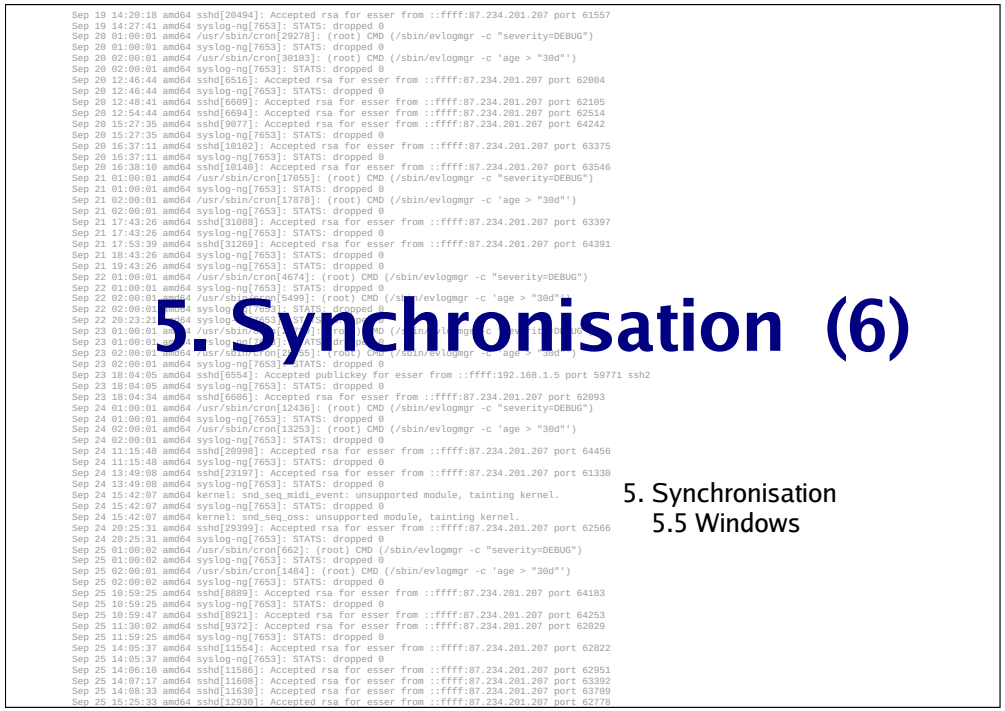
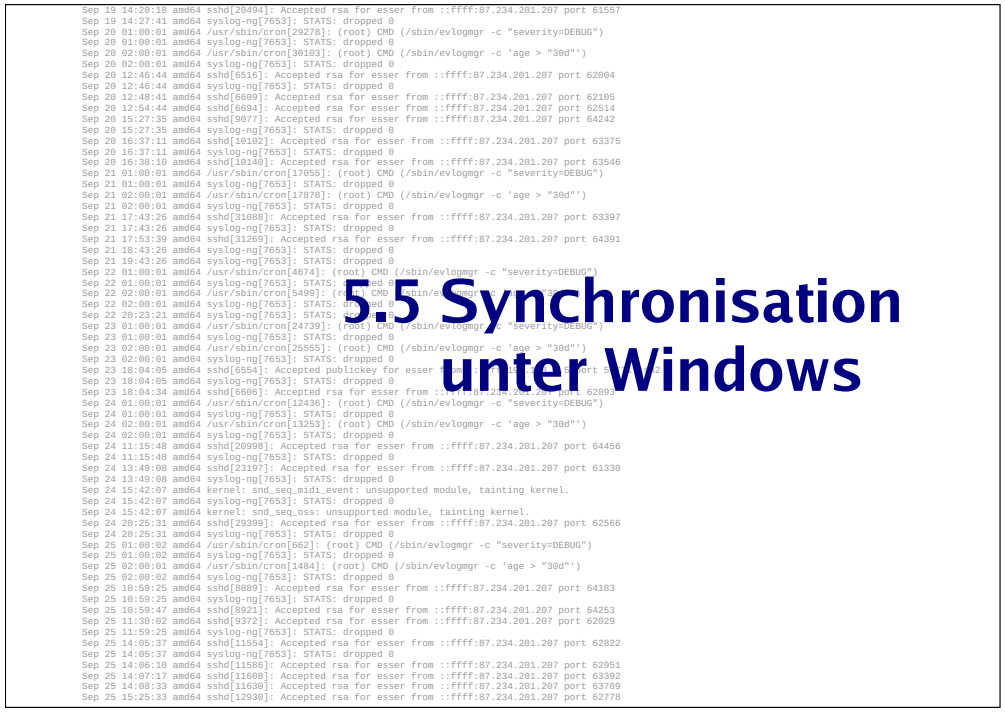


