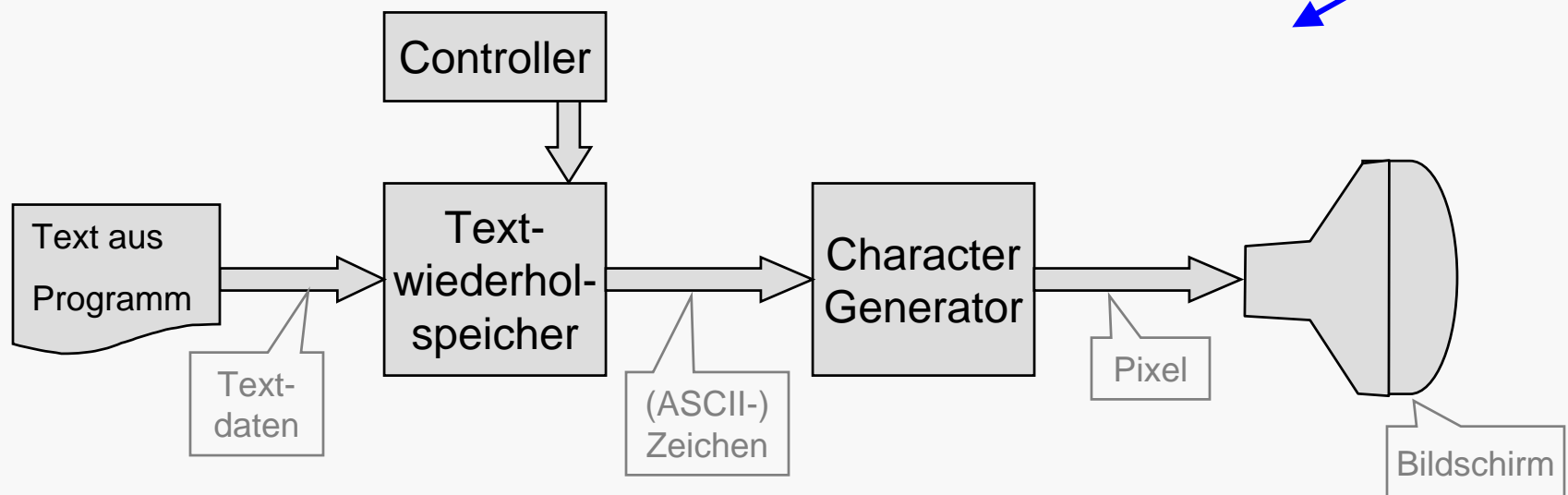


Vorteile der Anwendung grafischer Benutzungsschnittstellen:

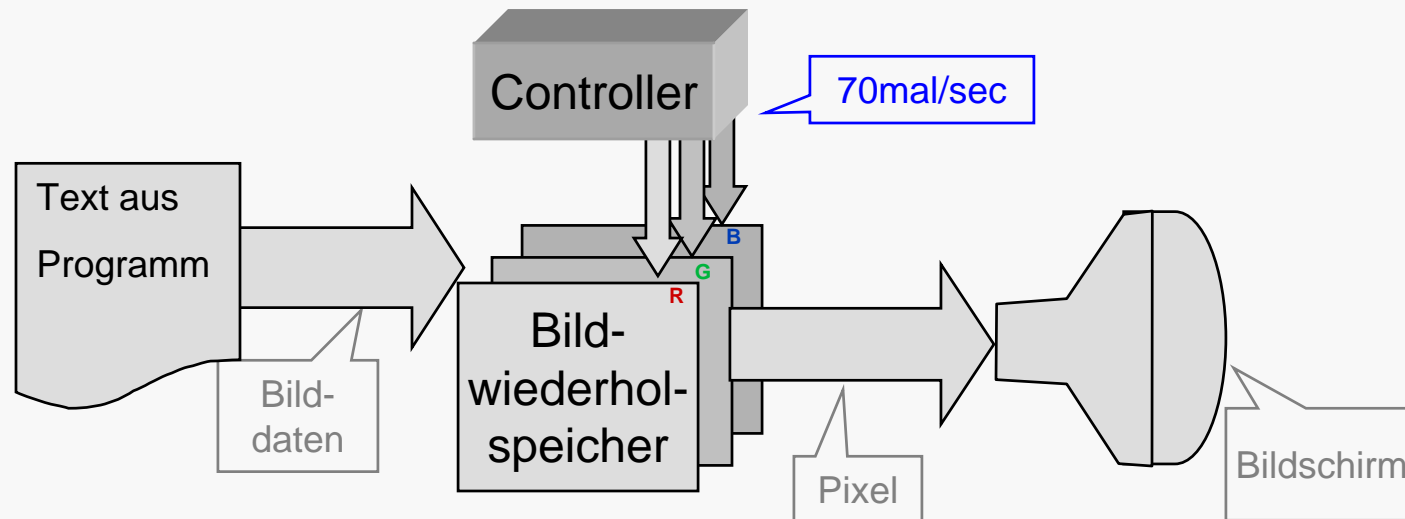
- Intuitive Dialoggestaltung
(z.B.: Animierte Tastatur-Bilder für Einloggen)
↳ Leichte Erlernbarkeit von Programmen
- Ähnliche Strukturierung unterschiedlicher Programme
(z.B.: „Datei - Bearbeiten - Ansicht - ... /Senden an:)
↳ Schnelle Einbeziehung neuer Lebensbereiche in DV
- Fortentwicklung des Bildschirms zum „Eingabegerät“
↳ Interaktion: Computer als Arbeits-/Lernpartner
- WYSIWYG *
↳ Planbarkeit von Arbeit u. Ergebnis

