

U6 6. Übung

- Organisatorisches
- Besprechung Aufgabe 5 (crawl)
- POSIX-Threads
 - Motivation
 - Thread-Konzepte
 - pthread-API
 - Koordinierung
- Aufgabe 6: palim — rekursive, parallele Suche nach einer Zeichenkette

U6-2 Motivation von Threads

- UNIX-Prozesskonzept: eine Ausführungsumgebung (virtueller Adressraum, Rechte, Priorität, ...) mit einem Aktivitätsträger (= Kontrollfluss, Faden 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

U6-1 Organisatorisches

- Zusätzliche Tafelübung zur SP1-Klaurvorbereitung
 - ◆ am 19.07.2011, um 12:15 Uhr im 00.153
 - ◆ *SP1-Klausur schreiben Mathematiker und Technomathematiker*
 - ◆ In der Tafelübung wird SP1-Klausur vom Juli 2010 gemeinsam erarbeitet
 - Eine Vorbereitung der Klausur im Vorfeld der Tafelübung wird empfohlen

U6-3 Vergleich von Thread-Konzepten

- **User-Level Threads:** Federgewichtige Prozesse
 - 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

U6-3 Vergleich von Thread-Konzepten (2)

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

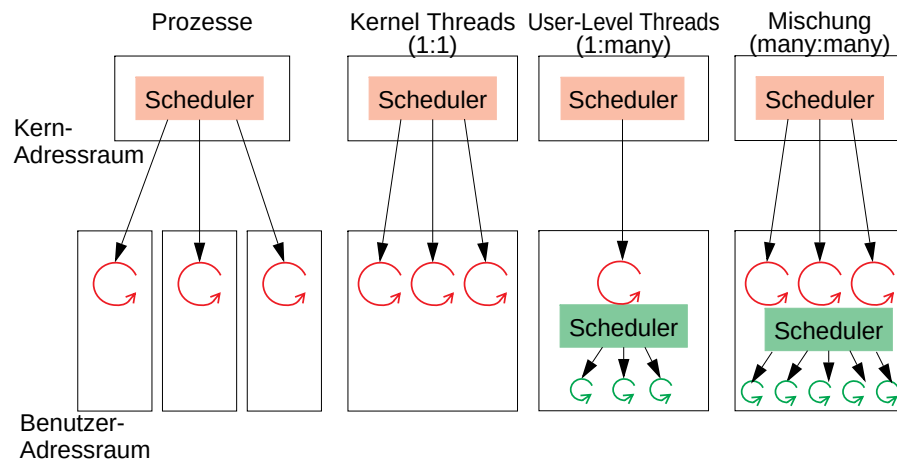
Bewertung:

- + 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 teurer als bei User-Level Threads

U6-4 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 "gemultiplext" (*many:1*)
 - reine Kernel 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

U6-4 Thread-Konzepte in UNIX/Linux (2)



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

U6-5 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_detach** Thread in den *detached-state* versetzen
 - pthread_self** Eigene Thread-Id abfragen
 - pthread_yield** Prozessor zugunsten eines anderen Threads abgeben

- Funktionen in Pthreads-Bibliothek zusammengefasst
gcc ... -pthread

U6-5 pthread-Benutzerschnittstelle (2)

U6-5 pthread-Benutzerschnittstelle

■ 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 wird **errno** nicht gesetzt, aber ein Fehlercode als Ergebnis zurückgeliefert. Um **perror(3)** verwenden zu können, muss der Rückgabewert erst in der **errno** gespeichert werden.

Systemprogrammierung — Übungen

© Jürgen Kleinöder, Michael Stillerich • Universität Erlangen-Nürnberg • Informatik 4, 2011

U06.fm 2011-07-06 13.57

U6.9

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

U6-6 Beispiel (Multiplikation Matrix mit Vektor)

U6-6 Beispiel (Multiplikation Matrix mit Vektor)

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

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

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

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

■ Kein schreibender Zugriff auf ein gemeinsames Datum

Systemprogrammierung — Übungen

© Jürgen Kleinöder, Michael Stillerich • Universität Erlangen-Nürnberg • Informatik 4, 2011

U06.fm 2011-07-06 13.57

U6.11

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

U6-5 pthread-Benutzerschnittstelle (3)

U6-5 pthread-Benutzerschnittstelle

■ 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, Ressourcen freigeben 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.

■ Ressourcen automatisch bei Beendigung freigeben:

```
int pthread_detach(pthread_t thread)
```

Die mit dem Thread **thread** verbundenen Systemressourcen werden bei dessen Beendigung automatisch freigegeben. Der exit-Status kann nicht abgefragt werden.

Systemprogrammierung — Übungen

© Jürgen Kleinöder, Michael Stillerich • Universität Erlangen-Nürnberg • Informatik 4, 2011

U06.fm 2011-07-06 13.57

U6.10

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

U6-7 Koordinierung - Motivation

U6-7 Koordinierung - Motivation

```
static double a[100][100], sum;

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

static void *sumRow(void *arg) {
    int i = (int) arg;
    double localSum = 0;

    for (int j = 0; j < 100; j++)
        localSum += a[i][j];
    sum += localSum;
    return NULL;
}
```

■ Welches Problem kann hier auftreten?

