

Übungsblatt 3

Übung zur Betriebssystemarchitektur
WS 2004/05

Dipl.-Inf. Bernhard Rabe
Betriebssysteme & Middleware

Fehlerbehandlung

- ◆ `perror(const char* msg)`
 - schreibt msg gefolgt “: “ und Fehlerbeschreibung auf `stderr`

```
execl("blah", "blah", 0);  
perror("Fehler");  
>Fehler: No such file or directory
```
- ◆ `perror` mit `fprintf`
 - `fprintf(stderr, "Fehler: %s", strerror(2));`



Inhalt

- ◆ Thread Erzeugung Win32/Linux
- ◆ Synchronisation
- ◆ Übungsaufgabe 3.4 Fraktal

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

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(hThread);`
 - 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

WaitableTimer

- ◆ Erzeugt ein zeitbestimmtes Ereignis
- ◆ `CreateWaitableTimer()`
- ◆ `SetWaitableTimer()` startet Timer
- ◆ Timer wird nach abgelaufener Zeit signalisiert (wie Event)
- ◆ `CancelWaitableTimer()` deaktiviert Timer
- ◆ Prozessübergreifend

Posix Threads-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

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); }
```

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()`
- ◆ `var` Mutex-Variable: statische Initialisierung
 - `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*/ }
```

pthread_prime.c

Posix Semaphore

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

Posix Semaphore

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

Read/Write Lock

- ◆ beliebig viele Read Locks gleichzeitig, aber kein Write Lock
- ◆ nur ein Write Lock und keine Read Locks
- ◆ Read Lock Zähler
 - gleiche Anzahl von Locks /Unlocks notwendig

Read/Write Lock

- ◆ `int pthread_rwlock_init(pthread_rwlock_t *, const pthread_rwlockattr_t *);`
- ◆ `int pthread_rwlock_rdlock(pthread_rwlock_t *);`
- ◆ `int pthread_rwlock_tryrdlock(pthread_rwlock_t *);`
- ◆ `int pthread_rwlock_wrlock(pthread_rwlock_t *);`
- ◆ `int pthread_rwlock_trywrlock(pthread_rwlock_t *);`
- ◆ `int pthread_rwlock_unlock(pthread_rwlock_t *);`
- ◆ `int pthread_rwlockattr_destroy(pthread_rwlockattr_t *);`

mandelbrot.c

- ◆ berechnet eine 500x500 Pixel großes Fraktal
- ◆ Ausgabe der Pixelwert auf die Standardausgabe
- ◆ Ausgabe in Blöcken
 - XSTART
 - YSTART
 - WIDTH
 - HEIGHT
 - <HEIGHT Zeilen der Länge WIDTH >
 - R
 - G
 - B

mandelbrot.c

- ◆ Ausgabe eines 20x20 Pixel großen roten Rechtecks beginnend ab 250,250

250

250

20

20

255

0

0

399 mal wiederholen

mandeltest.c

Aufgabe 3.4

- ◆ Erweiterung auf 2 Arbeits-Thread
 - Verteilung der Blöcke auf die Threads
 - Berechnung der Blöcke dauert unterschiedlich lange
 - rufen getMandelbrotPixelAt()
 - Ergebnisse sollen synchronisiert, gültige Blöcke auf die Standardausgabe ausgeben
 - Fortschritt der Berechnung soll am Fraktalviewer in „Echtzeit“ verfolgt werden können

Abgabeformat

- ◆ Zip-Datei
 - keine Unterverzeichnisse
 - Makefile
 - target aufg34.exe erzeugt aufg34.exe unter Win32
 - target aufg34 erzeugt aufg34 unter Linux
- ◆ Abgabe 24 Stunden werktags vor dem Tutoriumstermin (Termin Montag heißt Abgabe Freitag)

Literatur

- ♦ Java ist auch eine Insel, Christian Ullenboom
Galileo Computing / <openbook>
- ♦ <http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html>