I 7. Übung

I-1 Überblick

- Besprechung Aufgabe 5 (mysh)
- Online-Evaluation
- Posix Threads

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-2 Evaluation

- Online-Evaluation von Vorlesung und Übung SOS
- zwei TANs, zwei Fragebogen
  - ➤ TANs werden in den Tafelübungen verteilt
- Fragebogen vom 21. Juni bis 9. Juli auszufüllen
- Ergebnisse werden ab 12. Juli an Dozenten verschickt
  - ➤ Diskussion in Vorlesung und Übungen
  - ➤ Veröffentlichung auf den Web-Seiten der Lehrveranstaltung
- Ergebnisse der Evaluation vom WS stehen im Netz
- bitte unbedingt teilnehmen - das Feedback ist für die Dozenten und Organisatoren sehr wichtig (wir wollen unseren Job so gut wie möglich machen!)
  - ➤ und interessiert natürlich vor allem, was wir seit dem letzten Winter besser (oder schlechter) gemacht haben!

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-3 Motivation von Threads

- UNIX-Prozesskonzept: eine Ausführungsumgebung (virtueller Adressraum, Rechte, Priorität, ...) mit einem Aktivitätsträger (= Kontrollfluss oder Thread)
- Problem: UNIX-Prozesskonzept ist für viele heutige Anwendungen unzureichend
  - ➤ in Multiprozessorsystemen werden häufig parallele Abläufe in einem virtuellen Adreßraum benötigt
  - ➤ zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adreßraums nützlich
  - ➤ typische UNIX-Server-Implementierungen benutzen die fork-Operation, um einen Server für jeden Client zu erzeugen
    - ➥ Verbrauch unnötig vieler System-Ressourcen (Datei-Deskriptoren, Page-Table, Speicher, ...)
- Lösung: bei Bedarf weitere Threads in einem UNIX-Prozess erzeugen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-4 Vergleich von Thread-Konzepten

- ***User-Level Threads***
  - ➤ Realisierung von Threads auf Anwendungsebene innerhalb eines Prozesses
  - ➤ Systemkern sieht nur den Prozess mit einem Kontrollfluss (Thread)

  Bewertung:
  - \+ Erzeugung von Threads und Umschaltung extrem billig
  - – Systemkern hat kein Wissen über diese Threads
    - ➥ Scheduling zwischen den Threads schwierig (Verdrängung meist nicht möglich - höchstens über Signal-Handler)
    - ➥ in Multiprozessorsystemen keine parallelen Abläufe möglich
    - ➥ wird ein Thread wegen eines *page faults* oder in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-4 Vergleich von Thread-Konzepten (2)

- ***Kernel Threads***: leichtgewichtige Prozesse
  (lightweight processes)

  Bewertung:

  + eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln (= Prozess)
  + jeder Thread ist aber als eigener Aktivitätsträger dem Betriebssystemkern bekannt
  – Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei "schwergewichtigen" Prozessen, aber erheblich teuerer als bei User-level Threads

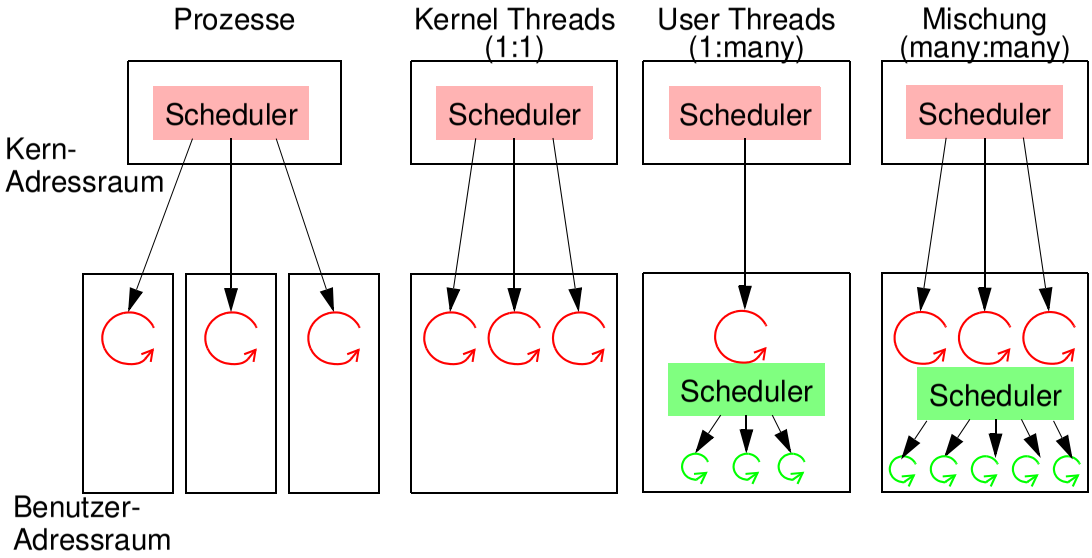
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-5 Thread-Konzepte in UNIX/Linux

