann durch `timeout_in_msec` vorgeben werden, bei `INFINITE` wird beliebig lange gewartet. Wenn der Mutex frei ist, wird der Mutex als belegt gekennzeichnet und der Thread läuft weiter.

3 `ReleaseMutex(mutexHandle)`

Sobald die Ressource nicht mehr verwendet wird, muss sie freigegeben werden, damit andere Threads auch auf die Ressource zugreifen können.

Der Warte-Freigabe-Mechanismus muss von jedem Thread an all denjenigen Stellen eingesetzt werden, die die Ressource verwenden. Wenn mehrere Ressourcen von mehreren Threads gemeinsam verwendet werden, sollte für jede Ressource ein eigenes Mutex-Objekt definiert werden.

Wie bei Semaphoren muss sorgfältig darauf geachtet werden, dass keine Blockadesituation (**Deadlock**) auftritt, bei der zwei Threads wechselseitig auf Ressourcen warten, die jeweils vom anderen Thread blockiert werden. Beispiel für einen Deadlock:



Abhilfe hier: Im Thread B die Ressourcen 1 und 2 in derselben Reihenfolge anfordern wie im Thread A.

Weitere Informationen zum Multithreading finden Sie in der Visual C++ - Dokumentation – Abschnitt 'Platform SDK'.

4. Exception Handling unter Windows NT/2000/XP

4.1 Grundprinzip

Bei bestimmten Fehlern in Anwendungsprogrammen löst die CPU einen Ausnahmefehler-(Exception)-Interrupt aus. Die zugehörige ISR des Betriebssystems bricht dann üblicherweise das Anwenderprogramm ab. Bsp.:

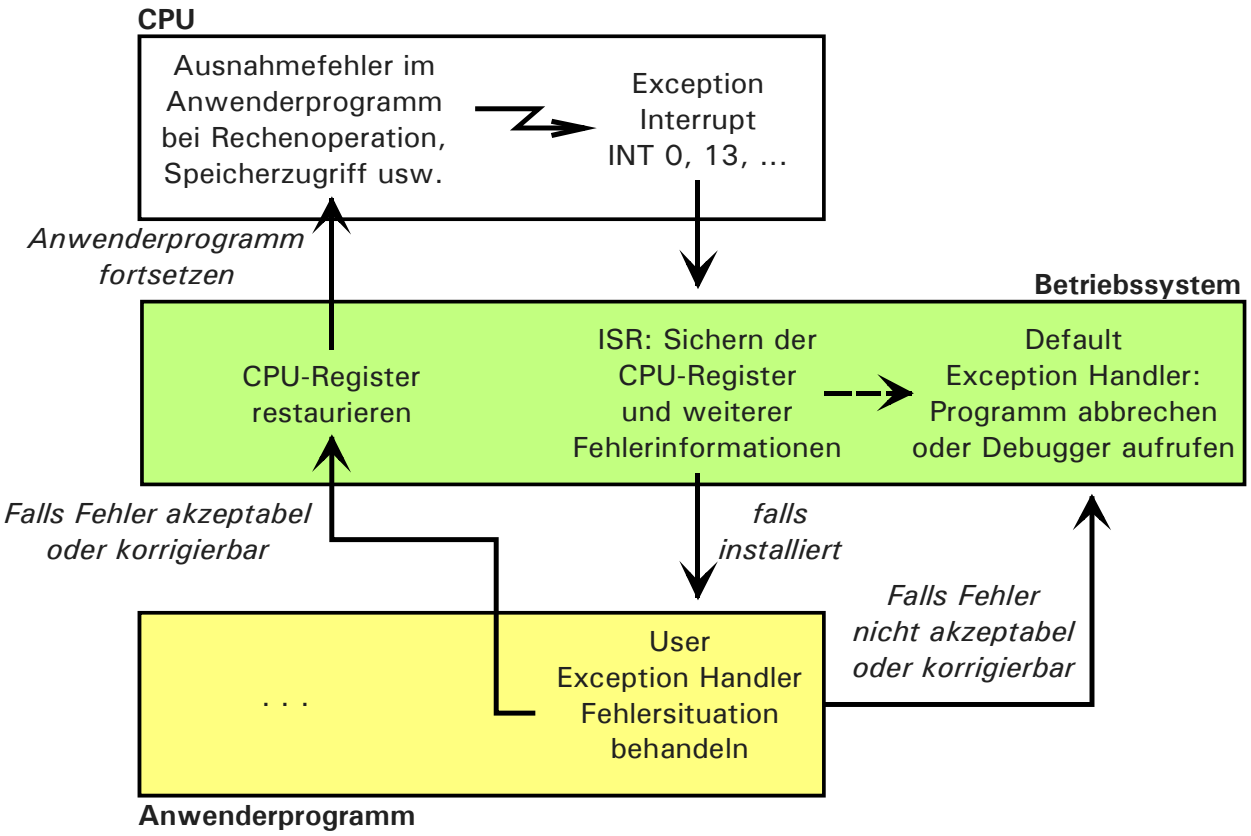
Except1.cpp

```
#include <windows.h>
#include <stdio.h>

void main(void)
{     int ergebnis, dividend, divisor;
      . . .
      printf("Dividend =    ");     scanf("%d", &dividend);
      printf("Divisor  =    ");     scanf("%d", &divisor);

      ergebnis = dividend / divisor;  ← Divisor 0 führt zu Abbruch!!!
      printf("Quotient =    %d / %d = %d\n", dividend, divisor, ergebnis);
}
```

Das folgende Bild zeigt das Zusammenwirken von CPU und Betriebssystem:



Natürlich lassen sich viele derartige Fehler durch Abfragen im Programmcode vermeiden, z.B.:

```
if (divisor != 0)       ergebnis = dividend / divisor;
else                    printf("Division durch 0 nicht zulässig");
```

In manchen Situationen, z.B. in komplexem Programmcode, ist es aber günstiger, eine eigene Behandlungsroutine für derartige Ausnahmefehler zu realisieren. Moderne Programmiersprachen wie C++ oder Java unterstützen dies durch entsprechende Sprachkonstrukte. Der Ausnahmefehler-Behandlung in den Programmiersprachen liegt der **Structured Exception Handling** (SEH) - Mechanismus des Betriebssystems zugrunde:

Except2.cpp

```
#include <windows.h>
#include <stdio.h>

void main(void)
{     int ergebnis, dividend, divisor;
      . . .
      printf("Dividend =    ");     scanf("%d", &dividend);
      printf("Divisor  =    ");     scanf("%d", &divisor);
      __try                                                          1
      {   ergebnis = dividend / divisor;← Befehl, der den Fehler auslösen kann
          printf("Quotient =    %d / %d = %d\n", dividend, divisor, ergebnis);
      } __except( EXCEPTION_EXECUTE_HANDLER )                        2
      {   printf("Divisionsfehler\n");  ← Ausnahmefehler-Behandlung
      }
}
```

Beispielablauf:

Dividend = 456789 Divisor = 56 Quotient = 456789 / 56 = 8156	*Divisor != 0:* *Code im __try-Block wird vollständig ausgeführt.*
Dividend = 3355 Divisor = 0 Divisionsfehler	*Divisor == 0:* *Bei Division im __try-Block tritt Ausnahmefehler auf. Die folgenden Befehle im __try-Block werden nicht mehr ausgeführt, stattdessen der Code im __except-Block.*

4.2 Implementierungsdetails

1 __try { ... }

Der Programmcode, von dem man befürchtet, dass er einen Ausnahmefehler auslöst, wird in einen __try { ... } Block eingeschlossen.

2 __except (...) { ... }

Die Behandlungsroutine für den Ausnahmefehler kann entweder unmittelbar im { ... }-Block stehen. Dieser Block wird ausgeführt, wenn der Ausdruck in der runden (...) den Wert EXCEPTION_EXECUTE_HANDLER hat.

Wenn man die Ausnahmebehandlung in eine eigene C-Funktion 3 verlegt, wird die Fehlerbehandlung flexibler:

Except3.cpp

```
#include <windows.h>
#include <stdio.h>

int ergebnis, dividend, divisor;

DWORD exHandler(DWORD exceptionCode)                                   3
{   if (exceptionCode != EXCEPTION_INT_DIVIDE_BY_ZERO)
        return EXCEPTION_CONTINUE_SEARCH;                              6

    printf("Wollen Sie einen neuen Divisor eingeben: J/N ? ");
    if (toupper(getch()) == 'J')
    {   printf("Divisor  =    ");  scanf("%d", &divisor);
        return EXCEPTION_CONTINUE_EXECUTION;                           5
    } else
        return EXCEPTION_EXECUTE_HANDLER;                              4
}

void main(void)
{     . . .
      __try                                                            1
      {  ergebnis = dividend / divisor;
          printf("Quotient =    %d / %d = %d\n", dividend, divisor, ergebnis);
      } __except( exHandler( GetExceptionCode() ) )          2         3
      {   printf("Divisionsfehler\n"); . . .
      }
}
```


Die Ausnahmefehler-Behandlungsroutine wird als

3 `DWORD exHandler(DWORD exceptionCode, ...)`

deklariert. Der Parameter exceptionCode wird von der Funktion GetExceptionCode() bereitgestellt und enthält Informationen über die Ursache des Ausnahmefehlers. Definiert sind u.a. folgende Werte:

- EXCEPTION_ACCESS_VIOLATION
  Zugriffsfehler beim Lesen oder Schreiben des Speichers (unzulässige Adresse, schreibgeschützter Speicherbereich o.ä.)
- EXCEPTION_PRIV_INSTRUCTION
  Versuch, einen privilegierten Maschinenbefehl, z.B. CLI, IN oder OUT in einem Anwendungsprogramm (User Mode) auszuführen.
- EXCEPTION_INT_DIVIDE_BY_ZERO
  Division mit ganzzahligen Operanden durch 0
- . . . Weitere Werte werden in der WIN32-API Dokumentation beschrieben.

Je nach Fehler kann die Behandlungsroutine . . .

4 EXCEPTION_EXECUTE_HANDLER

den weiteren Programmcode im __try-Block nicht mehr ausführen, dazu gibt die Behandlungsroutine EXCEPTION_EXECUTE_HANDLER zurück. Dann wird stattdessen der Programmcode im { ... } Bereich des __except-Blocks ausgeführt.

5 EXCEPTION_CONTINUE_EXECUTION

den Fehler beseitigen und danach den ursprünglich fehlerhaften Befehl wiederholen und im __try -Block weitermachen. Dazu muss die Routine EXCEPTION_CONTINUE_EXECUTION zurückgeben. Wenn zur Fehlerbeseitigung auf lokale Parameter des __try -Blocks zugegriffen werden muss, können der Behandlungsroutine zusätzlich Pointer auf diese lokalen Variablen übergeben werden..

6 EXCEPTION_CONTINUE_SEARCH

eine Behandlungsroutine eines übergeordneten Blocks aufrufen, indem sie den Wert EXCEPTION_CONTINUE_SEARCH zurückgibt. __try-__except-Blöcke können geschachtelt werden. Wenn im Anwenderprogramm keine übergeordnete Behandlungsroutine definiert ist, wird die Behandlungsroutine des Betriebssystems aufgerufen.