5. Synchronisation (6)

5. Synchronisation
5.5 Windows



5.5 Synchronisation unter Windows

Windows: Prozesse und Threads

- Prozesse
  - Programmstart erzeugt neuen Prozess
  - kein **fork()** wie bei Unix / Linux; Prozesserzeugung startet immer ein neues Programm
  - Prozess definiert durch: Adressraum, Ressourcen, Sicherheitsprofil
- Threads
  - separate Kontrollflüsse (wie Unix / Linux)
  - gemeinsamer Adressraum (der des Prozesses)
  - aber: jeder Prozess besteht aus mind. einem Thread
  - Scheduler aktiviert Threads (nicht Prozesse!)
  - Synchronisation: krit. Bereich, Mutex, Event, Semaphor

fork & exec?

- Für den Unix-**fork()**-Aufruf gibt es unter Windows keinen gleichartigen Ersatz
- Bei Portierung von Unix-Programmen nach Windows muss man die Programmlogik von **fork()**-basierten Anwendungen ändern
- Die Kombination **fork()** & **exec()** für den Start eines neuen Programms ersetzt **CreateProcess()**
- statt **fork()** (ohne **exec**) mit Threads arbeiten

fork *ohne* exec

Linux: fork()

```
main(){

...

while (TRUE) {
  read (&aufgabe);
  if (aufgabe=="new") {
    pid = fork();
    if (pid==0) { /* child process */
      do_some_work();
      exit(0); /* Proz. beenden */
    }
    else {        /* parent process */
    }
  }
}

...
wait();
}
```

Windows: CreateThread()

```
main(){

...

while (TRUE) {
  read (&aufgabe);
  if (aufgabe=="new") {
    CreateThread(do_some_work, ...);
  }
}

...
WaitForMultipleObjects(...)
}
```


Windows vs. Linux

- multi-threaded – von Anfang an
- bei Prozessstart nur ein Thread

Prozess

Speicher
Dateien
...

(hier kein Code)

Haupt-Thread

```
main(){
  ...
  ...
}
```

- Standard: single-threaded
- multi-threaded nur auf Wunsch

Prozess

Speicher
Dateien
...

```
main(){
  ...
  ...
}
```


Windows vs. Linux

nach **CreateThread()**

Prozess

Speicher
Dateien
...

Haupt-Thread

```
main(){
  ...
  ...
}
```

2. Thread

```
thr(){
  ...
  ...
}
```

nach **pthread_create()**

Prozess

Speicher
Dateien
...

```
main(){
  ...
  ...
}
```

Thread

Speicher
Dateien
...

```
thr(){
  ...
  ...
}
```


Windows vs. Linux

- Unterschied zwischen Windows und Linux bzgl. Threads/Prozessen eher theroretisch, denn:
- Linux behandelt Threads und Prozesse intern gleich (alles ist ein „task“), also:
  - Non-threaded-Prozess unter Linux entspricht Windows-Prozess mit einem (einzigen) Haupt-Thread
  - Threaded-Prozess unter Linux mit *n* Threads entspricht Windows-Prozess mit *n*+1 Threads (Haupt-Thread plus *n* weitere Threads)

Fibers

- Fibers sind spezielle User-Level-Threads (innerhalb eines Threads)
- werden nicht vom Scheduler bedient
- Einsatz:
  - Vor erster Fiber-Nutzung: **ConvertThreadToFiber()**
  - Weitere Fibers erzeugen: **CreateFiber()**
  - Wechsel zu einem anderen Fiber: **SwitchToFiber()**
- Beispiel-Code:
  http://msdn.microsoft.com/library/en-us/dllproc/base/using_fibers.asp

Prozess-Erzeugung (1)