- verschiedene Implementierungen von Thread-Paketen verfügbar
  - ➤ reine User-level Threads
    eine beliebige Zahl von User-level Threads wird auf einem Kernel Thread "gemultiplexed" (*many:1*)
  - ➤ reine Kernel-level Threads
    jedem auf User-level sichtbaren Thread ist 1:1 ein Kernel Thread zugeordnet (*1:1*)
  - ➤ Mischungen: eine große Zahl von User-level Threads wird auf eine kleinere Zahl von Kernel Threads abgebildet (*many:many*)
    + User-level Threads sind billig
    + die Kernel Threads ermöglichen echte Parallelität auf einem Multiprozessor
    + wenn sich ein User-level Thread blockiert, dann ist mit ihm der Kernel Thread blockiert in dem er gerade abgewickelt wird — aber andere Kernel Threads können verwendet werden um andere, lauffähige User-level Threads weiter auszuführen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-5 Thread-Konzepte in UNIX/Linux (2)



- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE POSIX Standard P1003.4a

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-6 pthread-Benutzerschnittstelle

- **Pthreads-Schnittstelle (Basisfunktionen)**:

pthread_create	Thread erzeugen & Startfunktion angeben
pthread_exit	Thread beendet sich selbst
pthread_join	Auf Ende eines anderen Threads warten
pthread_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-6 pthread-Benutzerschnittstelle (2)

- Threaderzeugung

```
#include <pthread.h>

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

thread	Thread-Id
attr	modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.

Thread wird erzeugt und ruft Funktion **start_routine** mit Parameter **arg** auf

Als Rückgabewert wird 0 geliefert. Im Fehlerfall -1 und **errno** wird gesetzt

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-6 pthread-Benutzerschnittstelle (3)

- Thread beenden (bei return aus **start_routine** oder):

```
void pthread_exit(void *retval)
```

Der Thread wird beendet und **retval** wird als Rückgabewert zurück geliefert (siehe pthread_join)

- Auf Thread warten und exit-Status abfragen:

```
int pthread_join(pthread_t thread, void **retvalp)
```

Wartet auf den Thread mit der Thread-ID **thread** und liefert dessen Rückgabewert über **retvalp** zurück.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall -1 und **errno** wird gesetzt

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-7 Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];

int main(int argc, char* argv[]) {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,
                                  (void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}

void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung

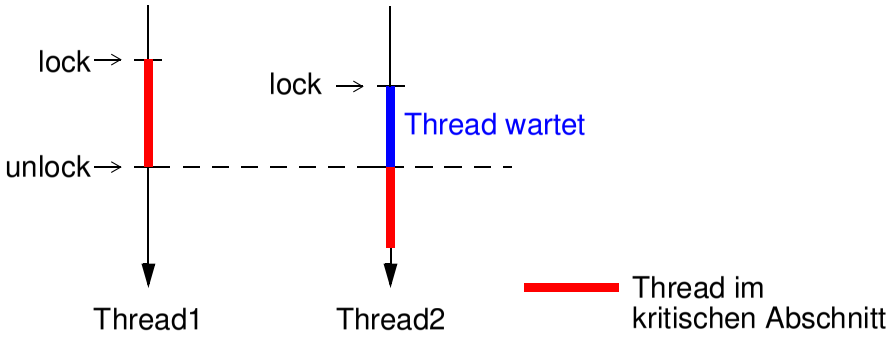
- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
  - Implementierung durch den Systemkern
  - komplexe Datenstrukturen, aufwändig zu programmieren
  - für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache ***Mutex***-Varianblen
  - gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung (2)

★ **Mutexes**

- Koordinierung von kritischen Abschnitten



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung (3)

... Mutexes (3)

- Schnittstelle
  - ◆ Mutex erzeugen