Ersetzt man den Aufruf der Behandlungsroutine durch

```
__except( exHandler( GetExceptionInformation() ) ) { }
```

und definiert man die Behandlungsroutine als

```
DWORD exHandler (EXCEPTION_POINTERS* pEx, ...)
```

so erhält man noch detailliertere Informationen über den Ausnahmefehler. Der Parameter pEx zeigt auf eine Struktur von Pointern, die ihrerseits auf zwei weitere Strukturen zeigt (siehe Beispielprogramm Except4.cpp):

- Über das Element pEx–>ExceptionRecord–>ExceptionCode kann wieder die Ursache des Ausnahmefehlers ermittelt werden.
- pEx–>ExceptionRecord–>ExceptionAddress zeigt auf die Offset-Adresse des Befehls, der den Ausnahmefehler ausgelöst hat.
- Der Wert der CPU-Register zum Zeitpunkt des Ausnahmefehlers kann direkt über die Context-Struktur (siehe auch Abschnitt 3) ausgelesen werden, z.B. pEx–>ContextRecord–>Cseg für CS, pEx–>ContextRecord–>Eip für EIP, pEx–>ContextRecord–>Eax für EAX, usw.. Modifiziert man den Inhalt der Context-Struktur und gibt anschliessend EXCEPTION_CONTINUE_EXECUTION zurück, so wird der fehlerhafte Befehl mit den neuen Registerinhalten nochmals ausgeführt.

Except4.cpp

```
DWORD exHandler(EXCEPTION_POINTERS *pEx)
{       printf("Exception Code=%8Xh \n",    pEx–>ExceptionRecord–>ExceptionCode);
        printf("Address= %8Xh\n",           pEx–>ExceptionRecord–>ExceptionAddress);
        printf("Context Record:  EIP = %Xh   EAX = %Xh\n",
                                            pEx–>ContextRecord–>Eip, pEx–>ContextRecord–>Eax);

        if (pEx–>ExceptionRecord–>ExceptionCode != EXCEPTION_INT_DIVIDE_BY_ZERO)
                return EXCEPTION_CONTINUE_SEARCH;

        printf("Wollen Sie einen neuen Divisor eingeben: J/N ?    ");
        if (toupper(getch()) == 'J')
        {       printf("\nDivisor  =    ");
                scanf("%d", &divisor);
                return EXCEPTION_CONTINUE_EXECUTION;
        } else
                return EXCEPTION_EXECUTE_HANDLER;
}

void main(void)
{       . . .
        __try
        {   . . .
        } __except( exHandler( GetExceptionInformation() ) )
        { . . .
        }
}
```

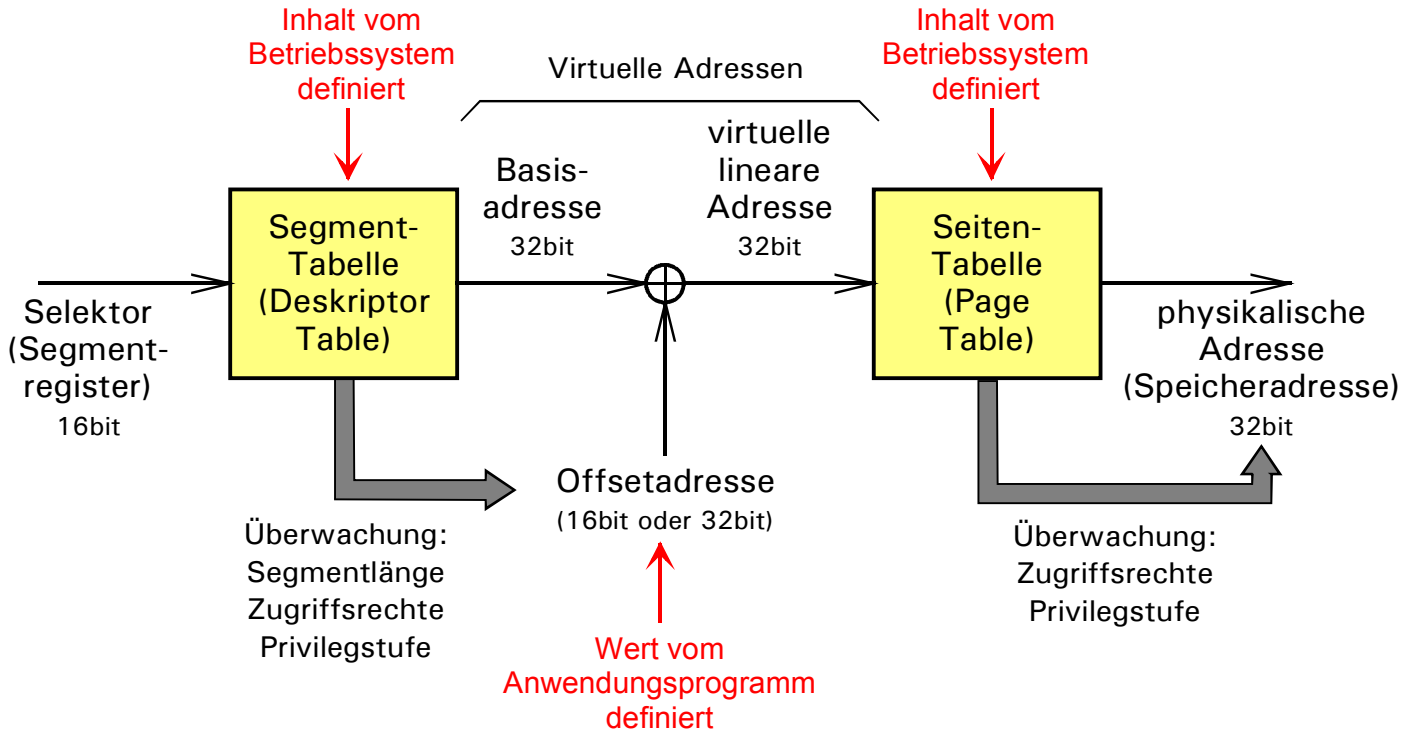
Weitere Informationen zum Structured Exception Handling finden Sie in der Visual C++ - Dokumentation Abschnitt 'Platform SDK' - 'Windows Base Services' - 'Structured Exception Handling'.

5. Verwaltungsstrukturen des 80x86-Protected Mode – Nutzung unter Windows und Linux

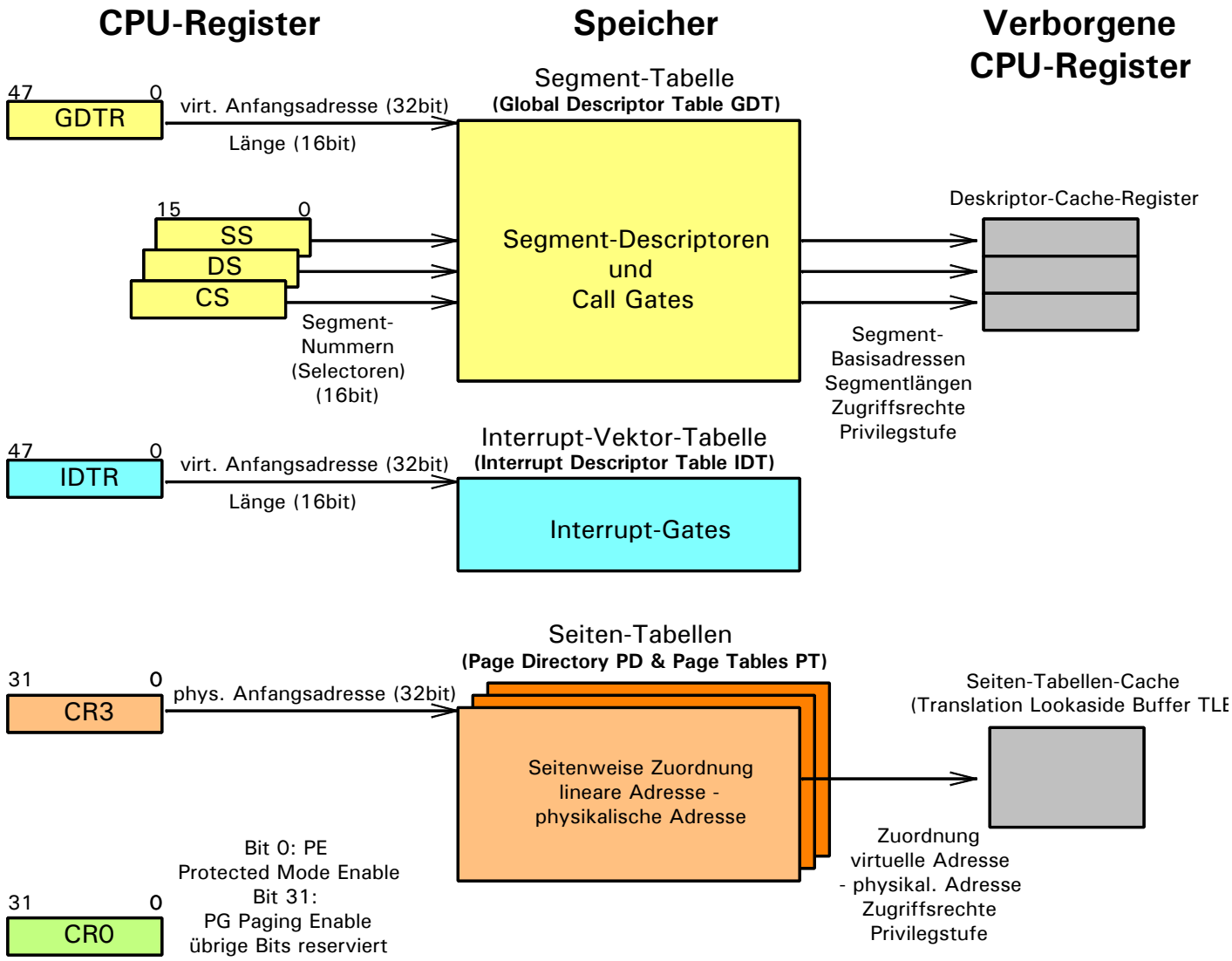
5.1 Überblick

- Nach dem Reset starten 80x86-CPUs im Real Mode (16bit Offsetadressen, keine Schutzmechanismen). Während des Bootvorgangs des Betriebssystems werden die Verwaltungsstrukturen des Protected Mode initialisiert und die CPU in den Protected Mode umgeschaltet:

Grundschema



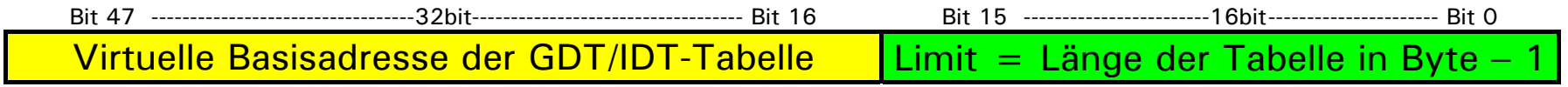
80x86-Register und Speicher-Tabellen für den Protected Mode



- **Einschalten des Protected Mode und des Paging**
  Der Betriebsmodus der CPU wird über das Steuerregister CR0 eingestellt. Im Kernel-Modus ist CR0 ist über normale MOV-Befehle zugänglich. Bevor das Betriebssystem die CPU in den Protected Mode um- und das Paging einschaltet, muss es die Tabellen initialisieren.

- Segment-Tabelle, Seiten-Tabelle und Interruptvektor-Tabelle stehen im Speicher an beliebigen Stellen. Damit die CPU die Verwaltungsstrukturen finden kann, müssen die Anfangsadressen der Tabellen in die **Register GDTR, IDTR und CR3** geladen werden.

