E Überblick über die 5. Übung

- Fragen zur Aufgabe 2?
- Organisatorisches
  - ◆ Übungsbetrieb an den Windows-Rechnern
- Sockets unter Windows
- Threads unter Windows

E.1 Sockets unter Windows

- Vorraussetzungen
- Socket Funktionen

1 Windows Sockets

- benötigte Bibliothek: ws2_32.lib
  (VS6: Projekteinstellungen -> Linker (Allgemein) -> Bibliotheksmodule)
  (VS.NET: Projekteigenschaften -> Linker : Eingabe -> zusätzliche Abhängigkeiten)
- Bibliothek initialisieren:

```
int WSAStartup( WORD wVersionRequested, LPWSDATA lpWSAData );
```

  - ◆ **wVersionRequested**: angeforderte Version
  - ◆ **lpWSAData**: Informationen über die verwendete Bibliothek
- Struktur WSADATA:

```
typedef struct WSAData {
    WORD           wVersion;
    WORD           wHighVersion;
    char           szDescription[WSADESCRIPTION_LEN+1];
    char           szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets
    unsigned short iMaxUdpDg
    char           lpVendorInfo;
} WSADATA;
```


1 Windows Sockets (2)

- von der Bibliothek abmelden

```
int WSACleanup(void);
```

  - ◆ gibt alle durch die Bibliothek belegten Ressourcen wieder frei
  - ◆ schließt alle offenen Verbindungen
- Beispiel:

```
    WORD wVersionRequested = MAKEWORD( 2, 2 );;
    WSADATA wsaData;
    int err;

    if ( WSAStartup( wVersionRequested, &wsaData ) != 0)
        return false;
    // Confirm that the WinSock DLL supports 2.2.
    if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 2 ) {
        // no usable WinSock DLL found
        WSACleanup( );
        return false;
    }
    // The WinSock DLL is acceptable. Proceed.
    return true;
```

1 Windows Sockets (3)

- Fehlerstatus abfragen:

```
int WSAGetLastError (void);
```

  - ◆ statt eine globale Variable `errno` zu setzen, muss hier der Fehlercode durch eine Funktion abgefragt werden

- Beispiel: (vom Fehlercode zur Fehlermeldung)

```
int eno = WSAGetLastError();
LPVOID lpMsgBuf = NULL;
FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
               FORMAT_MESSAGE_FROM_SYSTEM |
               FORMAT_MESSAGE_IGNORE_INSERTS,
               NULL, eno,
               MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
               (LPTSTR) &lpMsgBuf, 0, NULL
               );
// Display the string.
if (lpMsgBuf != NULL)
    cerr << msg << (char *)lpMsgBuf << endl;
// Free the buffer.
LocalFree( lpMsgBuf );
```


2 Socket Funktionen

- Headerdatei

```
#include <windows.h> // oder <winsock2.h>
```

- Erzeugen eines neuen Sockets:

```
SOCKET socket(int af, int type, int protocol);
```

- Binden von Sockets:

```
int bind (SOCKET s,
          const struct sockaddr *name, int namelen);
```

  - ◆ Socket Internetadressen wie unter UNIX: `struct sockaddr_in`

- Socket-Adresse aus Hostnamen erzeugen:

```
struct hostent *gethostbyname(const char *name);
```

  - ◆ Struktur `hostent` wie unter UNIX

2 Socket Funktionen (2)

- TCP-Server
  - ◆ Warteschlangenlänge festlegen

```
int listen(SOCKET s, int backlog);
```

  - ◆ Verbindung annehmen:

```
int accept(SOCKET s,
              struct sockaddr *addr, socklen_t *addrlen);
```

- TCP-Klient
  - ◆ Verbindung zu einem Server herstellen:

```
int connect(SOCKET sockfd,
    const struct sockaddr *name, socklen_t namelen);
```


2 Socket Funktionen (3)

- TCP
  - ◆ Lesen von Sockets

```
int recv ( SOCKET s, char *buf, int len, int flags);
```

  - ◆ Schreiben auf Sockets

```
int send (SOCKET s, const char *buf, int len, int flags);
```

- UDP
  - ◆ Lesen von Sockets

```
int recvfrom ( SOCKET s, char *buf, int len, int flags,
                      struct sockaddr *from, int *fromlen);
```

  - ◆ Schreiben auf Sockets

```
int sendto (SOCKET s, const char *buf, int len, int flags,
                      const struct sockaddr *to, int tolen);
```