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

void _tmain( VOID ) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    LPTSTR szCmdline=_tcsdup(TEXT("MyChildProcess"));

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        szCmdline,      // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    ) {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Quelle: http://msdn.microsoft.com/library/en-us/dllproc/base/creating_processes.asp

Prozess-Erzeugung (2)

```
typedef struct
_PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION,
 *LPPROCESS_INFORMATION;
```

hProcess A handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread A handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId A value that can be used to identify a process. The value is valid from the time the process is created until all handles to the process are closed and the process object is freed; at this point, the identifier may be reused.

dwThreadId A value that can be used to identify a thread. The value is valid from the time the thread is created until all handles to the thread are closed and the thread object is freed; at this point, the identifier may be reused.

Quelle: http://msdn.microsoft.com/library/en-us/dllproc/base/process_information_str.asp

Thread-Erzeugung (1)

- ähnlich wie mit POSIX-Threads:
  - Angabe einer Funktion, die im Thread ausgeführt werden soll
  - neuer Thread verwendet gleichen Adressraum wie der Prozess
  - Warten auf das Ende von Threads möglich

Thread-Erzeugung (2)

```
#include <windows.h>
#include <strsafe.h>
#define MAX_THREADS 3
#define BUF_SIZE 255
typedef struct _MyData { int val1; int val2; } MYDATA, *PMYDATA;

DWORD WINAPI ThreadProc( LPVOID lpParam ) {
    HANDLE hStdout;
    PMYDATA pData;
    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.
    pData = (PMYDATA)lpParam;

    // Print the parameter values using thread-safe functions.
    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n"), pData->val1, pData->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars, NULL);

    // Free the memory allocated by the caller for the thread data structure.
    HeapFree(GetProcessHeap(), 0, pData);

    return 0;
}
```


Thread-Erzeugung (3)

```
void main() {
    PMYDATA pData;
    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    int i;
    // Create MAX_THREADS worker threads.
    for( i=0; i<MAX_THREADS; i++ ) {
        // Allocate memory for thread data.
        pData = (PMYDATA) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(MYDATA));
        if( pData == NULL ) ExitProcess(2);
        // Generate unique data for each thread.
        pData->val1 = i;
        pData->val2 = i+100;

        hThread[i] = CreateThread(
            NULL,              // default security attributes
            0,                 // use default stack size
            ThreadProc,        // thread function
            pData,             // argument to thread function
            0,                 // use default creation flags
            &dwThreadId[i]);   // returns the thread identifier

        // Check the return value for success.
        if (hThread[i] == NULL) { ExitProcess(i); }
    }
    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);
    // Close all thread handles upon completion.
    for(i=0; i<MAX_THREADS; i++) { CloseHandle(hThread[i]); }
}
```

Quelle: http://msdn.microsoft.com/library/en-us/dllproc/base/creating_threads.asp

Thread-Synchronisation

Threads eines einzigen Prozesses synchronis.	Mehrere Prozesse synchronisieren

• **Kritische Abschnitte**
`EnterCriticalSection();`
`LeaveCriticalSection();`
	• **Mutexe**
	`hMutex = CreateMutex(...);`
	`WaitForSingleObject(hMutex,...);`
	`ReleaseMutex(hMutex);`
	• **Semaphore**
	`hSem = CreateSemaphore(...);`
	`WaitForSingleObject(hSem,...);`
	`ReleaseSemaphore(hSem);`
	• **Events**
	`hEvent = CreateEvent(...);`
	`WaitForSingleObject(hEvent);`
	`SetEvent(hEvent);`
	`ResetEvent(hEvent);`

Kritische Abschnitte (1)

- Kritische Abschnitte sind nur von den Threads **eines** Prozesses nutzbar
- nur ein Thread kann gleichzeitig in den krit. Abschnitt (gegenseitiger Ausschluss)
- werden initialisiert, aber haben kein Handle
- Rekursiv: Thread darf mehrmals **EnterCriticalSection** aufrufen, muss aber beim Verlassen genauso oft **LeaveCriticalSection** aufrufen
- Nicht möglich herauszufinden, ob bereits ein Thread im krit. Abschnitt ist

Kritische Abschnitte (2)