  GDTR (Global Descriptor Table Register), IDTR (Interrupt Descriptor Table Register)



  Lesen von GDTR und IDTR mit dem Befehl SGDT bzw. SIDT (Store ... Table Register):
  Im Kernel und im User Mode.

  Schreiben von GDTR und IDTR mit dem Befehl LGDT bzw. LIDT (Load ... Table Register):
  Nur im Kernel Mode

Beispielprogramm zum Anzeigen von GDTR und IDTR: *TableReg.cpp*

```
#pragma pack(1)
struct
{   WORD limit;
    DWORD baseAddress;
} xGdtr, xIdtr;
#pragma pack()
```

```
void main(void)
{   _asm SGDT FWORD PTR xGdtr
    _asm SIDT FWORD PTR xIdtr
     printf("GDT: Basisadresse=%Xh  Segmentlimit=%Xh\n",
                                                xGdtr.baseAddress, xGdtr.limit);
}
```

CR3 (Control Register 3)

Bit 31 ----------32bit---------- Bit 0

Physikalische Basisadresse der Seiten-Tabelle

Lesen und Schreiben von CR0...CR3 mit MOV-Befehlen: Nur im Kernel Mode

CR3 ist zwar ein vollständiges 32bit-Register, da Seiten-Tabellen aber immer auf Adressen beginnen müssen, die ein ganzzahliges Vielfaches von 4KB sind, müssen die Bits 11...0 in CR3 immer auf 0 gesetzt sein.

- **CR3 und die Seiten-Tabellen** enthalten ausschließlich **physikalische Adressen**. An **allen anderen Stellen** (GDTR, ..., Segment-Tabellen und in Programmen) werden ausschließlich **virtuelle Adressen** verwendet.
- Die **Segment-, Seiten- und Interrupt-Tabellen** befinden sich im Kernel-Mode-Teil des virtuellen Adressraums, sind also **von Anwendungsprogrammen aus weder les- noch schreibbar**. Bei Windows NT/2000/XP:

  GDT bei 8003 6000h IDT bei 8003 6400h Page Directory bei C000 0000h
- Damit die bei jedem Speicherzugriff im Protected Mode erforderlichen zusätzlichen Zugriffe auf die Segment- und Seiten-Tabellen die Befehle nicht verlangsamen, speichert die CPU automatisch die letzten aus den Tabellen ausgelesenen Tabelleneinträge in verborgenen Registern der CPU zwischen (**Descriptor Cache, Translation Lookaside Buffer**). Dadurch ist nur beim ersten Zugriff auf ein Segment bzw. eine Seite ein Auslesen der Tabellen notwendig. Solange nicht auf andere Segmente bzw. Seiten zugegriffen wird, werden die Tabelleneinträge dann aus den schnellen CPU-internen verborgenen Registern gelesen.

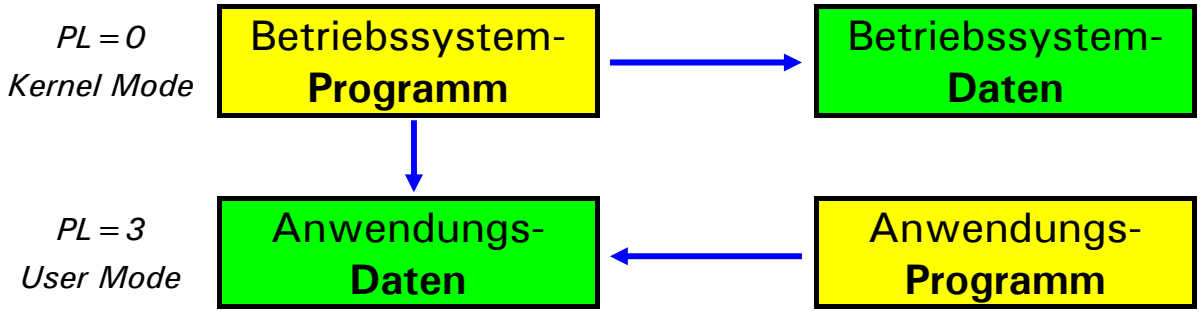
Privilegstufen (Privilege Level PL)

Windows und Linux verwenden zwei der vier möglichen 80x86-Privilegstufen:

PL = 0	Höchste Stufe	**Kernel Mode**	für das Betriebssystem verwendet
PL = 3	Niedrigste Stufe	**User Mode**	für Anwendungsprogramme verwendet

- Jedem Speicherbereich, d.h. **jedem Segment bzw. jeder Seite** wird über die Segment- bzw. die Seiten-Tabelle genau **eine** der beiden **Privilegstufe**n **zugeordnet**.
- Die **Privilegstufe des gerade laufenden Programms** wird durch die Privilegstufe seines Code-Segments definiert (Code bzw. **Current Privilege Level CPL**).
- Bei jedem Speicherzugriff prüft die CPU automatisch zunächst, ob die **Privilegstufe des Speichersegments, auf das zugegriffen werden soll** (**Descriptor Privilege Level DPL**), zulässig ist. Anschließend erfolgt eine weitere Überprüfung für die Privilegstufe der Speicherseite.

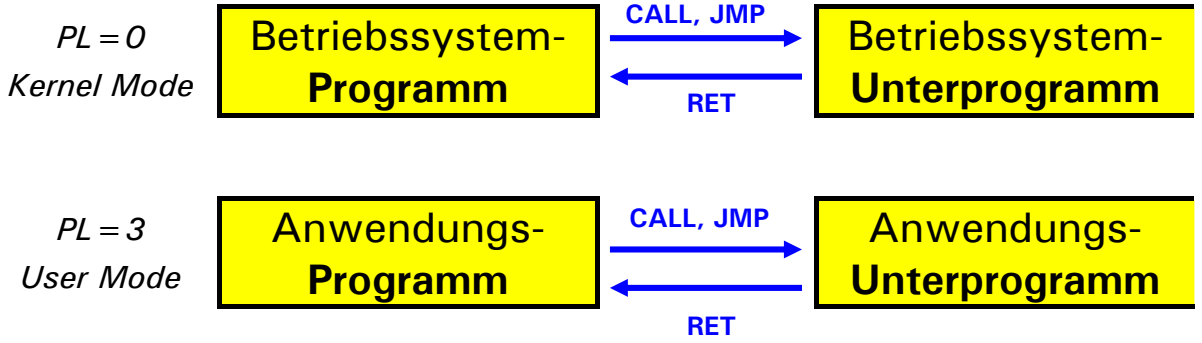
  Bei einer Verletzung der Privilegstufen erzeugt die CPU einen Exception Interrupt INT 13 („General Protection Fault GPF, allgemeiner Ausnahmefehler).
- **Beim Wechsel der Privilegstufe schaltet** die **CPU automatisch** vom **Stack** des Anwendungsprogramms auf den Stack des Betriebssystems **um** und umgekehrt. Dadurch wird das Betriebssystem vor Stackfehlern des Anwendungsprogramms geschützt.
- Für **Software-Interrupts** gelten dieselben **Regeln wie** für **Unterprogramm-Aufrufe**. Da Windows ISRs aber nur im Kernel Mode installiert, können 32bit-(Protected Mode)-Programme im allgemeinen keine Interrupt-Aufrufe verwenden.
- **Hardware-Interrupts und Ausnahmefehler**-Interrupts werden **unabhängig von CPL** ausgeführt.



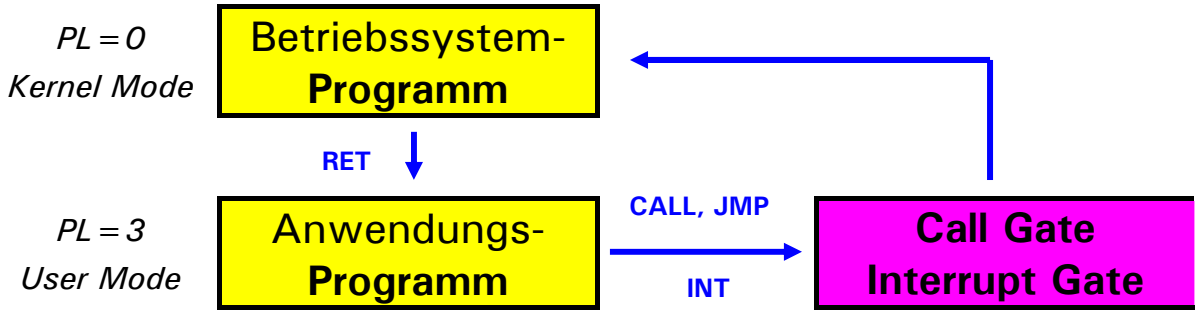
Zugriff auf Daten

Durch diesen Mechanismus schützt das Betriebssystem auch die Segment- und Seiten-Tabellen, die als Betriebssystemdaten ja ebenfalls im Speicher stehen, gegen direkte Änderungen aus Anwendungsprogrammen.

Regeln für Programmsprünge und Unterprogrammaufrufe



Programmsprünge und Unterprogrammaufrufe ohne Wechsel der Privilegstufe



Programmsprünge und Unterprogrammaufrufe mit Wechsel der Privilegstufe

Ein **Call Gate** ist ein Eintrag in der Segment-Tabelle, der die eigentliche Segment-Offset-Adresse des aufzurufenden Programms enthält. Das Call Gate muss **dieselbe Privilegstufe** haben **wie** das **Anwendungsprogramm**. Das aufzurufende Segment selbst darf eine höhere Privilegstufe haben:

Aufrufendes Programm mit CPL

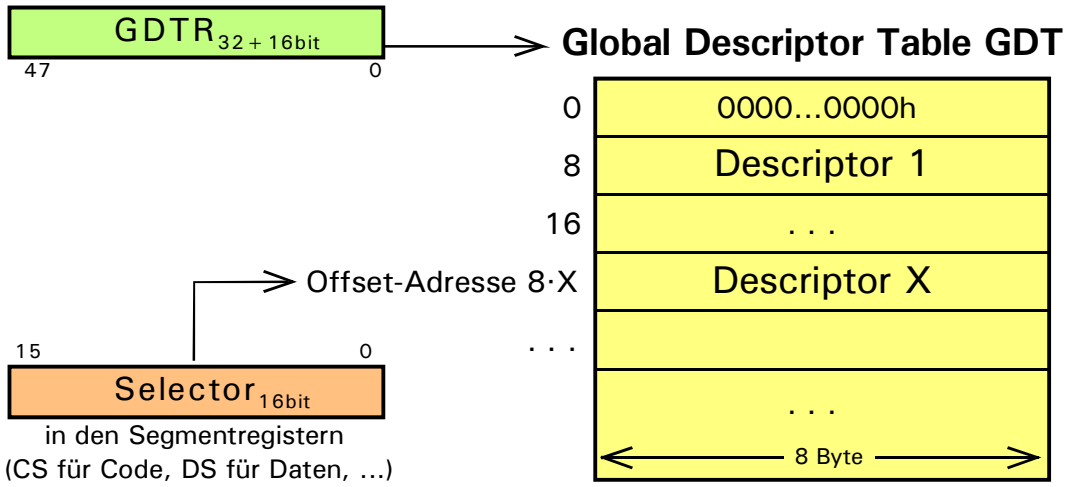
→ Call Gate mit $DPL_{\text{Call Gate}} \geq CPL_{\text{Aufrufendes Programm}}$

→ Aufgerufenes Programm mit $DPL_{\text{Aufgerufenes Programm}} \leq CPL_{\text{Aufrufendes Programm}}$

Da das Call Gate in der Segment-Tabelle steht, die vom Anwendungsprogramm aus nicht verändert werden kann, kann das Betriebssystem kontrollieren, welche Betriebssystemfunktionen von Anwendungsprogrammen aus aufgerufen werden können.

