

# Übungsblatt 3

Übung zur Betriebssystemarchitektur  
SS 2005

Dipl.-Inf. Bernhard Rabe  
Betriebssysteme & Middleware



---

# Inhalt

---



- ◆ Fehlerbehandlung
- ◆ Prozesse
- ◆ Threads
- ◆ Synchronisation

# Fehlerbehandlung

- ◆ API-Ruf konnte nicht „erfolgreich“ ausgeführt werden
- ◆ Rückgabewert zeigt Status bzw. Fehlercode an
- ◆ Ermitteln der Fehlerbeschreibung aus dem Fehlercode

# Windows

## ◆ Fehlerindikator unterschiedlich

### ■ BOOL → Null bzw. FALSE

- CreateProcess(..)

### ■ HANDLE → NULL, INVALID\_HANDLE\_VALUE

- CreateThread(..) NULL
- CreateFile(..) INVALID\_HANDLE\_VALUE

### ■ DWORD → MSDN

- WaitForSingleObject(..) Status bzw. WAIT\_FAILED

# Windows

- ◆ **DWORD GetLastError(void)**
  - liefert den Fehlercode des letzten Win32 API Rufs durch den rufenden Thread, der einen Fehler liefert → MSDN
- ◆ **DWORD FormatMessage( .. )**
  - Formatiert eine Nachricht, z.B. aus der Systemtabelle → Fehlercode
- ◆ Funktioniert nur für Win32 API !

# C-Bibliothek

- ◆ LibC Rufe liefern in der Regel  $<0$  im Fehlerfall
  - `open()`, `fork()`
  - `malloc` → `NULL`
- ◆ Fehlercode des letzten Rufs ist in der Variable **`errno`** enthalten (`errno.h`)
  - POSIX API weicht teilweise von diesem Verhalten ab
- ◆ **`void perror( const char *string )`**
  - "string: Fehlermeldung" nach `stderr`
  - "Fehlermeldung" wenn `string==NULL`

# C-Bibliothek

- ◆ **char \*strerror( int *errnum* )**
  - Liefert einen Pointer auf die Systemfehlermeldung zu **errnum** zurück
  - perror Verhalten

```
#include <errno.h>
if(-1 == fork() )
{

    close(f);
    perror("Error at fork");
    exit(1);
}
```

# C-Bibliothek

- ◆ **char \*strerror( int *errnum* )**
  - Liefert einen Pointer auf die Systemfehlermeldung zu **errnum** zurück
  - perror Verhalten

```
#include <errno.h>
if(-1 == fork() )
{
    int errval=errno;
    close(f);
    fprintf(stderr, "%s: %s\n", string, strerror(errval));
    exit(1);
}
```





# Prozesse

# CreateProcess

- ◆ Erzeugt einen neuen Prozess aus einer ausführbaren Datei
- ◆ **BOOL** CreateProcess( **LPCTSTR** *lpApplicationName*,  
**LPTSTR** *lpCommandLine*,  
**LPSECURITY\_ATTRIBUTES** *lpProcessAttributes*,  
**LPSECURITY\_ATTRIBUTES** *lpThreadAttributes*,  
**BOOL** *bInheritHandles*,  
**DWORD** *dwCreationFlags*, **LPVOID** *lpEnvironment*,  
**LPCTSTR** *lpCurrentDirectory*,  
**LPSTARTUPINFO** *lpStartupInfo*,  
**LPPROCESS\_INFORMATION** *lpProcessInformation*  
);

# CreateProcess

- ◆ `CreateProcess(NULL, "notepad.exe",  
NULL, NULL, FALSE, 0,  
NULL, NULL,  
&si,  
&pi  
);`
- ◆ *lpApplicationName* Programmname ohne Argumente
- ◆ *lpCommandLine* Kommandozeile (Programm + Argumente)

# STARTUPINFO

- ◆ **typedef struct** \_STARTUPINFO {  
    DWORD cb; //Groese in Byte  
    ...  
    HANDLE hStdInput;  
    HANDLE hStdOutput;  
    HANDLE hStdError;  
} STARTUPINFO;
- ◆ STARTUPINFO si;