2 Socket Funktionen (4)

- Schließen einer Socketverbindung

```
int closesocket(SOCKET s);
```

- Senden oder Empfangen abschalten

```
int shutdown(int s, int how);
```

- Socketoptionen

```
int setsockopt(SOCKET s, int level, int optname,
                      const char *optval, int optlen);
```


E.2 Windows Threads

1 Windows Threads Benutzerschnittstelle (2)

- Threaderzeugung

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

  - Rückgabe: Thread-Handel
  - **lpThreadAttributes**: Sicherheitsattribute (Vererbung des Handles)
  - **dwStackSize**: Stackgröße in Bytes (**NULL** =Standard)
  - Thread startet mit der Funktion **lpStartAddress** mit **lpParameter** als Parameter
  - **dwCreationFlags**: wie wird der Thrad erzeugt (z.B.: **CREATE_SUSPENDED**)
  - **lpThreadId**: Thread Identifier (!= **NULL**)

1 Windows Threads Benutzerschnittstelle (2)

- Handle zu existierendem Thread erstellen

```
HANDLE OpenThread( DWORD dwDesiredAccess,
                   BOOL bInheritHandle, DWORD dwThreadId );
```

  - **dwDesiredAccess**: Zugriffsart (z.B. **THREAD_QUERY_INFORMATION**)
  - **bInheritHandle**: soll das Handle vererbbar sein?
  - **dwThreadId**: Thread ID

- explizites beenden eines Threads:

```
void ExitThread(DWORD dwExitCode);
```

- beenden eines anderen Threads:

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

- Exit Status eines Threads abfragen:

```
BOOL GetExitCodeThread( HANDLE hThread, LPDWORD lpExitCode );
```

  - wenn der Thread noch läuft: **ExitCode = STILL_ACTIVE**

1 Windows Threads Benutzerschnittstelle (3)

- Auf Thread warten:

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

  - **hHandle**: Thread-Handle
  - **dwMilliseconds**: Timeout

- Fehlerstatus abfragen:

```
int GetLastError (void);
```

  - (vgl. **WSAGetLastError)**

- Handle freigeben:

```
BOOL CloseHandle (HANDLE hObject);
```

2 Thread-Koordinierung

- Mutex erzeugen

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes,
                   BOOL bInitialOwner, LPCTSTR lpName );
```

  - ◆ **lpMutexAttributes**: Sicherheitsattribute (Vererbung des Handles)
  - ◆ **bInitialOwner**: soll der Mutex gleich belegt sein
  - ◆ **lpName**: Name des Mutexes

- existierendes Mutex öffnen

```
HANDLE OpenMutex( DWORD dwDesiredAccess,
                  BOOL bInheritHandle, LPCTSTR lpName );
```

  - ◆ **dwDesiredAccess**:Zugriffsart
  - ◆ **bInheritHandle**: soll das Handle vererbbar sein?
  - ◆ **lpName**: Name des Mutexes

2 Thread-Koordinierung (2)

- Lock

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

  - ◆ **hHandle**: Mutex-Handle
  - ◆ **dwMilliseconds**: Timeout (**INFINITE** = kein Timeout)

- Unlock

```
BOOL ReleaseMutex( HANDLE hMutex );
```


2 Thread-Koordinierung (3)

- Beispiel:

```
HANDLE hMutex = CreateMutex( NULL, FALSE, "myMutex");
if (hMutex == NULL) { /* error */}

...

// Mutex anfordern
DWORD dwWaitResult = WaitForSingleObject( hMutex, 5000L);
switch (dwWaitResult) {
        // The thread got mutex ownership.
        case WAIT_OBJECT_0:
            // kritischer Abschnitt
            if (! ReleaseMutex(hMutex)) { /* error */ }
            break;
        case WAIT_TIMEOUT: // timeout
             return FALSE;
        case WAIT_ABANDONED: // abandoned mutex
             return FALSE;
};
```


3 Zusammenfassung

- Windows Sockets
  - ◆ Bibliothek initialisieren
  - ◆ Aufrufe analog zu POSIX-Standard
  - ◆ Fehlercode mittels **WSAGetLastError**
  - ◆ Abmelden von der Bibliothek
- Windows Threads
  - ◆ Threaderzeugung mittels **CreateThread**
  - ◆ Mutexes zur Koordination