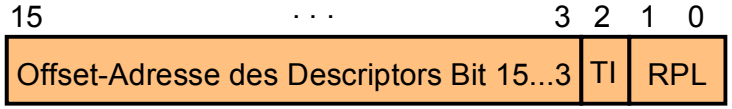
Erstaunlicherweise kann das Betriebssystem selbst übrigens kein Anwenderprogramm direkt oder über ein Call Gate direkt aufrufen, da Aufrufe von User Mode-Programmen aus dem Kernel Mode unzulässig sind. Um ein Anwendungsprogramm zu starten, "simuliert" das Betriebssystem daher einen Rücksprung aus einem Unterprogramm, d.h. es legt die Startadresse des Anwenderprogramms als "Rücksprungadresse" auf den Stack und führt einen RET-Befehl aus.

5.2 Aufbau der Segment-Tabellen: Selectoren und Descriptoren



- Für jedes Segment existiert ein 8 Byte großer Eintrag (**Descriptor**) in der GDT, max. 8192 Einträge.
- Der erste Eintrag muss 0 sein.
- Das Segment-Register der CPU zeigt auf die Offset-Adresse des Eintrags innerhalb der Segment-Tabelle (**Selector**).

Selector



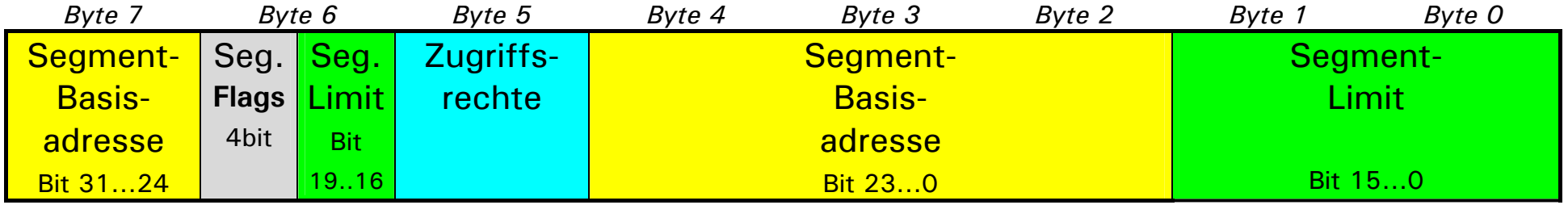
TI Table Indicator = 0: Selector bezieht sich auf die **GDT**
= 1: Selector bezieht sich auf die **LDT**

Außer der globalen sind auch lokale Segment-Tabellen (LDT) möglich (wird für 32bit Programme unter Windows NT/2000/XP aber nicht verwendet, d.h. immer TI = 0).

RPL Requested Privilege Level (im allgemeinen RPL = DPL)
Dient zur dynamischen Veränderung der Privilegstufe. Die Privilegstufe beim Zugriff auf ein Speichersegment wird aus max(RPL,DPL) ermittelt (wird für 32bit Programme unter Windows NT/2000/XP aber nicht verwendet, dort immer RPL = DPL).

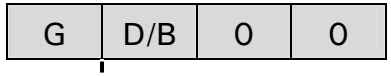
- Da die Descriptoren jeweils 8 Byte lang sind, sind die Bits 2...0 der Offset-Adresse immer 0 und werden nicht gespeichert. Die Bits 2...0 des Selectors enthalten daher andere Informationen (TI, RPL).

Descriptoren für Code-, Stack- und Daten-Segmente



Da der Protected Mode bereits bei der 16bit-CPU 80286 eingeführt und beim Übergang zur 32bit-CPU 80386 aufwärtskompatibel erweitert wurde, ist die Descriptor-Struktur leider zerstückelt. Die Felder haben folgende Bedeutung:

- **Segment-Basisadresse** = Virtuelle 32bit-Anfangsadresse eines Speicher-Segmentes
- **Segment-Limit** = Segmentlänge – 1 in Byte, wenn G = 0, in 4KB, wenn G = 1 (siehe unten)
- **Segment-Flags**

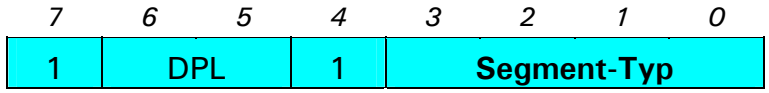


Granularity Bit | **Default-Datengröße D/B**

D/B = 0 16bit-Segment, d.h. Adressierung erfolgt mit 16bit-Offset-Adressen

D/B = 1 32bit-Segment, d.h. Adressierung erfolgt mit 32bit-Offset-Adressen

- **Zugriffsrechte**



Descriptor Privilege Level (Privilegstufe) 0...3

0h Schreibgeschütztes Daten-Segment

2h Les- und schreibbares Daten- oder Stack-Segment

8h Nur ausführbares Code-Segment

Ah Les- und ausführbares Code-Segment

Descriptoren für Interrupt-Gates (in der IDT) **und Call-Gates** (in der GDT)

Byte 7 *Byte 6*	*Byte 5*	*Byte 4*	*Byte 3* *Byte 2*	*Byte 1* *Byte 0*
EIP Bit 31…16	Zugriffs-rechte	Parameter-Anzahl	CS	EIP Bit 15…0

Die Felder haben folgende Bedeutung:

- **CS : EIP** = Code-Segment-Selector und Offset-Anfangsadresse der Interrupt Service Routine bzw. des Unterprogramms, das aufgerufen werden soll

- **Zugriffsrechte**

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	1	DPL		0	ST	1	Gate-Typ	

Descriptor Privilege Level (Privilegstufe) des Gates 0…3

ST = 0 16bitStack
ST = 1 32bitStack

00 Unterprogramm
10 Hardware-Interrupt-Service-Routine (HW-ISR)
11 Software-Interrupt-Service-Routine (SW-ISR)
bei HW-ISR setzt die CPU IF = 0, wenn sie die ISR aufruft,
bei SW-ISR bleibt IF unverändert.

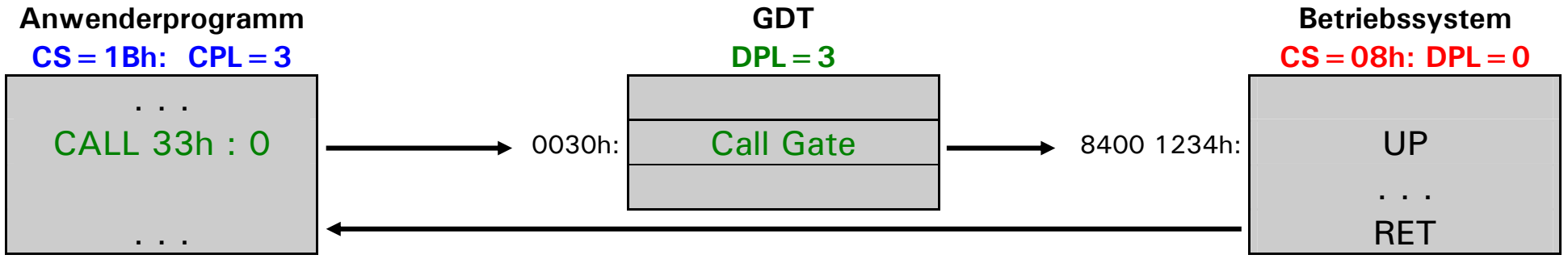
- **Anzahl der Stack-Parameter**

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	0	0	0	Anzahl WORDs (bei ST = 0) bzw. DWORDs (bei ST = 1)				

Beim Wechsel der Privilegstufe wird automatisch auch der Stack umgeschaltet (für jede Privilegstufe gibt es einen eigenen Stack, siehe Abschnitt Task State Segment). Die hier angegebene Anzahl wird dabei vom Stack des aufrufenden Programms auf den Stack der aufgerufenen Routine kopiert. Bei Interrupt-Gates ist die Anzahl in der Regel 0.

Beispiel:

- Code-Segmente für das Betriebssystem CS = 08h (4GB, 32bit, Kernel Mode) **Descriptor**
- Code-Segment für Anwenderprogramme CS = 1Bh (4GB, 32bit, User Mode) **Descriptor**
- Call Gate für den Aufruf der Betriebssystemfunktion bei Offset-Adresse 84001234h im Code-Segment des Betriebssystems mit 3 DWORD-Parametern aus dem Anwendungsprogramm



GDT

	Byte 7	*Byte 6*		*Byte 5*	*Byte 4*	*Byte 3*	*Byte 2*	*Byte 1*	*Byte 0*
0000h	00h	0h	0h	00h	000000h			0000h	
0008h	00h	Ch	Fh	9Ah	000000h			FFFFh	
0010h									
0018h	00h	Ch	Fh	FAh	000000h			FFFFh	
0020h									
0028h									
0030h	8400h			ECh	03h	0008h		1234h	
. . .	. . .								

Segmente unter Windows (und Linux) im Speichermodell FLAT

- Folgende **Segmente** sind **in der Segment-Tabelle** GDT (Global Descriptor Table) definiert:
  - CS = 08h 32bit Code-Segment] für das
  - DS = SS = 10h 32bit Daten- und Stack] Betriebsystem DPL = 0
  - CS = 1Bh 32bit Code-Segment] für alle Anwender-
  - DS = SS = 23h 32bit Daten- und Stack] programme DPL = 3
- Zugriffsrechte für Code-Segmente: 'les- und ausführbar', für die Daten/Stack-Segmente 'les- und schreibbar'.
- Alle Segmente haben die Basisadresse 0, sind 4GB lang und in der globalen Segment-Tabelle (Global Descriptor Table GDT) definiert, **d.h. alle Segmente zeigen auf denselben virtuellen linearen 4GB-Adressraum**. Mit **32bit-Offset-Adressen** kann jedes Programm auf den gesamten Speicher-Adressraum der CPU zugreifen, d.h. **faktisch** gibt es **keine Segmentierung.**

  (Die Einträge in der Segmenttabelle sind trotzdem erforderlich, da 80x86-CPUs nur auf diese Weise Privilegstufen für die Schutzmechanismen definieren können).
- **Betriebssystemaufrufe** erfolgen über das **Interrupt-Gate INT 2Eh** in der IDT.

Die **Segment-Register werden nur umgeladen**, **wenn** ein **Wechsel zwischen User- und Kernel-Modus** erfolgt.

Wenn ältere 16bit-(DOS oder Windows 3.x)-Programme ausgeführt werden, legt das Betriebssystem zusätzliche Segmente, Interrupt- und Call Gates an.