# STARTUPINFO

- ◆ Initialisierung

1. `ZeroMemory(&si, sizeof(si));`  
`si.cb = sizeof(STARTUPINFO);`

2. `GetStartupInfo(&si);`

- Einstellungen des aktuellen Prozesses übernehmen

# PROCESS\_INFORMATION

- ◆ **typedef struct**

```
PROCESS_INFORMATION {  
HANDLE hProcess;  
HANDLE hThread;  
DWORD dwProcessId;  
DWORD dwThreadId; }  
PROCESS_INFORMATION;
```

- ◆ `PROCESS_INFORMATION pi;`

# PROCESS\_INFORMATION

## ◆ Verwendung

### ■ Initialisieren

- `ZeroMemory(&pi, sizeof(pi));`

```
CreateProcess( . . . , &pi );
```

```
. . .
```

```
WaitForSingleObject( pi.hProcess, INFINITE );
```

# HANDLE

- ◆ **BOOL** CloseHandle( **HANDLE** *hObject* );
  - Nicht mehr benötigte Handle's schließen



# fork

- ◆ Erzeugt eine Kopie\* des aktuellen Prozesses
  - \*Copy-On-Write für Speicherseiten
  - neue PID, PPID des Eltern-Prozesses im Kind
- ◆ `pid_t fork( ) ;`
  - Rückgabecode == -1 Elternprozess, Fehler
  - Rückgabecode == 0 Kindprozess
  - Rückgabecode > 0 Elternprozess, Prozess ID des Kinds

# fork

```
pid_t pid;
int status;
pid=fork();
if(pid<0){ perror("Fehler bei fork\n"); exit(1); }

if(pid==0)
{
    printf("I'm the child: %d of %d\n",getpid(),getppid());
    exit(42);
}
printf("I'm the parent: %d of %d\n",getpid(),pid);
if(waitpid(pid,&status,0)<0){ perror("waitpid"); exit(1);}
```

# fork

- ◆ `waitpid(pid_t pid, int* status, int options);`
  - liefert den Status des Prozess pid
  - `options=0` wird auf Beendigung gewartet
- ◆ Makros zur Auswertung des Status
  - `WIFEXITED(status)`
  - `WEXITSTATUS(status)`

# exec\*

- ◆ Ersetzt den aktuellen Prozess durch neu zu Startenden
- ◆ Varianten von exec
  - `int execl(const char *path, const char *arg, ...);`
  - `int execlp(const char *file, const char *arg, ...);`
  - `int execlp(const char *file, const char *arg, ..., char * const envp[]);`
  - `int execv(const char *path, char *const argv[]);`
  - `int execvp(const char *file, char *const argv[]);`
- ◆ Aliase für `execve(const char *filename, char *const argv [], char *const envp[]);`

# execl

- ◆ 1. Parameter Pfad zum Programm
- ◆ 2.- n-1 Parameter Argumente
  - 1. Argument ist per Konvention der Programmname
  - Liste muss mit Null Pointer terminiert werden
- ◆ `execl ( " /bin/ls " , "ls" , 0 ) ;`
- ◆ kehrt im Normalfall nicht zurück

fexec.c

# execv

- ◆ Erwartet die Argumente als Nullterminiertes char Array

```
char *args[] = { "ls", 0 };  
execv( "/bin/ls", args );
```

- ◆ Übergabe von Environment Variables in  
`execle( "/bin/ls", args, envargs );` analog

# Threads

- ◆ Leichtgewichtprozesse
  - laufen im Kontext eines Prozesses
  - Abbildung von Usermode-Threads auf Kernel-Threads ist Betriebssystemabhängig
  - teilen sich den Speicher des Prozesses
    - globale Variablen
  - haben eigenen Stack

# Threads/Win32

- ◆ HANDLE CreateThread(  
LPSECURITY\_ATTRIBUTES *lpThreadAttributes*,  
SIZE\_T *dwStackSize*,  
LPTHREAD\_START\_ROUTINE *lpStartAddress*,  
LPVOID *lpParameter*,  
DWORD *dwCreationFlags*,  
LPDWORD *lpThreadId*  
);