- Datentyp: LPCRITICAL_SECTION
- Erzeugen und Zerstören:

```
InitializeCriticalSection( LPCRITICAL_SECTION sec );
DeleteCriticalSection( LPCRITICAL_SECTION sec );
```

- Betreten:

```
IEnterCriticalSection( LPCRITICAL_SECTION sec );
bool TryEnterCriticalSection ( LPCRITICAL_SECTION sec );
```

- Verlassen:

```
LeaveCriticalSection( LPCRITICAL_SECTION sec );
```


Kritische Abschnitte (3)

```
// Global variable
CRITICAL_SECTION CriticalSection;

void main() {
    ...
    // Initialize the critical section one time only.
    if (!InitializeCriticalSection(&CriticalSection)) return;
    ...
    // Release resources used by the critical section object.
    DeleteCriticalSection(&CriticalSection)
}

DWORD WINAPI ThreadProc( LPVOID lpParameter ) {
    ...
    // Request ownership of the critical section.
    EnterCriticalSection(&CriticalSection);
    // Access the shared resource.
    // Release ownership of the critical section.
    LeaveCriticalSection(&CriticalSection);
    ...
}
```


Synchronisationsobjekte (1)

- Windows-Kernel kennt Synchronisationsobjekte, die in einem von zwei Zuständen sein können:
  - *Signalled* State
  - *Unsignalled* State
- Thread kann sich mit Hilfe der Wait Services des Objektmanagers mit einem oder mehreren Objekt(en) synchronisieren:
  - Der Thread blockiert, bis das Synchr.-Objekt in den Signalled State übergeht
  - Wenn der Kernel ein Synchr.-Objekt auf *Signalled* setzt, macht er die Threads lauffähig, die darauf warten

Synchronisationsobjekte (2)

- Das Win32-API stellt für den Zugang zu den Wait Services zwei Funktionen zur Verfügung:
  - **WaitForSingleObject()**
  - **WaitForMultipleObjects()**
    - Angabe, ob auf eines oder alle Objekte gewartet werden soll (atomar!)
  - Bei beiden Funktionen ist die Angabe eines Timeout möglich.

Synchronisationsobjekte (3)

Übersicht

Objekttyp	Signalled State, wenn	Geweckt werden
Prozess	der Prozess beendet wird	alle Threads
Thread	der Thread beendet wird	alle Threads
Datei	eine I/O-Operation für diese Datei beendet wird	alle Threads
Mutex	der Mutex keinem Thread gehört	ein Thread
Semaphor	der Semaphorenwert größer 0 ist	mehrere Threads
Event	ein Thread das Event explizit setzt	abh. vom Event-Typ
Timer	der bestimmte Zeitpunkt erreicht ist	abh. vom Timer-Typ

Mutexe (1)

- benannte / unbenannte Mutexe für gegenseitigen Ausschluss; Typ: *HANDLE*
- für Threads innerhalb eines Prozesses oder für mehrere Prozesse (benannte Mutexe)
- Mutex-Operationen
  - erzeugen: **CreateMutex()**
  - erwerben: **WaitForSingleObject()**
  - freigeben: **ReleaseMutex()**

Mutexe (2)

- Mutex-Handle kann bei **CreateProcess()** an Sohnprozess vererbt werden
- Wenn mehrere Prozesse getrennt mit **CreateMutex()** den gleichen benannten Mutex erzeugen, führt
  - der erste Aufruf zum Erzeugen,
  - der zweite zur gemeinsamen Verwendung
- Alle Threads eines Prozesses haben Zugriff auf den Mutex

Mutexe (3)

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

HANDLE hMutex;

// Create a mutex with no initial owner.
hMutex = CreateMutex(
    NULL,     // default security attributes
    FALSE,    // initially not owned
    NULL);    // unnamed mutex

if (hMutex == NULL) { printf("CreateMutex error: %d\n", GetLastError()); }
```

http://msdn.microsoft.com/library/en-us/dllproc/base/using_mutex_objects.asp

Mutexe (4)