Lokale Descriptor Tabellen (LDT)

Die LDT enthält wie die GDT Descriptoren für Speicher-Segmente oder Call Gates und hat dasselbe Format wie die GDT. LDTs sind normale Speicher-Segmente, die durch einen Descriptor in der GDT beschrieben werden. Lediglich die Felder Zugriffsrechte und Segmentflags unterscheiden sich von einem normalen Speichersegment:

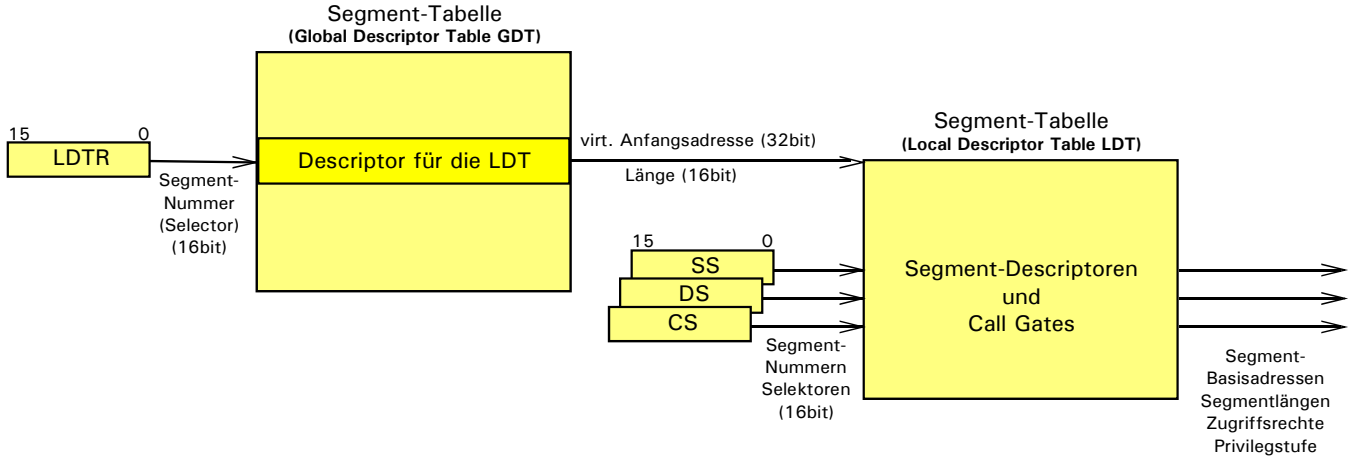
- Zugriffsrechte

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	1	DPL		0	0	0	0	0

- Segmentflags: 0h

LDTs werden bei Windows nur für die veralteten 16bit-Protected-Mode-Programme verwendet. Dabei enthält jedes 16bit-Protected-Mode-Programm eine eigene LDT. Der GDT-Selector der gerade aktiven LDT wird vom Betriebssystem mit dem Befehl LLDT in das 16bit LDTR-Register Register geladen. Er kann mit dem Befehl SLDT ausgelesen werden.

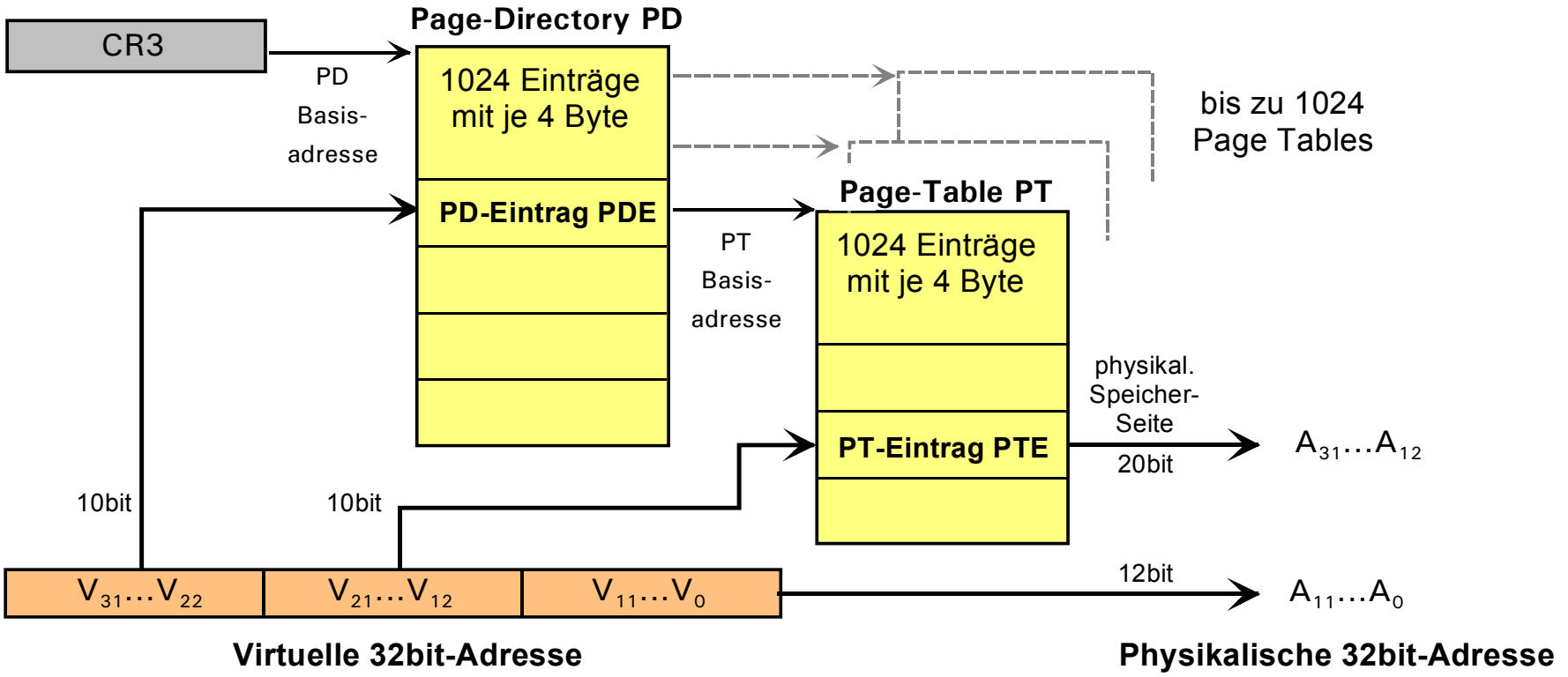
Die CPU erkennt am TI-Bit eines Selectors (TI = 1), dass der Descriptor des zugehörigen Speichersegments nicht in der GDT sondern in der LDT steht. Über den Selector im LDTR-Register findet die CPU den Descriptor der LDT und damit die LDT:



Für 32bit-Programme wird die LDT unter Windows nicht verwendet.

5.3 Aufbau der Seiten-Tabellen: Page Directory und Page Table

- Seitengröße 4KB[*1], d.h. der 4GB-Adressraum enthält $\frac{4GB}{4KB} = \frac{2^{32}}{2^{12}} = 2^{20} = 1$ Mio Seiten.
- Würde man alle Seiten in einer einzigen Tabelle ("einstufig") verwalten und nimmt man an, dass jeder Eintrag 32bit groß ist, so würde die Seiten-Tabelle sehr viel Speicherplatz belegen:
  1 Mio Seiten x 4Byte = 4MB
- Daher verwendet Intel eine zweistufige Tabellen-Struktur mit **Page Directory** und **Page Table**:



- Der virtuelle Adressraum wird zusätzlich zu den 2^{12}Byte = 4KB [*1] großen Seiten (Page) in 2^{10} · 4KB große Seiten-Blöcke (Page Block) eingeteilt.
- Das CPU-Register CR3 zeigt auf die Basisadresse einer Tabelle, in der die Seiten-Blöcke aufgelistet sind (**Page Directory PD**). Die Page Directory Einträge (Page Directory Entry PDE) zeigen auf die Basisadressen von bis zu 1024 Seiten-Tabellen (Page Tables PT).
- Jede **Page Table** PT enthält 1024 Seiten-Einträge (Page Table Entry), von denen jeder auf genau eine Seite des physikalischen Speichers zeigt, d.h. jede Page Table verwaltet 4MB des Adressraums.
- Die PDE- und PTE-Einträge im Page Directory und in den Page Tables sind 32bit = 4Byte lang, so dass auch PD und PT 1024 · 4Byte = 4KB groß sind. Alle Tabellen müssen an Speicheradressen beginnen, die ganzzahlige Vielfache von 4KB sind, d.h. 00000000h, 00001000h, 00002000h, ...
- Da die meisten Programme deutlich weniger als 4GB Speicherbereich benötigen, legt das Betriebssystem nur für die Seiten-Blöcke Page Tables an, die tatsächlich verwendet werden. Da die Page Tables die selbe Größe haben wie normale Speicherseiten, kann das Betriebssystem die Page Tables bei Speicherplatzmangel ebenfalls auf die Festplatte auslagern.
- Wie in Abschnitt 2.5 beschrieben, werden **jedem Anwendungsprogramm** eigene Seiten-Tabellen und damit ein **eigener virtueller Adressraum zugeordnet**. Beim **Multitasking** erfolgt das Umschalten zwischen den Adressräumen durch einfaches **Umladen** des **Registers CR3**.
- **Sämtliche Einträge** in CR3, Page Directory und Page Tables sind **physikalische Speicheradressen**.

Format der Page-Directory- Einträge PDE bzw. Page-Table-Einträge PTE:

bit	*31*	...	*12*	*11 ... 9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	Physikal. Basisadresse bit 31...12			AVL	0	0*1	D	A	0	0	U	W	P

Zugriffsschutz durch Privilegstufen

U	User Bit	U = 0	Seite nur im Kernel Modus zugänglich
		U = 1	Seite im User und Kernel Modus zugänglich

Schreibschutz

W	Write Bit	W = 1	Seite schreib- und lesbar
		W = 0	Seite nur lesbar

Unterstützung des Swapping (Auslagern von Seiten auf die Festplatte)

P	Page Present	P = 1	Seite ist im Speicher
		P = 0	Seite ist nicht im Speicher, Zugriff löst Page Fault Exception aus, die ISR des Betriebssystems holt die ausgelagerte Seite dann von der Festplatte.

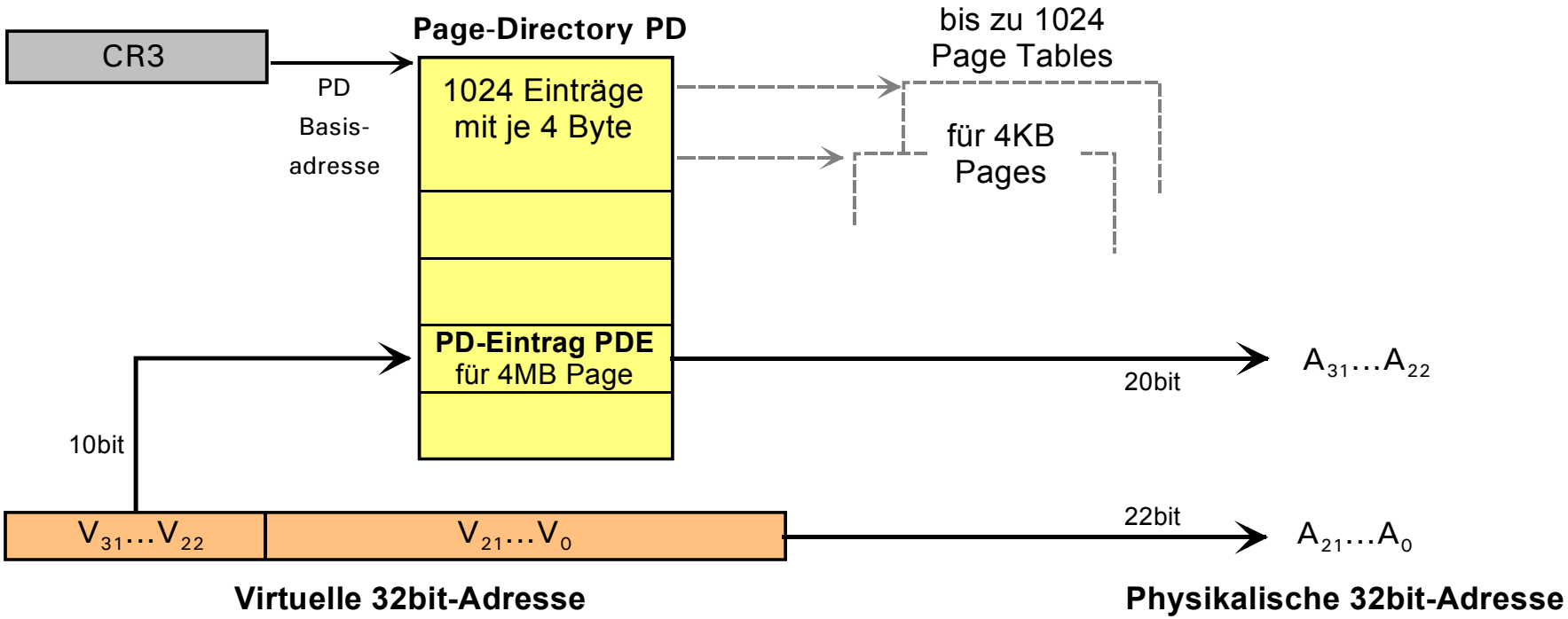
Zur Unterstützung des Betriebssystems, welche Speicherseite auf die Festplatte ausgelagert werden soll, dienen folgende Bits, die von der CPU automatisch gesetzt werden, wenn ...