* Akronym: „What You See Is What You Get“
(bzgl. eines Text-Druckbildes)

- Starke Ähnlichkeit zwischen GUIs von MS-Windows, Apple, X Window System u.a.
- Grundlage: Forschung am Xerox PARC Mitte der 70er (damals kontrovers, heute Standard)
- Voraussetzung für GUI: Ablösung der alphanumerischen durch (raster- o.) pixelorientierte Sichtgeräte



Pixelorientierte Sichtgeräte:



Daten-Aufkommen im Vergleich:

- monochromes, alphanumerisches Display vs.
- pixelorientiertes RGB-Display

- Annahme:

25-Zeilen-Display; 80 Zeichen/Zeile; 8x16 Pixel/Zeichen

⇒ 25 Textzeilen x 80 Zeichen/Textzeile = 2.000 Zeichen

⇒ 8x16 Pixel/Zeichen = 128 Pixel/Zeichen

⇒ 25 Textzeilen x 16 Pixel/Textzeile = **400** Pixel (hoch)

⇒ 80 Zeichen x 8 Pixel / Zeichen = **640** Pixel (breit)

- monochromes, alphanumerisches Display:

Character Generator (ROM):

256 Zeichen x 128 Pixel/Zeichen x **1** Bit/Pixel = 32.768 Bit = 4kB

Textwiederholtspeicher (RAM):

2.000 Zeichen x 1 Byte/Zeichen = 2.000 Byte = **1,95 kB**

- monochromes, pixelorientiertes Display:

Bildwiederholtspeicher (RAM):

640 Pixel/Zeile x 400 Zeilen x 1 Bit/Pixel = 256.000 Bit = **31,25 kB**

- Annahme:
pixelorientiertes 1280 x 1024-Pixel-RGB-Display mit einer
Bildwiederholrate von 70 Hz (non-interlaced):

- Bildwiederholtspeicher (RAM):
 $1280 \text{ Pixel/Zeile} \times 1024 \text{ Zeilen} \times 3 \text{ Byte/Pixel} = \mathbf{3,75 \text{ MB}}$
- Datenrate bei 70 Hz:
 $3,75 \text{ MB} \times 70/\text{sec} = \mathbf{262,5 \text{ MB/sec}}$

Zum Vergleich:

- monochromes, alphanumerisches Display (handelsunüblich)
 $1280 \text{ Pixel/Zeil.} \times 1024 \text{ Zeil.} \times 1 \text{ Byte/Zeich.} / 128 \text{ Pixel/Zeich.}$
 $= \mathbf{10 \text{ kB}} \text{ RAM}$
- Datenrate bei 70 Hz:
 $10 \text{ kB} \times 70/\text{sec} = \mathbf{0,68... \text{ MB/sec}}$
➡ GUI braucht zusätzliche Hw (Grafikkarte)

GUI seit Windows NT (07/93) u. Windows 95 (08/95):

➔ Nutzung der **32-Bit-Technologie** (ab i386, '85):

„Win32-API“: Lineares 32-Bit-„flat addressing“ für Windows NT u. Windows9x gleich (Win9x: intern z.T. aufgehoben); weiter Unterstützung der Win16-API

früher: Speicher-Stückelung in 64 kB-Segmente und Adressierung mit (16-Bit-) „far pointer“ (Segment) und „near pointer“ (Offset).