```
BOOL FunctionToWriteToDatabase(HANDLE hMutex) {
    DWORD dwWaitResult;
    // Request ownership of mutex
    dwWaitResult = WaitForSingleObject(
        hMutex,   // handle to mutex
        5000L);   // five-second time-out interval

    switch (dwWaitResult) {
        case WAIT_OBJECT_0:  // The thread got mutex ownership
            __try     { /* Write to the database */ }
            __finally { /* Release ownership of the mutex object */
                if (! ReleaseMutex(hMutex)) { /* Deal with error */ } }
            break;
        case WAIT_TIMEOUT:   // Cannot get mutex ownership due to time-out
            return FALSE;
        case WAIT_ABANDONED: // Got ownership of the abandoned mutex object
            return FALSE;
    }
    return TRUE;
}
```


Mutexe (5)

Prozess 1

```
main () {
  HANDLE hMutex;
  LPCTSTR MutexName;

  MutexName = "MeinTestMutex";
  hMutex = CreateMutex(NULL,
             FALSE, MutexName);
  if (hMutex == NULL) {
    printf("CreateMutex error: %d\n",
    GetLastError());
  }

  while (TRUE) {
    dwWaitResult =
    WaitForSingleObject(hMutex, ...);
    ...
    /* Lesezugriff auf krit. Bereich */
    ReleaseMutex(hMutex);
  }
}
```

Prozess 2

```
main () {
  HANDLE hMutex;
  LPCTSTR MutexName;

  MutexName = "MeinTestMutex";
  hMutex = CreateMutex(NULL,
             FALSE, MutexName);
  if (hMutex == NULL) {
    printf("CreateMutex error: %d\n",
    GetLastError());
  }

  while (TRUE) {
    dwWaitResult =
    WaitForSingleObject(hMutex, ...);
    ...
    /* Schreibzugriff auf krit. Bereich */
    ReleaseMutex(hMutex);
  }
}
```


Mutexe (6)

Endet ein Thread, ohne einen ihm gehörenden Mutex freizugeben,

- wird der Mutex als „verlassen“ (abandoned) betrachtet,
- wird ein Thread, der auf diesen Mutex wartet, geweckt,
- wird der geweckte Thread durch den Return-Status darüber informiert, dass der Mutex „verlassen“ ist.

Mutexe (7)

Vergleich Linux- und Windows-Mutexe

Posix-Thread-Mutexe	Windows-Mutexe
pthread_mutex_init()	CreateMutex()
pthread_mutex_destroy()	CloseHandle()
pthread_mutex_lock()	WaitForSingleObject()
pthread_mutex_unlock()	ReleaseMutex()
pthread_mutex_trylock()	WaitForSingleObject(timeout=0)

Semaphore (1)

Ähnliche Eigenschaften wie Mutexe:

- benannte / unbenannte Semaphore;
  Typ: *HANDLE*
- für Threads innerhalb eines Prozesses oder für mehrere Prozesse (benannte Semaphore)
- Semaphor-Operationen
  - erzeugen: **CreateSemaphore()**
  - Wait-Operation: **WaitForSingleObject()**
  - Signal-Operation: **ReleaseSemaphore()**

Semaphore (2)

```
HANDLE hSemaphore;
LONG cMax = 10;

// Create a semaphore with initial and max. counts of 10.

hSemaphore = CreateSemaphore(
    NULL,   // default security attributes
    cMax,   // initial count
    cMax,   // maximum count
    NULL);  // unnamed semaphore

if (hSemaphore == NULL)
{
    printf("CreateSemaphore error: %d\n", GetLastError());
}
```

http://msdn.microsoft.com/library/en-us/dllproc/base/using_semaphore_objects.asp

Semaphore (3)

```
DWORD dwWaitResult;
// Try to enter the semaphore gate
dwWaitResult = WaitForSingleObject(
        hSemaphore,   // handle to semaphore
        0L);          // zero-second time-out interval
switch (dwWaitResult) {
    case WAIT_OBJECT_0:  // The semaphore object was signaled
        // OK to open another window
        break;
    case WAIT_TIMEOUT:   // Semaphore was nonsignaled; a time-out occurred
        // Cannot open another window
        break;
}
...