A	Page Accessed	A = 1	... mindestens einmal auf die Seite zugegriffen wurde.
D	Dirty Page	D = 1	... mindestens einmal auf die Seite geschrieben wurde.

Seiten, auf die nie zugegriffen wurde, werden zuerst ausgelagert. Falls eine Seite schon früher ausgelagert wurde und noch in der Auslagerungsdatei steht, muss sie nur dann neu ausgelagert werden, wenn sie durch einen Schreibzugriff geändert wurde. Ansonsten kann sie einfach überschrieben werden. Mit den von der CPU selbst nicht verwendeten **AVL (Available Bits)** kann das Betriebssystem weitere Strategien zur Auslagerung implementieren, z.B. einen Least Recently Used Algorithmus).

*1 **4MB große Seiten:**

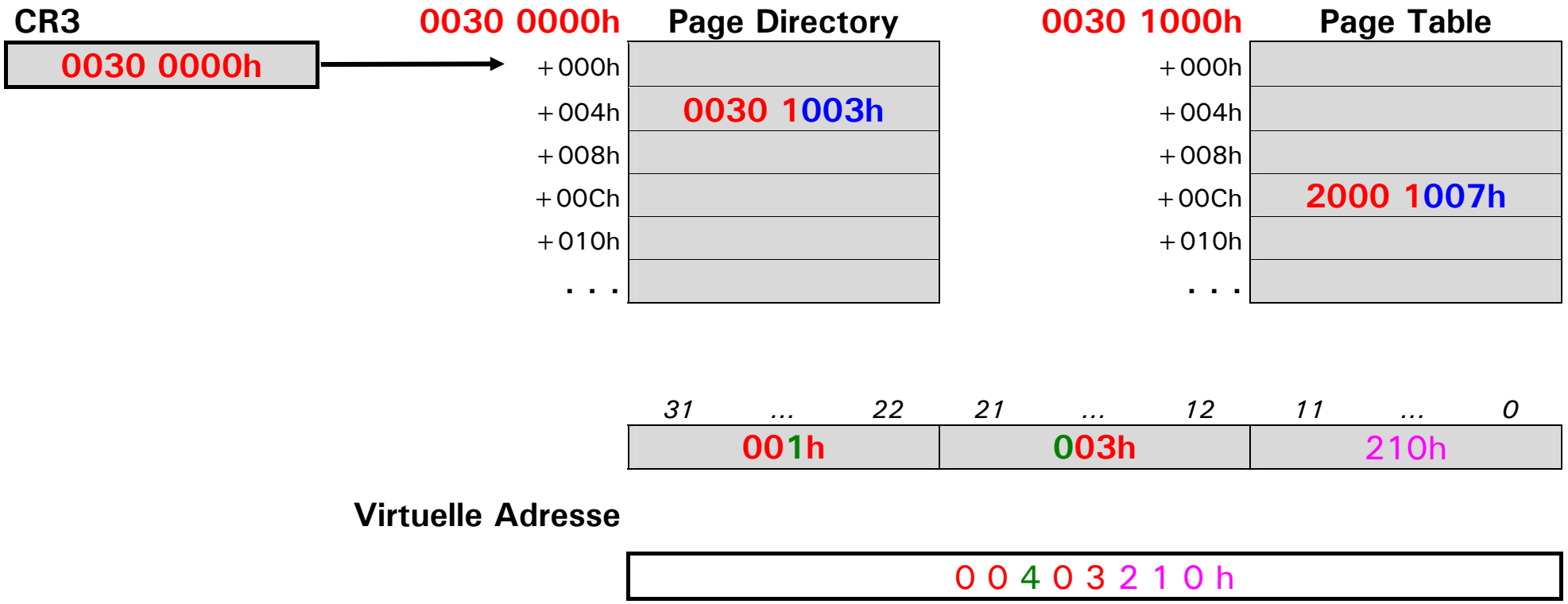
Wenn sehr große zusammenhängende physikalische Speicherbereiche vorhanden sind, kann der Verwaltungsaufwand reduziert werden, indem in einzelnen Einträgen des Page Directory auf eine Seitengröße von 4MB (statt 4KB) umgeschaltet wird, indem das Bit 7 des zugehörigen Page Directory Eintrags auf 1 (statt 0) gesetzt wird. Für den 4MB großen Speicherbereich entfällt die Page Table, der Page Directory Eintrag zeigt direkt auf die physikalische Anfangsadresse des Speicherbereichs.



4MB-Seiten müssen bei physikalischen Adressen beginnen, die ganzzahlige Vielfache von 4MB sind, d.h. 0000 0000h, 0040 0000h, 0080 0000h, ...

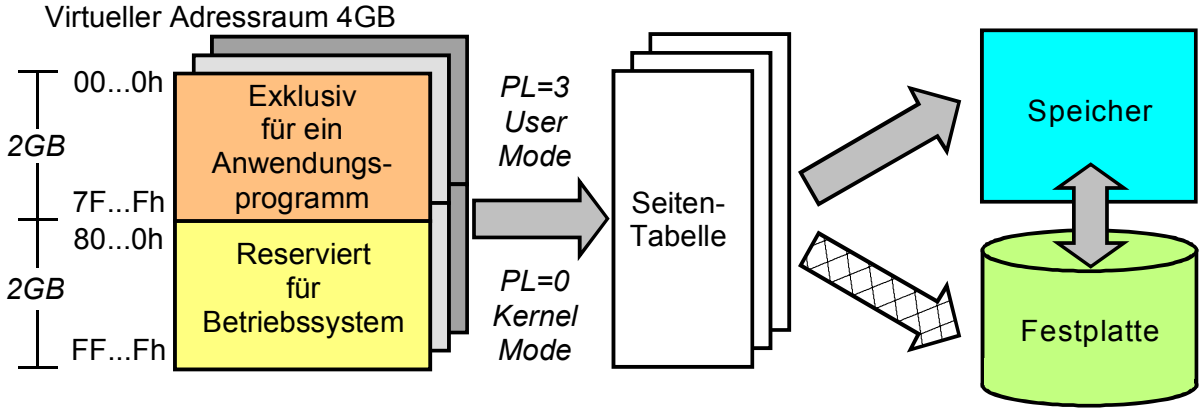
Beispiel:

- Ein Anwenderprogramm soll unter der virtuellen Adresse 0040 3210h auf eine Variable bei der physikalischen Adresse 2000 1210h zugreifen. Das Page Directory soll bei der physikalischen Adresse 0030 0000h beginnen, die Page Table für den Speicherblock, der die Variable enthält, direkt dahinter. Die Seitengröße sei 4KB.



Seiten-Tabellen unter Windows (und Linux)

- **Jedem Anwendungsprogramm** wird eine **eigene Seiten-Tabelle zugeordnet**, d.h. **beim Multitasking schaltet** das **Betriebssystem** die **Seiten-Tabelle um**, wenn es ein anderes Anwendungsprogramm ausführt. Dadurch hat jedes Anwendungsprogramm seinen eigenen virtuellen 4GB-Adressraum und physikalischen Speicherbereich und hat **kein**en **Zugriff auf** den physikalischen **Speicherbereich eines anderen Anwendungsprogramms**.



- Die **Speicherseiten für die oberen 2GB** des virtuellen Adressraums werden mit U = 0 gekennzeichnet, d.h. sie sind **nur** für das Betriebssystem **im Kernel Modus** zugänglich.
- Seiten für Programmcode und konstante Daten werden mit W = 0, Seiten für Variablen und den Stack werden mit W = 1 eingetragen.
- Speicherbereiche, die von einem Programm nicht genutzt werden, werden mit P = 0 im Page Directory bzw. in den Page Tables eingetragen. Fordert ein Programm mit malloc() dynamisch Speicher an, werden weitere Speicherseiten eingeblendet (**"Paging on Demand"**).

- Da das **Betriebssystem und** viele Bibliotheksfunktionen (in so genannten **Dynamic Link Libraries** (Windows Bezeichnung: DLL) bzw. Shared Libraries (Linux-Bezeichnung)) von mehreren Programmen verwendet werden, werden diese **nur einmal in** den **physikalischen Speicher geladen** („**Shared Memory“**) **und** über die Seiten-Tabellen **in** den **virtuellen Adressraum mehrerer Programme eingeblendet**. Die **gemeinsamen Speicherseiten** sind in den Seiten-Tabellen **als nur lesbar gekennzeichnet**. Soweit diese Seiten Variablen enthalten, die einem einzelnen Anwendungsprogramm zugeordnet sind, löst ein **Schreibzugriff** auf eine derartige Seite einen Ausnahmefehler-Interrupt (Page Fault Exception INT 14D) aus. Die zugehörige ISR des Betriebssystems legt dann eine nur für dieses Anwendungsprogramm zugängliche, schreibbare Kopie der Seite an ("**Copy on Write**").
- Falls der physikalische Speicher nicht ausreicht, werden Seiten vom Speicher in eine spezielle Datei auf der Festplatte kopiert (**Page File** oder **Swap File**) und in der Seiten-Tabelle als "**ausgelagert**" (P = 0) gekennzeichnet. Wenn die CPU auf den Adressbereich einer ausgelagerte Seite zugreift, wird ein Ausnahmefehler-Interrupt (**Page Fault Exception**) erzeugt. Die zugehörige ISR des Betriebssystems lagert dann eine andere Seite aus und kopiert die entsprechende Seite von der Festplatte zurück in den Speicher (**Swapping**). Beim Auslagern sucht das Betriebssystem diejenige Seite im Speicher, auf die am längsten nicht mehr zugegriffen wurde ("**Least Recently Used**"). Dabei wird das Betriebssystem durch das **A-Bit** sowie die vom Betriebssystem verwendeten AVL-Bits unterstützt. Seiten, die seit dem letzten Auslagervorgang nicht schreibend verändert wurden (**D = 0**), können ohne Auslagerung direkt neu verwendet werden, da sie bereits in der Auslagerungsdatei stehen.

