52 Überblick über die 12. Übung

- Vergleich von Prozeß und Thread-Konzepten
- POSIX-Threads

! Anmeldung zur Klausur bis zum 10.02. über das W.A.S. !

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

53 Motivation von Threads

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, ...)

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

54 Vergleich von Prozeß- und Thread-Konzepten

- mehrere **UNIX-Prozesse** mit gemeinsamen Speicherbereichen

Bewertung:

+ echte Parallelität möglich
– viele Betriebsmittel zur Verwaltung eines Prozesses notwendig; Prozeßumschaltungen aufwendig → teuer
– innerhalb einer solchen Prozeßfamilie wäre häufig ein anwendungsorientiertes Scheduling notwendig: schwierig realisierbar

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

54.0 Vergleich von Prozeß- und Thread-Konzepten (2)

- ***User-Level-Threads*** (Koroutinen) — Realisierung von Threads auf Benutzerebene innerhalb eines Prozesses

Bewertung:

+ Erzeugung von Threads und Umschaltung extrem billig
+ Verwaltung und Scheduling anwendungsorientiert möglich
– Systemkern hat kein Wissen über diese Threads
  - ➥ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
  - ➥ in Multiprozessorsystemen keine parallelen Abläufe möglich
  - ➥ wird eine Koroutine 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.

54.0 Vergleich von Prozeß- und Thread-Konzepten (3)

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

  Bewertung:

  + eine Gruppe leichtgewichtiger Prozesse nutzt gemeinsam eine Menge von Betriebsmitteln
  + jeder leichtgewichtige Prozeß ist aber als eigener Aktivitätsträger dem Betriebssystemkern bekannt
    - eigener Programmzähler
    - eigener Registersatz
    - eigener Stack

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

54.0 Vergleich von Prozeß- und Thread-Konzepten (4)

- Vergleich

	Prozesse	Kernel-Threads	User-Threads
Kosten	– teuer	○ mittel	+ billig
Betriebssystemeingliederung	+ gut	+ gut	– schlecht
Interaktion untereinander	– schwierig	+ einfach	+ einfach
Benutzerkonfigurierbarkeit	– nein	– nein	+ ja
Gerechtigkeit	– nein	+ ja	± teils

  – Gerechtigkeit bedeutet:
  wie kommt das System damit klar, wenn eine Anwendung eine große Anzahl von Aktivitätsträgern erzeugt, eine andere dagegen eine geringe — werden Zeitscheiben an Anwendungen oder an Aktivitätsträger vergeben?

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

54.0 Vergleich von Prozeß- und Thread-Konzepten (3)

... Bewertung *Kernel-Threads (lightweight processes)*

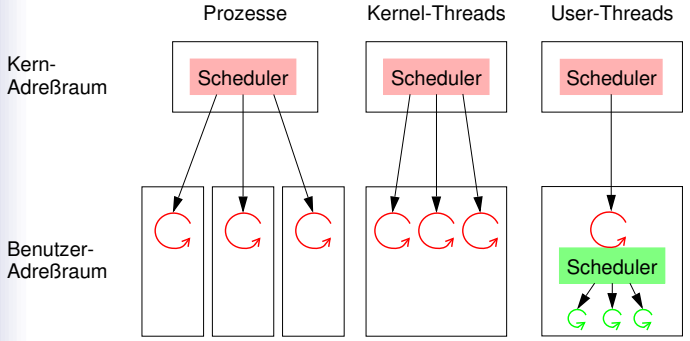
+ Umschalten zwischen zwei leichtgewichtigen Prozessen einer Gruppe ist erheblich billiger als eine normale Prozeßumschaltung
  - ➥ es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf)
  - ➥ Adreßraum muß nicht gewechselt werden
  - ➥ alle Systemressourcen bleiben verfügbar

– Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei "schwergewichtigen" Prozessen, aber erheblich teuerer als bei *user-level-Threads*

– Verwaltung und Scheduling meist durch Kern vorgegeben

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

54.0 Vergleich von Prozeß- und Thread-Konzepten (5)



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

55 UNIX — Prozesse, LWPs & Threads

- Thread-Konzept zunehmend auch in UNIX-Systemen realisiert
  - Solaris
  - HP UX
  - Digital UNIX
  - Linux
  - …
- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE POSIX Standard P1003.4a
- Pthreads-Implementierungen aber sehr unterschiedlich!
  - ➤ reine User-level-Threads (HP-UX)
  - ➤ reine Kernel-Threads (Linux, MACH, KSR-UNIX, Digital UNIX)
  - ➤ parametrierbare Mischung (Solaris)
- Daneben z. T. auch andere Thread-Bibliotheken (z. B. Solaris-Threads)

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

56 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.

56.0 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 und -1 im Fehlerfall und **errno** wird gesetzt

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

56.0 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 und -1 im Fehlerfall und **errno** wird gesetzt

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

56.1 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.

57 Pthreads-Koordinierung

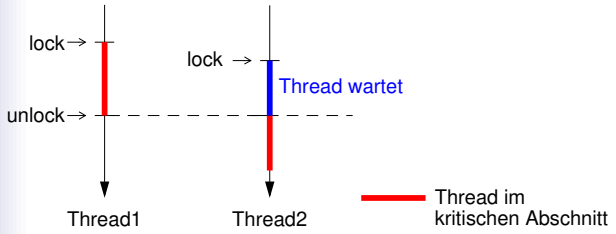
- UNIX-Semaphore für Koordinierung von leichtgewichtigten Prozessen zu teuer
  - Implementierung durch den Systemkern
  - komplexe Datenstrukturen
- Bei Koordinierung von Threads reichen meist einfache ***mutex***-Semaphore
  - gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

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

57.0 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.

57.0 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.

57.0 Pthreads-Koordinierung (5)

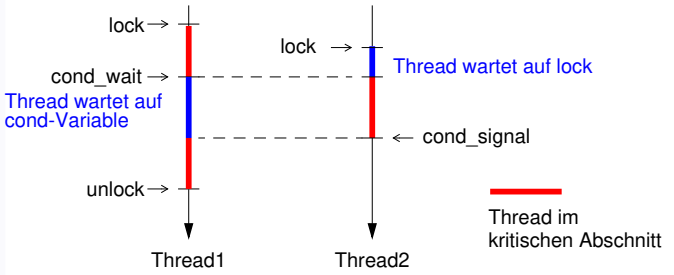
- Komplexere Semaphore können alleine mit Mutexes nicht implementiert werden
  - ➥ Problem:
    - Ein Mutex sperrt die Datenstruktur des komplexen Semaphors
    - Der Zustand der Datenstruktur erlaubt die Operation nicht
    - Blockieren an einem weiteren Mutex kann zu Verklemmungen führen
  - ➥ Lösung: 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.

57.0 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.

57.0 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 (Analogie zu UNIX sleep/wakeup !)

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

57.0 Pthreads-Koordinierung (8)

… Condition Variables (3)

- Schnittstelle
  - Condition Variable erzeugen

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

  - Betriebsmittel belegen

```
pthread_mutex_lock(&m1);
   while ( resource_busy )
     s = pthread_cond_wait
           (&c1, &m1);
   resource_busy = TRUE;
pthread_mutex_unlock(&m1);
... /* Betriebsmittel nutzen */
```

  - Betriebsmittel freigeben

```
pthread_mutex_lock(&m1);
   resource_busy = FALSE;
   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.

57.0 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.