# Threads

- ◆ *dwCreationFlags* = CREATE\_SUSPENDED
  - erzeugt den Thread im Suspend Modus
- ◆ ResumeThread ( ) dekrementiert den Suspend Zähler und startet den Thread wenn gleich 0
- ◆ SuspendThread ( ) Aussetzen der Ausführung und Erhöhen des Suspend Zähler
  - nur für Debugging (Deadlock-Gefahr)

# Threads-Routine

```
DWORD WINAPI ThreadFunc(LPVOID data)
{
    int val=*(int*)data;
    printf("%d\n",val);
    return 0;
}

/* Unvollständig */
int a=42; DWORD id;
HANDLE hThread;
hThread=CreateThread(NULL,0,ThreadFunc,&a,
    /*CREATE_SUSPENDED*/ 0, &id);
WaitForSingleObject(hThread,INFINITE);
```

# Sleep()

- ◆ **VOID Sleep( DWORD *dwMilliseconds* );**
- ◆ setzt die Ausführung für den Rest des Quantums und mindst. *dwMilliseconds* Millisekunden aus, um dann in den ready Zustand zurück zu kehren
- ◆ *dwMilliseconds=0* restliches Quantum aufgeben

# Thread Beenden

- ◆ **VOID ExitThread( DWORD *dwExitCode* );**
  - beendet den aktuellen Thread mit *dwExitCode*
- ◆ **BOOL GetExitCodeThread( HANDLE *hThread*, LPDWORD *lpExitCode* );**
- ◆ **BOOL TerminateThread( HANDLE *hThread*, DWORD *dwExitCode* );**
  - nur in Ausnahmefällen benutzen

# POSIX Pthread-API

- ◆ ANSI/IEEE POSIX 1003.1c - 1995
- ◆ `#include <pthread.h>`
- ◆ Bibliothek `-lpthread`
- ◆ alle Bibliotheksfunktionen beginnen mit `pthread_`
- ◆ Threads, Mutex, Condition Variables

# Pthread API

```
int pthread_create(pthread_t * thread,  
                  pthread_attr_t * attr,  
                  void * (*start_routine)(void *),  
                  void * arg);
```

- ◆ thread: Thread ID
- ◆ attr: Attribute für neuen Thread
- ◆ start\_routine: Funktionspointer
- ◆ args: Übergabeparameter
- ◆ Rückgabewert != 0 Fehlercode
- ◆ benutzt nicht **errno** !!!

# Pthread API

- ◆ `void pthread_exit(void *retval);`
  - beendet aufrufenden Thread mit Rückgabewert `*retval`
- ◆ `int pthread_join(pthread_t th, void **thread_return);`
  - wartet auf das Ende des Threads `th`
  - und liefert den Rückgabewert
  - muss sich im *joinable* Status befinden

# Pthread API

- ◆ Thread Attribute
- ◆ `pthread_attr_init(pthread_attr_t *attr);`
  - Initialisiert *attr* mit Standardwerten
- ◆ `pthread_attr_get...`
  - `detachedstate();` z.B. `PTHREAD_CREATE_JOINABLE`
  - Werte von Attributen
- ◆ `pthread_attr_set...`
  - `detachedstate();` z.B. `PTHREAD_CREATE_DETACHED`
  - Setzen von Attributen
- ◆ `pthread_attr_destroy();`
  - gibt Attribute Objekt wieder frei
- ◆ *attr* wird durch `pthread_create();` nicht verändert



# Pthread Beispiel

```
void* ThreadFunc(void *data)
{
    int val=*(int*)data;
    printf("Thread %d\n",val);
    pthread_exit((void*)0);
}

pthread_t thread_id;
if(0!=pthread_create(&thread_id,NULL,ThreadFunc,&val)
)
    { fprintf(stderr,"pthread_create\n"); exit(1);}
if(0!=pthread_join(thread_id,NULL))
    { fprintf(stderr,"pthread_join\n"); exit(2); }
```

# Synchronisation

- ◆ Mutex
  - erlaubt den exklusiven Zugriff
- ◆ Semaphore
  - erlaubt die Begrenzung des gleichzeitigen Zugriffs
- ◆ Events
  - Benachrichtigung mehreren wartenden Threads

# Deadlock



# Deadlock

- ◆ Unauflösbare Verklemmung zweier Tasks, die beim Wettstreit (Race Condition) um eine Ressource in einen Zustand geraten, in dem sie sich gegenseitig am Weiterkommen hindern.