➔ **Präemptives Multitasking:**

Betriebssystem entscheidet über Rechenzeit-Zuweisung („verdrängender Mehrprozeßbetrieb“).

➔ **Dynamische Speicherverwaltung:**

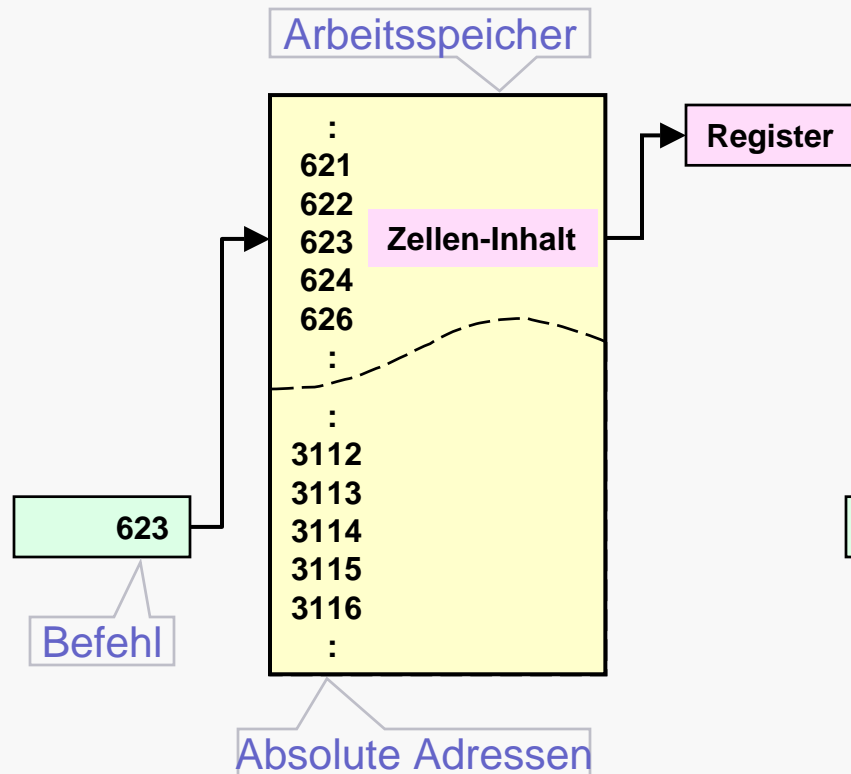
Umordnung freiwerdenden Arbeitsspeichers zur Vermeidung von Fragmentierung.

➔ **Indirekte Adressierung** (Indirektion):

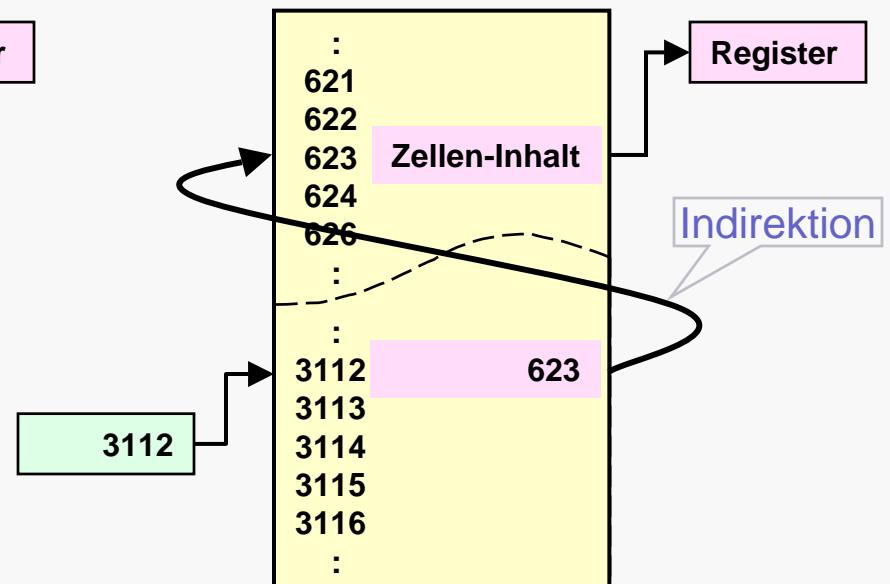
Symbolische Adressen anstelle physischer ⇒ effektive / robuste Speicher-Reorganisation