5.4 Task State Segmente

80x86-CPUs besitzen mit dem Task Register TR und dem Task State Segment TSS einen Mechanismus, der die Realisierung von Multitasking-System vereinfachen würde. Da Windows aber leicht auf verschiedene andere CPU-Familien portiert werden sollte, die diesen Mechanismus nicht besitzen, verwendet Windows den 80x86-Multitasking-Mechanismus nicht. Trotzdem muss ein Teil der zugehörigen Verwaltungsstrukturen vom Betriebssystem initialisiert werden, weil er auch für andere Zwecke dient. Der unter Windows tatsächlich verwendete Teil wird hier beschrieben:

Das Task State Segment TSS ist ein normales Speichersegment, das durch einen Descriptor in der GDT beschrieben wird. Lediglich die Felder Zugriffsrechte und Segmentflags des Descriptors unterscheiden sich von einem normalen Speichersegment:

- Zugriffsrechte

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	1	DPL		0	1	0	0	1

- Segmentflags: 0h

Der Selector des Task State Segments wird vom Betriebssystem mit dem Befehl LTR ... in das 16bit CPU-Register TR (Task Register) geladen. Dieses Register kann auch von normalen Anwendungsprogrammen mit dem Befehl STR ... ausgelesen werden. Unter Windows ist normalerweise TR = 28h.

Die Daten innerhalb des TSS haben das auf der folgenden Seite dargestellte Format. Davon sind aber nur zwei Feldergruppen unter Windows tatsächlich von Bedeutung:

- Wie im Abschnitt 2.1 dargestellt, schaltet die CPU beim Wechsel der Privilegstufe automatisch den Stack um. Die Informationen über den Stack der aktuellen Privilegstufe speichert die CPU innerhalb des TSS und holt sich aus dem TSS die Informationen über den Stack der neuen Privilegstufe. Dabei werden in den Feldern ‚SS : ESP für CPL = 0' die aktuellen Werte des Kernel-Mode-Stacks und in den Feldern ‚SS : ESP' die aktuellen Werte des User-Mode-Stacks.
- Wie im Abschnitt 2.3 dargestellt, ist unter Windows normalerweise IOPL = 0, so dass I/O-Befehle nur im Kernel-Mode ausgeführt werden können. Für einzelne Ports kann das Betriebssystem diese Überwachung ausschalten und so für I/O-Befehle für bestimmte I/O-Adressen für Anwendungsprogramme freischalten. Dazu ist im TSS die bis zu 8kB große IO-Bitmap vorgesehen, bei der für jede I/O-Adresse des 64KB-I/O-Adressraums ein Bit anzeigt, ob die I/O-Adresse freigegeben ist oder nicht. Dieser Mechanismus wird z.B. vom Kernel-Treiber CrackNT.sys verwendet, um im Labor Rechnertechnik 2 auch unter

Windows NT/2000/XP auf die Labor-Hardware zugreifen zu können. Im Defaultzustand allerdings setzt Windows den Wert im Feld I/O-Bitmap-Offset auf einen Wert außerhalb der TSS und blockiert so diese Freigabe.

Task State Segment TSS

Offset-Adresse	← 32bit →
0	Reserviert (0)
4	ESP für CPL0
8	SS für CPL0
C	ESP für CPL1
10	SS für CPL1
14	ESP für CPL2
18	SS für CPL2
1C	CR3
20	EIP (Einsprungadresse)
24	EFLAGS
28	EAX
2C	ECX
30	EDX
34	EBX

	← 32bit →	
38	ESP	
3C	EBP	
40	ESI	
44	EDI	
48	ES	
4C	CS	
50	SS	
54	DS	
58	FS	
5C	GS	
60	0	LDT
⇐⇐ 64	I/O-Bitmap Offset	Reserviert (0)
⇓ 68	Beliebige Daten	
⇒⇒ . . .	I/O-Permission Bitmap	

Achtung:
Bei 32bit-Programmen werden die Segmentregister bei PUSH und POP-Operationen und im Task State Segment immer als 32bit-Werte behandelt, wobei die oberen 16bit auf 0 gesetzt werden.

5.5 V86-Modus: 16bit-Programme unter Windows

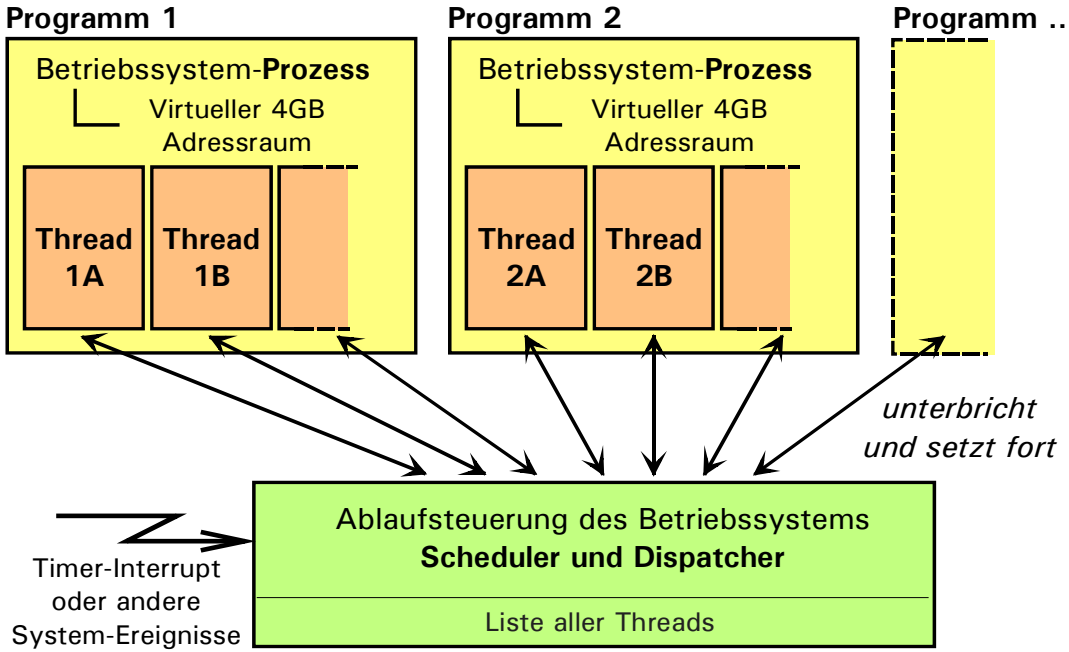
Um auch unter den Protected Mode-Betriebssystemen Windows bzw. Linux noch 'alte' 16bit-Real-Mode-Programme ausführen zu können, die für das Real-Mode-Betriebssystem DOS geschrieben wurden, kann ein Teil der in den Abschnitten 2.1-2.3 beschriebenen Überwachungsmechanismen der CPU abgeschaltet werden. Dazu schaltet das Betriebssystem die CPU vor Ausführung des Programms in den sogenannten V86-Modus, alle übrigen Programme arbeiten weiterhin im echten Protected Mode. Im V86 Modus ...

- ... erfolgt die Adressierung wie im Real Mode ohne Überwachung mit 16bit-Segment : 16bit-Offset-Adressierung, d.h. die Segment-Tabelle wird nicht verwendet.
- ... werden aber immer noch die Seiten-Tabelle (Paging) verwendet.
- ... sind die privilegierten Befehle weiterhin privilegiert, allerdings installiert das Betriebssystem andere Fehlerbehandlungs-ISRs. Für die bei DOS offiziell dokumentierten INT-Funktionen und Ein-/Ausgabe-Adressbereiche werden die Befehle dann durch diese ISRs tatsächlich ausgeführt, bei den anderen Funktionen spiegelt das Betriebssystem die erfolgreiche Ausführung lediglich vor.

Außer 16bit-Real Mode-Programmen, für die in jedem Fall der V86-Modus benötigt wird, gibt es auch ältere **16bit-Protected-Mode-Programme**, die aus der Zeit der Betriebssystemversion **Windows 3.x ("16bit-Windows")** stammen. Windows 3.x und auch die Consumer-Windows-Versionen **Windows 95, 98 bzw. ME** waren auf 16bit-DOS basierende Betriebssysteme, bei denen im Laufe der Zeit zunehmend größere Teile zunächst vom 16bit-Real-Mode in den 16bit-Protected-Mode und anschließend dann in den 32bit-Protected-Mode portiert wurden. Dasselbe gibt für viele Anwendungsprogramme, z.B. Microsoft Word, ursprünglich ein Real Mode Programm, das als Microsoft WinWord 2.0 bzw. 6.0 (in Office 95) als 16bit-Protected-Mode-Programm und dann als Teil von Office 1997 (und neuer) in den 32bit-Protected Mode portiert wurde. Mit **Windows NT, 2000, XP** ("32bit-Windows") hat der **16bit-Protected-Mode keine Bedeutung mehr**, obwohl auch darunter 16bit-Protected-Mode-Programme aus Gründen der Aufwärtskompatibilität noch ausgeführt werden können.

6. Interprozesskommunikation unter Windows NT/2000/XP

6.1 Problemstellung



- **Von einander** (weitgehend) **unabhängige Programme** werden vom Betriebssystem **als Prozesse** verwaltet. **Jedem Prozess** wird vom Betriebssystem ein **eigener virtueller Adressraum** (und eine Reihe weiterer Ressourcen, z.B. die Standard-Ein/ Ausgaben) **zugeordnet**. Die **Parallelisierung** von Aufgaben **innerhalb eines Prozesses** erfolgt **durch Threads**.
- Da jeder Prozess seinen eigenen Adressraum hat, ist ein **Datenaustausch zwischen** den **Prozessen** nicht direkt möglich.

- Das Betriebssystem stellt aber eine Reihe von Mechanismen wie Messages, Pipes, Shared Memory, Mailboxes/Mailslots, Sockets, Remote Procedure Call o.ä. bereit (**Inter-Prozess-Kommunikation**).

6.2 Erzeugen und Synchronisieren von Prozessen

Das folgende Programm startet einen zweiten Prozess B und geht dann in eine Schleife, in der ein Zähler hochgezählt und der Zählerstand ausgegeben wird. Bei einem Tastendruck wird die Schleife verlassen, der zweite Prozess zwangsweise beendet, bevor das Programm selbst ebenfalls endet:

Process1A.cpp

```
#include <windows.h>
#include <process.h>
...
int i;                                          // Zählervariable
HANDLE hProcessB;                               // Handle für den Prozess B
void main(void)                                 // Hauptprogramm
{   printf("----- Prozess A gestartet -----\n");
    hProcessB=(HANDLE) _spawnl(_P_NOWAIT, "process1B.exe", NULL);     // Prozess B erzeugen und starten
    while(!kbhit())                             // Endlosschleife, solange keine Taste gedrueckt wird
    { printf("+++ Prozess A: i=%8Xh\n", i);     // Zählerstand ausgeben
      i++;                                      // Zähler inkrementieren
    }
    TerminateProcess(hProcessB, NULL);          // Prozess B beenden
    printf("+++ Prozess A beendet\n");
}
```


Der zweite Prozess befindet sich in einer zweiten ausführbaren Datei. Dort wird ebenfalls ein Zähler inkrementiert und ausgegeben:

Process1B.cpp

```
#include <windows.h>
#include <process.h>
...
int k;                                          // Zählervariable

void main(void)                                 // Hauptprogramm
{   printf("----- Prozess B gestartet-----\n");

    while(1)                                    // Endlosschleife (ohne Abbruchbedingung)
    { printf("*** Prozess B: k=%8Xh\n", k);     // Zählerstand ausgeben
      k++;                                      // Zähler inkrementieren
    }
}
```

Nachdem der erste Prozess den zweiten Prozess gestartet hat, laufen die beiden Prozesse völlig unabhängig voneinander und die beiden Zähler zählen asynchron.