# Synchronisation Win32

- ◆ CreateXXX
  - erzeugt|öffnet Synchronisationsobject
- ◆ OpenXXX
  - Prozessübergreifend über Namen erreichbar
- ◆ ReleaseXXX
  - gibt Objekt wieder frei

# Mutex Win32

- ◆ gegenseitiger Ausschluß
- ◆ kann nur von einem Thread gehalten werden
- ◆ FIFO Warteschlange für wartende Threads (ist nicht garantiert: APC)
- ◆ `CreateMutex()`
- ◆ `WaitForSingleObject()`
- ◆ `OpenMutex()`
- ◆ `ReleaseMutex()`
- ◆ Prozessübergreifend

# CreateMutex

- ◆ HANDLE CreateMutex(  
    LPSECURITY\_ATTRIBUTES *lpMutexAttributes*,  
    BOOL *bInitialOwner*,  
    LPCTSTR *lpName*);
- ◆ Name darf nicht für Event, Semaphore, Waitable Timer, Job oder File-mapping Objekt vergeben sein
- ◆ *lpName*=NULL für unbenanntes Mutex

# Zugriff

## ◆ mit Wait-Funktionen

- `DWORD WaitForSingleObject( HANDLE hHandle, DWORD dwMilliseconds );`
- `DWORD WaitForMultipleObjects( DWORD nCount, const HANDLE* lpHandles, BOOL bWaitAll, DWORD dwMilliseconds );`

## ◆ `dwMilliseconds=0` für Test auf Verfügbarkeit



# Critical Section

- ◆ Verhalten wie Mutex
- ◆ Nur innerhalb eines Prozesses verfügbar
- ◆ in C Stucturierte Fehlerbehandlung
- ◆ `InitializeCriticalSection()`
- ◆ `EnterCriticalSection()`
- ◆ `TryEnterCriticalSection()`
- ◆ `LeaveCriticalSection()`
- ◆ `DeleteCriticalSection()`

# Event

- ◆ Ereignis kann (bei Erzeugung) signalisiert werden
- ◆ Benachrichtigung mehrerer Threads gleichzeitig
- ◆ kann nach jeder Signalisierung (Thread) automatisch zurückgesetzt werden
- ◆ `CreateEvent`
- ◆ `OpenEvent`
- ◆ `SetEvent`
- ◆ `ResetEvent`
- ◆ `WaitFor...Object`

# CreateEvent

- ◆ `hEvent=CreateEvent(NULL, TRUE, FALSE, NULL);`
  - erzeugt Event, welches nach der Auslösung eines wartenden Thread **nicht zurückgesetzt** wird!
  - nach der Erzeugung nicht signalisiert ist
- ◆ `SetEvent(hEvent);`
  - weckt alle wartenden Threads

# Semaphore

- ◆ Erlaubt begrenzten Zugriff
- ◆ **HANDLE** **CreateSemaphore**(  
**LPSECURITY\_ATTRIBUTES** *lpSemaphoreAttributes*,  
**LONG** *lInitialCount*, **LONG** *lMaximumCount*,  
**LPCTSTR** *lpName* );
- ◆ *InitialCount*: verfügbare Zugriffe
- ◆ *lMaximumCount*: maximal verfügbare Zugriffe
- ◆ Wartefunktionen dekrementieren *InitialCount* wenn  $>0$
- ◆ **ReleaseSemaphore** erhöht *InitialCount* um einen angegebenen Wert  $>0$

# Pthread Mutex

- ◆ `int pthread_mutex_init();`
  - erzeugt Mutex
- ◆ `int pthread_mutex_lock();`
  - aquiriert Mutex blockierend
- ◆ `int pthread_mutex_trylock();`
  - aquiriert Mutex wenn frei
- ◆ `int pthread_mutex_unlock();`
  - gibt Mutex frei
- ◆ `int pthread_mutex_destroy();`
  - zerstört Mutex Objekt nur im Status unlocked