➔ (Prinzip der Indirektion)

Direkte Adressierung:



Indirekte Adressierung:



Win32 (Forts.)

➔ **Virtueller Speicher:**

Auslagerung nicht (bzw. selten) genutzter Programmteile auf die Festplatte

➔ **Dynamic-link libraries (*.DLL, *.EXE):**

Mehrfach benötigter Code wird nur einmal geladen u. ggf. von aktuell laufenden Programmen mitgenutzt („Dynamisch gebundene Bibliotheken“, z.B.:
Fenstereinrichten - vgl. UNIX: Shared Library)

Funktionsaufrufe eines typischen Anwendungsprogramms fallen in eine von 3 Kategorien von DLL-Aufrufen:

Drei wichtige (früher: einzige) DLL's:

- **KERNEL32.DLL (16-Bit: KRNL386.EXE):**
Systemkern mit
 - Speicherverwaltung (memory management),
 - Datei-Management (file I/O) und
 - Verwaltung von Prozessen (tasking)
- **USER32.DLL (16-Bit: USER.EXE):**
 - Benutzeroberfläche u. Fensterlogik (UI, windowing logic)
- **GDI32.DLL (16-Bit: GDI.EXE):**
 - Grafische Geräteschnittstelle (Graphics Device Interface)

- DOS 2.0: Hintergrundprozesse als „**TSR-Programme**“:
„Terminate and Stay Resident“ (d.h.: keine Speicher-Freigabe)
ermöglichte Verfügbarkeit/ Reaktivierbarkeit (meist über „Hotkey“)
 - z.B.: Verwaltung / Versorgung von Druckaufträgen (Screenshot, Drucker-Spooler); nicht-englische Tastaturtreiber
- 16-Bit-Windows(1985): **Nicht-präemptives Multitasking**:
Peripherie (Tastatur, Maus) konnte „Nachricht“ (Aufruf) auslösen,
die inaktives Programm im Speicher aktivierte.
Programm mußte für Kontroll-Abgabe konzipiert sein
(„kooperatives“ Multitasking), sonst Blockierung möglich.
- OS/2 (MS/IBM, 1988): **Präemptives Multitasking**:
Sw („Presentation Manager“) ordnete Ereignisse den laufenden
Prog. zu. Warten auf (serielle) Ereignis-Zustellg wirkte entgegen.
 - z.B.: lange Text-Sicherung verzögert Fokus-Wechsel

- 32-Bit-Windows (1995): **Multithreading:**

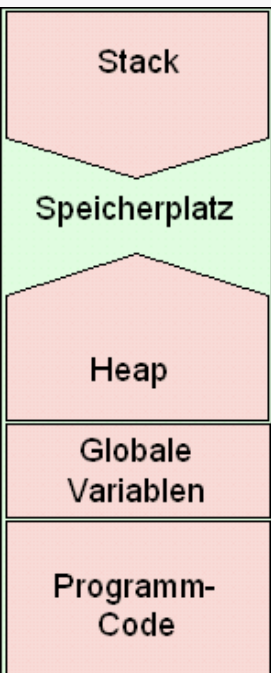
Aufteilung der Anweisungsfolgen eines Programms in mehrere „Ausführungsstränge“, die parallel (präemptiv) ausgeführt werden.