Einzelheiten zu den Aufruf- und Rückgabeparametern der hier und im folgenden verwendeten Funktionen _spawnl(), Terminate-Process() usw. finden Sie z.B. in der Dokumentation der Bibliothek des Microsoft C-Compilers bzw. der Win32-API.

Sollen Prozesse synchronisiert werden, ist dies wie bei der Synchronisation von Threads mit Hilfe von Events, Timern und Mutexen möglich:

Process2A.cpp

```
. . .
hEvent=CreateEvent(NULL, FALSE, FALSE, "myEvent");          // Event mit Namen "myEvent" erzeugen/öffnen
hProcessB=(HANDLE) _spawnl(_P_NOWAIT, "process2B.exe", NULL);     // Prozess B erzeugen und starten

while(!kbhit())                                             // Endlosschleife, bis Taste gedrueckt wird
{   printf("+++ Prozess A: i=%8Xh\n", i);                   // Zählerstand ausgeben
    i++;                                                    // Zähler inkrementieren
    SetEvent(hEvent);                                       // Prozess B über Event "myEvent" triggern
    Sleep(100);                                             // 100ms Wartezeit
}

. . .
```

Process2B.cpp

```
. . .
hEvent=CreateEvent(NULL, FALSE, FALSE, "myEvent");          // Event mit Namen "myEvent" erzeugen/öffnen

while(1)                                                    // Endlosschleife (ohne Abbruchbedingung)
{   WaitForSingleObject(hEvent, INFINITE);                  // Warten, bis Event durch Proz. A getriggert wird
    printf("*** Prozess B: k=%8Xh\n", k);                   // Zählerstand ausgeben
    k++;                                                    // Zähler inkrementieren
}
. . .
```

Beim Erzeugen des Events (analog bei Mutexen usw.) muss in beiden Prozessen derselbe Name angegeben (hier myEvent). Bei Events innerhalb von Threads konnte der Name entfallen.

Die Zähler in den beiden Prozessen laufen nun synchron.

6.3 Datenaustausch zwischen Prozessen

Der einfachste Datenaustausch zwischen Prozessen erfolgt über **„Shared Memory“**, d.h. über einen gemeinsamen Speicherbereich. Das Betriebsystem verwendet hierbei das Paging der CPU, d.h. derselbe physikalische Speicherbereich wird über die Seitentabellen in die virtuellen Adressräume der Prozesse eingeblendet:

Process4A.cpp

```
HANDLE hProcessB, hSM;
int *pSMa;                                          // Pointer auf den Shared Memory Bereich
. . .                                               // SharedMemory "mySharedMemory" erzeugen ...
hSM = CreateFileMapping((HANDLE) –1, NULL, PAGE_READWRITE, 0, sizeof(int), "mySharedMemory");
pSMa = (int*) MapViewOfFile(hSM, FILE_MAP_ALL_ACCESS, 0, 0, 0);  // ... und in den Adressraum einblenden
. . .
*pSMa = 0;                                          // Zugriff auf den Shared Memory Bereich via Pointer
. . .
```

Process4B.cpp

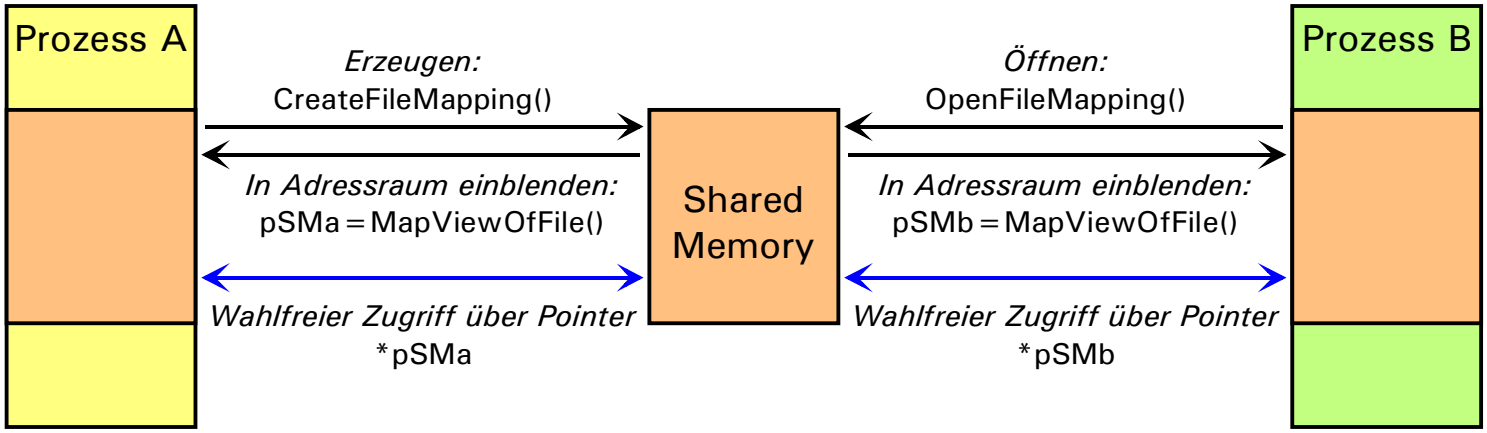
```
HANDLE hSM;
int *pSMb;                                          // Pointer auf den Shared Memory Bereich

hSM = OpenFileMapping(FILE_MAP_ALL_ACCESS,TRUE,"mySharedMemory");// Shared memory öffnen ...
pSMb = (int*) MapViewOfFile(hSM, FILE_MAP_ALL_ACCESS, 0, 0, 0);  // ... und in den Adressraum einblenden
. . .
*pSMb = *pSMb + 1;                                  // Zugriff auf den Shared Memory Bereich via Pointer
. . .
```

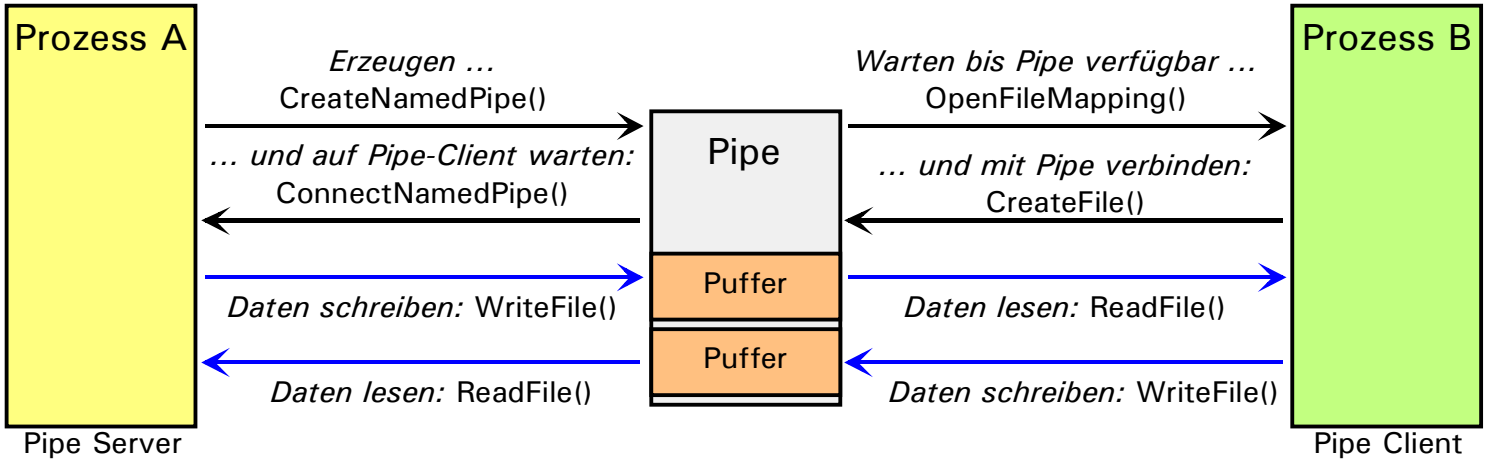
Beide Prozesse greifen über die beiden Pointer pSMa, pSMb auf denselben Speicherbereich zu.

In der Regel ist für den Datenaustausch über den Shared Memory Bereich eine Synchronisation über Threads bzw. ein gegenseitiger Ausschluss über Mutexe sinnvoll.

Datenaustausch über Shared Memory



Datenaustausch über Named Pipe



Im Gegensatz zum Shared Memory, der sich verhält wie ein Block von globalen Variablen, auf den beliebig zugegriffen werden kann, erfolgt der Datenaustausch über eine **Pipe** wie der Datenaustausch über Dateien („**Stream**“). Ein Prozess schreibt Daten zeichenweise in die Pipe, die vom Betriebssystem zwischengespeichert und zu einem beliebigen späteren Zeitpunkt wiederum zeichenweise vom zweiten Prozess gelesen werden können und umgekehrt. Beim Lesen wird dabei darauf gewartet, dass Zeichen in der Pipe stehen. Auf diese Weise wirkt eine Pipe auch synchronisierend.

Bei der Verbindungsaufnahme übernimmt ein Prozess die Aufgabe eines **Pipe-Server**s, der die Pipe erzeugt. Ein oder mehrere andere Prozesse können als **Pipe-Client**s Verbindungen zu diesem Server aufnehmen. Im Gegensatz zu Shared Memory lässt sich mit einer Pipe auch eine Verbindung über ein Netzwerk zwischen verschiedenen Rechnern herstellen.


```
HANDLE hPipe;                                                    Pipe-Server  Process3A.cpp
. . .
                                                        // Pipe "myPipe" erzeugen
hPipe=CreateNamedPipe("\\\\.\\pipe\\myPipe",PIPE_ACCESS_DUPLEX, 0, PIPE_UNLIMITED_INSTANCES, 0, 0, INFINITE, NULL);
. . .
ConnectNamedPipe(hPipe, NULL);                          // Warten, bis der andere Prozess sich mit der Pipe verbindet
while(!kbhit())                                         // Endlosschleife, solange keine Taste gedrueckt wird
{  WriteFile(hPipe, &i, sizeof(i), (DWORD *) &k, NULL);     //--- Wert an die Pipe senden
   ReadFile(hPipe,  &i, sizeof(i), (DWORD *) &k, NULL);     //--- Wert von der Pipe lesen
  . . .
}
DisconnectNamedPipe(hPipe);                             // Pipe-Verbindung abbrechen
. . .
```

```
HANDLE hPipe;                                                    Pipe-Client  Process3B.cpp
. . .
WaitNamedPipe("\\\\.\\pipe\\myPipe", INFINITE);         // Warten, bis die Pipe "myPipe" verfügbar ist
                                                        // Pipe "myPipe" zum Lesen und Schreiben öffnen ("verbinden")
hPipe = CreateFile("\\\\.\\pipe\\myPipe", GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
. . .
ReadFile(hPipe,  &i, sizeof(i), (DWORD *) &k, NULL); //--- Wert aus der Pipe lesen
. . .
WriteFile(hPipe, &i, sizeof(i), (DWORD *) &k, NULL); //--- Wert an die Pipe senden
. . .
CloseHandle(hPipe);                                     // Pipe schliessen
. . .
```