// Increment the count of the semaphore
if (!ReleaseSemaphore(
        hSemaphore,  // handle to semaphore
        1,           // increase count by one
        NULL) )      // not interested in previous count
{ printf("ReleaseSemaphore error: %d\n", GetLastError()); }
```


Semaphore (4)

Consumer-Producer-Problem mit benannten Semaphoren

```
/* consumer.c */

#define NUM_LOOPS 20 /* Anzahl Durchläufe    */

int main() {
  int i;             /* Counter für Schleife */
  LPCTSTR SemName;   /* Name des Semaphors   */
  HANDLE hSem;       /* Semaphor             */
  /* benannten Sem. erzeugen; Init 0, Max. 1 */
  SemName = "ProducerConsumerSem";
  hSem = CreateSemaphore(NULL,0,1,SemName);

  for (i=0; i<NUM_LOOPS; i++) {
    /* blockieren, es sei denn, Counter>0    */
    WaitForSingleObject(hSem);
    printf("consumer: '%d'\n", i);
  }
  return 0;
}
```

```
/* producer.c */

#define NUM_LOOPS 20

int main() {
  int i;
  LPCTSTR SemName;
  HANDLE hSem;

  SemName = "ProducerConsumerSem";
  hSem = CreateSemaphore(NULL,0,1,SemName);

  for (i=0; i<NUM_LOOPS; i++) {
    printf("consumer: '%d'\n", i);
    /* Semaphor-Counter erhöhen */
    ReleaseSemaphore(hSem);
  }
  return 0;
}
```

Semaphore (5)

Vergleich Linux- und Windows-Semaphore

Posix-Thread-Semaphore	Windows-Semaphore
sem_init()	CreateSemaphore()
sem_wait()	WaitForSingleObject()
sem_trywait()	WaitForSingleObject(timeout=0)
sem_post()	ReleaseSemaphore()
sem_destroy()	CloseHandle()

Events (1)

- Threads über Ereignisse informieren
- Ansatz ist hier eher: Zu bestimmten Anlässen Funktionen ausführen (statt dies zu verhindern)
  - Warten auf Event ist Normalzustand des Threads (anders als bei Mutex, Semaphore etc.)
- Event erzeugen mit **CreateEvent()**

```
HANDLE WINAPI CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,     /* TRUE, FALSE */
  BOOL bInitialState,    /* TRUE: signalisiert, FALSE */
  LPCTSTR lpName         /* Name für Event */
);
```


Events (2)

- automatischer Event:
  - jeder Aufruf von **WaitForSingleResource**(event) setzt den Event automatisch zurück (reset).
  - falls mehrere Threads warten, kehrt bei einem **SetEvent**() nur genau ein Thread von seinem **Wait**...()-Aufruf zurück.
- manueller Event:
  - Event muss manuell zurückgesetzt werden: **ResetEvent**().
  - falls mehrere Threads warten, kehren bei **SetEvent**() alle wartenden Threads aus ihren **Wait**...()-Aufrufen zurück.

Events (3)

Beispiel:
- Event 1: Thread hat Daten erzeugt
- Event 2: Hauptprogramm hat Daten ausgelesen

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

HANDLE hEvent1, hEvent2;
int a[5];

void Thread( void* pParams ) {
  int i, num = 0;

  while (TRUE) {
    WaitForSingleObject(hEvent2,INFINITE);
    for ( i = 0; i < 5; i++ ) a[i] = num;
    SetEvent(hEvent1);
    num++;
  }
}
```

```
int main( void ) {
  hEvent1 = CreateEvent(NULL,FALSE,TRUE,NULL);
  hEvent2 = CreateEvent(NULL,FALSE,FALSE,NULL);

  _beginthread( Thread, 0, NULL );

  while( TRUE ) {
    WaitForSingleObject(hEvent1,INFINITE);
    printf( "%d %d %d %d %d\n",
            a[ 0 ], a[ 1 ], a[ 2 ],
            a[ 3 ], a[ 4 ] );
    SetEvent(hEvent2);
  }
  return 0;
}
```

Quelle: http://www.codeproject.com/threads/sync.asp

Synchr. im Windows-Kernel

- atomare Test-and-Set-Operationen
- Spinlocks: Solange ein Lock nicht verfügbar ist, aktiv warten
- Queued Spinlocks: Spinlocks mit FIFO-Warteschlange
- seit 2003: sog. „Pushlocks“ (Windows-Name für Reader-Writer-Spinlocks)

Vorschau

Nächstes Kapitel:

6. Inter-Prozess-Kommunikation (IPC)