Der Hauptthread (*primary*) kann weitere (*secondary*) Threads erzeugen und diese wiederum andere. Alle Threads

- erhalten (verdrängend) Kontrolle/CPU-Zeit wie separate Programme
- bekommen eine eigene Nachrichtenwarteschlange zugewiesen
- gehören zum selben Prozeß (teilen Speicher und **static**-Variablen)
- haben eigene Stacks (lokale Var. und Prozessor-/Coproz.-Zustand)

Wichtige Kriterien bei der Einrichtung von Threads:

- Hauptthread f. Erzeugung v. Fenstern u. Verteilung v. Nachrichten, alle anderen zur Erledigung von Aufgaben im Hintergrund.
 - z.B.: Windows-Aufrufe jederzeit bedienen
- „1/10-sec-Regel“: was länger braucht, gehört in den Hintergrund
 - z.B.: Menü-/Programm-Nutzung während langwieriger Speicherung
- Thread-Abspaltung auf Sinnfälligkeit überprüfen
 - z.B.: Bildbearbeitung während Bildladen fragwürdig



Anlegen und Starten des Threads **ThreadProc ()**:

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

HANDLE CreateThread (LPSECURITY_ATTRIBUTES lpThrAttr,
                    DWORD dwStackSize, DWORD WINAPI ThreadProc,
                    LPVOID lpParam, DWORD dwFlags, LPDWORD lpThreadId);
```

Thread-
Kennung

Sofortiges Beenden eines Threads (bei **return** entbehrlich):

```
VOID ExitThread (DWORD dwExitCode); //beendet ThreadProc
BOOL CloseHandle(HANDLE handle); //Resrc.freigabe(@TRUE)
```

Parameter:

lpThrAttr:	Zeiger auf interne Struktur (betr.InterProz.-Kommun.): NULL
dwStackSize:	Startwert Stackgröße (0: Standard ⇒ autom. Anpassung)
ThreadProc:	Adresse DWORD WINAPI ThreadProc (PVOID *lpParam)
lpParam:	(32-Bit-)Argument von ThreadProc ()
dwFlags:	0 o. CREATE_SUSPENDED : wartet bis ResumeThread -Aufruf
lpThreadId:	Zeiger auf 32-Bit-Var., die Thread-Kennung erhält (o. NULL)
dwExitCode:	Exit-Code (individuelle 32-Bit-Variable – z.B. lpParam)

Empfohlene Funktion zum Anlegen und Starten von **ThreadProc ()**:

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

```
unsigned long _beginthread(  
void(__cdecl *ThreadProc)(void *),  
unsigned uiStackSize, void *pParam);
```

Thread-Kennung
(Handle)

Sofortiges Beenden und Ressourcen-Freigabe eines Threads:

```
void _endthread (void); //bei return entbehrlich
```

Parameter:

ThreadProc: Startadresse von void **ThreadProc**(void ***pParam**)
[genauer: void __cdecl ThreadProc (void ***pParam**);
__cdecl: „default calling convention for C / C++ programs“]

uiStackSize: anfängliche Stackgröße;
0: Standardwert (1 MB) – bei Bedarf automat. Anpassung

pParam: (32-Bit-)**ThreadProc ()**-Argument; meist: Struktur-Zeiger
Einzige Möglichkeit zur Thread-Individualisierung!

Thread-Synchronisation (I)

Blockierung des generierenden (primären) Threads, damit seine Beendigung nicht vorzeitig die sekundären Threads beendet:

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

```
DWORD WaitForMultipleObjects (DWORD nCount,  
    CONST HANDLE *handle, BOOL WaitAll, DWORD msec);
```

Parameter:

nCount: Anzahl zu synchronisierender (sekundärer) Threads

handle: Zeiger auf Feld mit (Thread- o. anderen Objekt-)Handles

WaitAll: Typ des Wartens (**TRUE**: warten auf alle Threads;
FALSE: warten, bis irgendein Thread fertig ist)

msec: max. Wartezeit in Millisekunden (**INFINITE**: beliebig)

Rückgabewert: Grund zur Beendigung des Wartens (Kennzahl)

```
C:\WINDOWS\system32\cmd.exe

[Idx:1]    Zeit:    8    [Weckzeiten:    5    10    15    8    71
```

Prof.Dr.A.Christidis•WS 2011/12

Beispiel: Multi-Timer