# Pthread Mutex

- ◆ `int pthread_mutex_init(  
pthread_mutex_t *var,  
const pthread_mutexattr_t *attr  
);`
- ◆ `attr`: Mutex Attribute
  - `pthread_mutexattr_init()`
- ◆ statische Initialisierung mit Makro
  - `PTHREAD_MUTEX_INITIALIZER`

# Pthread Mutex Beispiel

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER; //statisch
```

```
void* run(void* data)
{
    if(0!=pthread_mutex_lock(&mutex)) { /*Fehler*/}
    /* Zugriff auf geschützte Ressource */
    if(0!=pthread_mutex_unlock(&mutex)) { /*Fehler*/}
    pthread_exit((void*)0);
}
```

```
pthread_t thread1, thread2;
if(0!=pthread_mutex_init(&mutex,NULL)) { /*Fehler*/ }
for(l=0;l<i;l++)
if(0!=pthread_create(&thread1,0,run,&ids[l])) { /*Fehler*/ }
```

# POSIX Semaphore

- ◆ eingeschränkter Zugriff auf eine Ressource
- ◆ `#include <semaphore.h>`
- ◆ `int sem_init(sem_t *sem , int, unsigned int value)`
  - erzeugt Semaphore Variable sem, mit Zählerstand value
- ◆ `int sem_wait(sem_t *sem);`
  - wartet blockierend Zugriffserteilung, dekrementiert Zähler
- ◆ `int sem_trywait(sem_t *sem);`
  - aquiriert Semaphore wenn möglich, liefert sonst EAGAIN
- ◆ `int sem_post(sem_t *sem);`
  - Inkrementiert Zähler um 1, (Release)
- ◆ `int sem_getvalue(sem_t *sem, int *value);`
  - liefert den aktuellen Zählerstand
- ◆ `int sem_destroy(sem_t *sem);`
  - zerstört die Semaphore



# Posix Semaphore

- ◆ Rückgabecode 0=OK, -1 Fehler
- ◆ errno wird gesetzt

# Condition Variable

- ◆ `int pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);`
  - Attribute
    - `pthread_condattr_init(pthread_condattr_t *attr);`
  - erzeugt Condition Variable
  - statische Initialisierung mit Makro `PTHREAD_COND_INITIALIZER`
- ◆ `int pthread_cond_destroy(pthread_cond_t *cond);`
  - zerstört Condition Variable

# Warten auf die Bedingung

- ◆ `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- ◆ `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- ◆ Atomares freigeben des Mutex und Warten auf die Bedingung
- ◆ Mutex muss wieder freigegeben werden

# Signalisieren der Bedingung

- ◆ `int pthread_cond_signal (pthread_cond_t *cond);`
  - weckt genau einen wartenden Thread auf
- ◆ `int pthread_cond_broadcast (pthread_cond_t *cond);`
  - weckt alle wartenden Threads auf

```
//CreateEvent(NULL,FALSE,FALSE,NULL);
```

```
void *thread1(void* d)
{   pthread_mutex_lock(&mutex);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

```
void *thread2(void *d)
{   pthread_cond_wait(&cond,&mutex);
    /* hier cond signalisiert und mutex bekommen*/
    pthread_cond_signal(&cond); //anderen Thread wecken
    pthread_mutex_unlock(&mutex);
}
```

# Events mit Condition Variables

```
//CreateEvent(NULL,TRUE,FALSE,NULL);

pthread_cond_t cond=THREAD_COND_INITIALIZER;
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;

void *thread(void *d)
{
    pthread_cond_wait(&cond,&mutex);
    pthread_mutex_unlock(&mutex);
    /* */
}
main(){
    pthread_mutex_lock(&mutex);
    for(...) pthread_create(...,thread,...);
    pthread_cond_broadcast(&cond); //signal event
    pthread_mutex_unlock(&mutex);
}
```

# Literatur

- ◆ <http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html>
- ◆ <http://www.opengroup.org/onlinepubs/007908799/xsh/semaphore.h.html>
- ◆ MSDN, April 2005