Systemprogrammierung — Übungen

© Jürgen Kleinöder, Michael Stillerich • Universität Erlangen-Nürnberg • Informatik 4, 2011

U06.fm 2011-07-06 13.57

U6.12

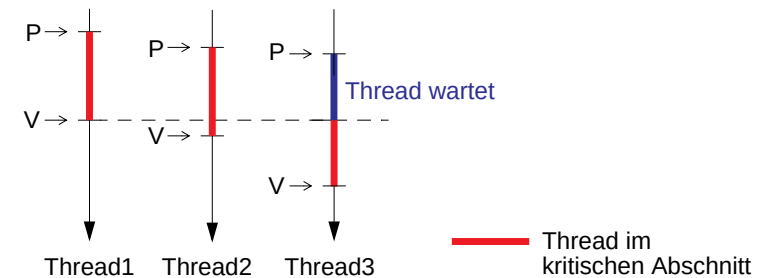
Reproduktion jeder Art oder Verwendung dieser Unterlage, an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

U6-8 Koordinierung - Semaphore

- Zur Koordinierung von Threads können **Semaphore** verwendet werden
- 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
- Stattdessen Verwendung einer eigenen Semaphorimplementierung mit atomaren **P()**- und **V()**-Operationen

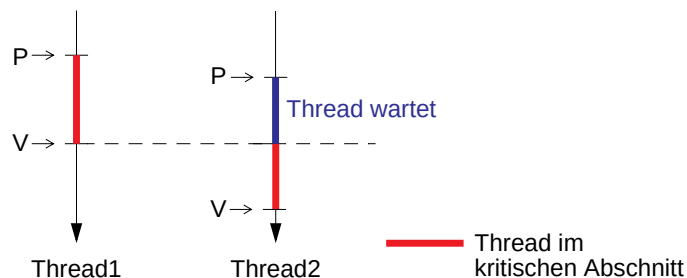
U6-9 Koordinierung - Limitierung von Ressourcen

- Verwendung eines zählenden Semaphors
- Beispiel: Nur zwei aktive Threads gleichzeitig gewünscht
 - ◆ Initialisierung des Semaphors mit 2



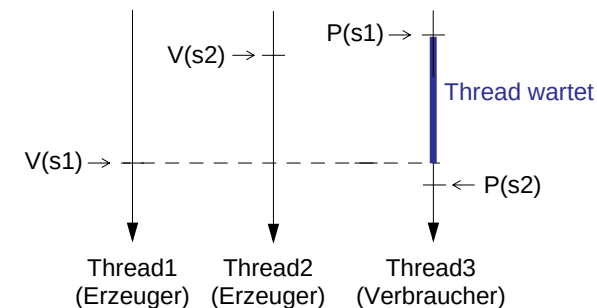
U6-10 Koordinierung - Gegenseitiger Ausschluss

- Spezialfall des zählenden Semaphors: Binärer Semaphor
 - ◆ Initialisierung des Semaphors mit 1
- Beispiel: Schreibender Zugriff auf ein gemeinsames Datum



U6-11 Koordinierung - Signalisierung

- Benachrichtigung eines anderen Threads über ein Ereignis
- Beispiel: Bereitstehen von Zwischenergebnissen



```
static double a[100][100], sum;
static SEM* sem;

int main(int argc, char* argv[]) {
    pthread_t tids[100];
    sem = sem_init(1);
    // Fehlerabfrage
    ...
    sem_del(sem);
}

static void *sumRow(void *arg) {
    int i = (int) arg;
    double localSum = 0;

    for (int j = 0; j < 100; j++)
        localSum += a[i][j];
    P(sem);
    sum += localSum;
    V(sem);
    return NULL;
}
```

- Mehrfädige, rekursive Suche nach einer Zeichenkette in Verzeichnisbäumen
- palim nicht ausführen auf:
 - ◆ SunRay-Servern (fau0sr0, faui0sr1, faui0sr2, faui05, faui01)
 - ◆ SunRay-Thin-Clients
 - ◆ faui09er und faui01er Rechnern

1 Haupt-Thread (main)

- Startet für jeden als Parameter übergebenen Verzeichnisbaum einen eigenen *crawl-Thread*
- Aktualisiert die Statusausgabe kontinuierlich, bis die Suche abgeschlossen ist, und terminiert anschließend den Prozess

2 crawl-Thread

- Durchsucht einen Verzeichnisbaum rekursiv
- Startet für jede gefundene reguläre Datei einen eigenen *grep-Thread*

3 grep-Thread

- Öffnet reguläre Datei und zählt u.a. die Anzahl der Zeilen, die die Suchzeichenkette enthalten

4 Semaphor-Modul

- Zählende P/V-Semaphoren zur Synchronisation von POSIX-Threads
- Ist vorgeben und muss nicht implementiert werden