```
typedef struct
{ time_t start, tBell[NUM];
} timerDat; // #define NUM 5

void bell(void *idx)
{ static timerDat *tim=NULL;
  time_t tj=0, tTick=0; // int j1=0;
  if (!tim) tim = init();
  do { time(&tj);
    if (tj>=tTick){ system(CLS);
      /* (...) */

      for (j1=0; j1<NUM; j1++)
        /* ..printf().. */
        tTick++; }
  } while(tj<tim->tBell[(int)idx]);
  printf("\a"); // beep
  return; }
```

```
timerDat *init(void) // Speicher:
{ static timerDat tim;
  tim.tBell[0]=5; tim.tBell[1]=10;
  tim.tBell[2]=15; tim.tBell[3]=8;
  tim.tBell[4]=7;
  time(&tim.start); return(&tim);
}

void multiBell(void)
{ int j1=0;
  for(j1=0; j1<NUM; j1++)
    bell((void*)j1);

  return;
}

int main (void)
{ multiBell(); system(CLS);
  return 0; } (MultiBell.exe)
```

int ⇔ Zeiger
konvertierbar

```
C:\WINDOWS\system32\cmd.exe

[Idx:1]    Zeit:    8    [Weckzeiten:    5    10    15    8    71
```

Prof.Dr.A.Christidis•WS 2011/12

Beispiel: Multi-Timer (Thread-Version)

```
typedef struct
{ time_t start, tBell[NUM];
} timerDat; // #define NUM 5

void bell(void *idx)
{ static timerDat *tim=NULL;
  time_t tj=0, tTick=0; // int j1=0;
  if (!tim) tim = init();
  do { time(&tj);
    if (tj>=tTick){ system(CLS);
      /* (...) */

      for (j1=0; j1<NUM; j1++)
        /* ..printf().. */;
        tTick++; }
  } while(tj<tim->tBell[(int)idx]);
  printf("\a"); // beep
  _endthread(); return; }
```

```
timerDat *init(void) // Speicher:
{ static timerDat tim;
  tim.tBell[0]=5; tim.tBell[1]=10;
  tim.tBell[2]=15; tim.tBell[3]=8;
  tim.tBell[4]=7;
  time(&tim.start); return(&tim);
}

void multiBell(void)
{ int j1=0;
  unsigned long hThread[NUM];
  for(j1=0; j1<NUM; j1++) hThread[j1]=
    _beginthread(bell, 0, (void*)j1);
  WaitForMultipleObjects(NUM,
    (CONST HANDLE)hThread, TRUE,
    INFINITE); return;
}

int main (void)
{ multiBell(); system(CLS);
  return 0; } (MultiBellThread.exe)
```


Thread-Synchronisation (II)

Einrichtung kritischer Abschnitte zur Verhinderung v. Kollisionen bei Inanspruchnahme exklusiv nutzbarer Ressourcen; Implementierung als Instanz einer (Windows-internen) Struktur `LPCRITICAL_SECTION`, die nur durch 4 Systemfunktionen manipuliert werden darf:

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

Vereinbarung und Initialisierung der benötigten Struktur für den Schutz einer Ressource:

```
CRITICAL_SECTION cs;
```

```
VOID InitializeCriticalSection(LPCRITICAL_SECTION *cs);
```

Eintritt in den bzw. Austritt aus dem kritischen Abschnitt:

```
VOID EnterCriticalSection (LPCRITICAL_SECTION *cs);
```

```
VOID LeaveCriticalSection (LPCRITICAL_SECTION *cs);
```

zeit-
kritisch!

Freigabe der Ressourcen eines kritischen Abschnitts:

```
VOID DeleteCriticalSection (LPCRITICAL_SECTION *cs);
```



Prof.Dr.A.Christidis•WS 2011/12

Beispiel: Multi-Timer (Thread/CS-Version)

```
typedef struct
{ /*...*/ CRITICAL_SECTION cs4pr;
} timerDat;
```

Thread-Identifikation!