```
pthread_mutex_t m1
s = pthread_mutex_init(&m1, pthread_mutexattr_default);
```

  - ◆ Lock & unlock

```
s = pthread_mutex_lock(&m1);
... kritischer Abschnitt
s = pthread_mutex_unlock(&m1);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung (5)

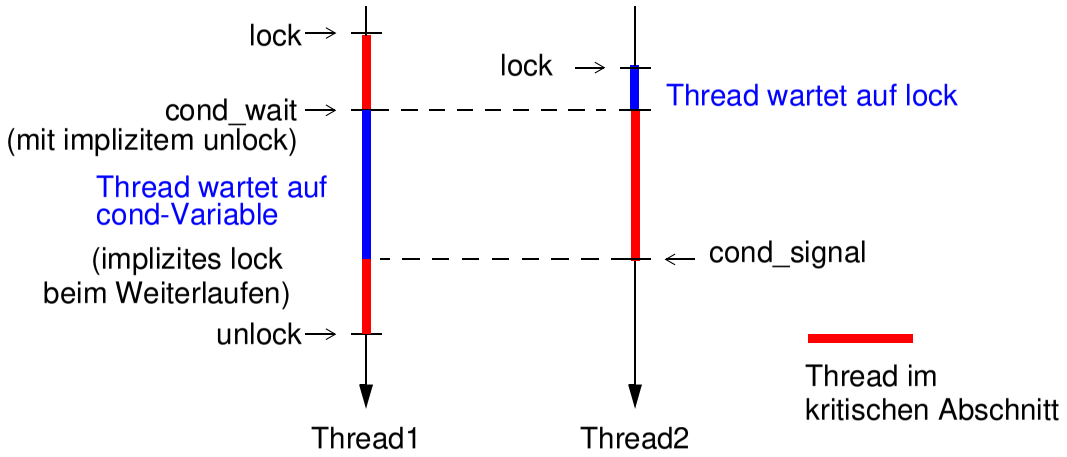
- Komplexere Koordinierungsprobleme können alleine mit Mutexes nicht implementiert werden
  - ➥ Problem:
    - Ein Mutex sperrt die eine komplexere Datenstruktur
    - Der Zustand der Datenstruktur erlaubt die Operation nicht
    - Thread muss warten, bis die Situation durch anderen Thread behoben wurde
    - Blockieren des Threads an einem weiteren Mutex kann zu Verklemmungen führen
  - ➥ Lösung: Mutex in Verbindung mit sleep/wakeup-Mechanismus
  - ➥ **Condition Variables**

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung (6)

★ **Condition Variables**

- Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung (7)

… Condition Variables (2)

- Realisierung
  - Thread reiht sich in Warteschlange der Condition Variablen ein
  - Thread gibt Mutex frei
  - Thread gibt Prozessor auf
  - Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
  - Deblockierter Thread muss als erstes den kritischen Abschnitt neu betreten (lock)
  - Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung (8)

… Condition Variables (3)

- Schnittstelle
  - Condition Variable erzeugen

```
pthread_cond_t c1;
s = pthread_cond_init(&c1,
                      pthread_condattr_default);
```

  - Beispiel: zählende Semaphore

P-Operation

```
void P(int *sem) {
  pthread_mutex_lock(&m1);
    while ( *sem == 0 )
      pthread_cond_wait
            (&c1, &m1);
    (*sem)--;
  pthread_mutex_unlock(&m1);
}
```

V-Operation

```
void V(int *sem) {
  pthread_mutex_lock(&m1);
    (*sem)++;
    pthread_cond_signal(&c1);
  pthread_mutex_unlock(&m1);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

I-8 Pthreads-Koordinierung (8)

… Condition Variables (4)

- Bei **`pthread_cond_signal`** wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher
- Mit **`pthread_cond_broadcast`** werden alle wartenden Threads aufgeweckt
- Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.