```
void bell(void *idx)
{ static timerDat *tim=NULL;
  if (!tim) tim = init();
  /* (...) */
  EnterCriticalSection
    (&tim->cs4pr);
  /*system(CLS); printf(...)*
  LeaveCriticalSection
    (&tim->cs4pr);
  /* (..._endthread();...) */
  return;
}
```

```
void multiBell(void)
{ int j1=0;
  unsigned long hThread[NUM];
  static timerDat *tim=NULL;
  if (!tim) tim = init();
  InitializeCriticalSection
    (&tim->cs4pr);

  for (j1=0; j1<NUM; j1++)
  { hThread[j1] = _beginthread
    (bell,0,(void *)j1);
    }
    „universeller Zeiger“!
  WaitForMultipleObjects
    (NUM, (CONST HANDLE)hThread,
    TRUE, INFINITE);
  DeleteCriticalSection
    (&tim->cs4pr);

  return;
}
```

(MultiBellThreadCS.exe)

Thread-Synchronisation (III)

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

Thread-Blockierung (-Suspendierung) für **msec** Millisekunden:

```
VOID Sleep(DWORD msec);
```

Verknüpfung d. **WaitFor...**-Funktionen mit Ereignissen ohne Handle:

„Event“ erzeugen, setzen, zurücksetzen:

```
HANDLE CreateEvent (LPSECURITY_ATTRIBUTES lpThrAttr,  
Event-Kennung      BOOL bManReset, BOOL bInitState, LPCTSTR lpName);  
BOOL SetEvent (HANDLE handle); //Event gesetzt (return != 0)  
BOOL ResetEvent (HANDLE handle); //Ev. zurückgesetzt (r. != 0)
```

Parameter:

lpThrAttr : Zeiger auf interne Struktur (betr. InterProz.-Kommun.): **NULL**
bManReset : Erwartetes Zurücksetzen (**TRUE**=„manuell“: durch Aufruf v. **ResetEvent**; **FALSE**: autom., nach d. ersten Thread-Start)
bInitState : Anfangszustand (**TRUE**: gesetzt; **FALSE**: zurückgesetzt)
lpName : Zeiger auf Event-Namen („sz“: string terminated with a zero")

Bemerkungen zu den vorausgegangenen Folien:

- **Hotkey**: Taste(nkombination) zur Aktivierung einer bestimmten Rechner-(BS-) Funktion.
- **Spool**: Simultaneous Peripheral Operations OnLine
- **Nonpreemptive (cooperative) Multitasking** wird ins Deutsche als „nicht-verdrängender Mehrprozeßbetrieb“ übertragen („preemption“ bezeichnet allg. das Vorkaufsrecht).
- **Thread** wird ins Deutsche als „Ausführungsstrang“ übertragen (engl. für: „Faden, Garn, Zwirn“) ⇒ Abwicklungsfolge: Kleinste Verarbeitungseinheit (BS, Appl.), der Rechenzeit zugewiesen wird.
- Typische Größenangaben: max. 2028 Threads je 1MB Stack; 20 msec (=1/50 sec) je Time Slice
- Der Bezeichner `__cdecl` ist (nur) insofern von Interesse, als er die Standard-Einstellungen des Compilers wieder einsetzt, falls sie zuvor außer Kraft gesetzt worden waren.
- Die Funktion `DWORD WaitForSingleObject(HANDLE handle,DWORD msec);` wurde hier nicht näher besprochen, weil sie gegenüber `WaitForMultipleObjects` (abgesehen vom Handle-Typ `HANDLE` gegenüber `CONST HANDLE`) hauptsächlich eine eingeschränkte Funktionalität bietet.
- `CreateThread()` hat gegenüber `_beginthread()` den Vorzug, daß sein Bezeichner, seine verwendeten Datentypen und sein "Outfit" zu einer "Familie" gehören (vgl. `CreateWindow`).
Seine Nachteile sind (u.a.):
 - Es benötigt mehr (und selten genutzte) Parameter, getrenntes `Exit` und `Close` (Erlernen, Rechenzeit, Fehler-Quellen);
 - Es erfordert eine Thread-Funktion vom nicht-universellen Typ `DWORD WINAPI ThreadProc()`;
 - Es kann zu „geringem Speicherschwund“ (small memory leaks) führen, wenn es mit der C-Laufzeitbibliothek verwendet wird. (BS bzw. Applikation)
- Bsp.: s. SysProg-Klausur WS 04/05