U9 9. Übung

U9-1 Überblick

- Besprechung Aufgabe 5 (printdir)
- Posix Threads

U9-2 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

U9-3 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

U9-3 Vergleich von Thread-Konzepten (2)

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

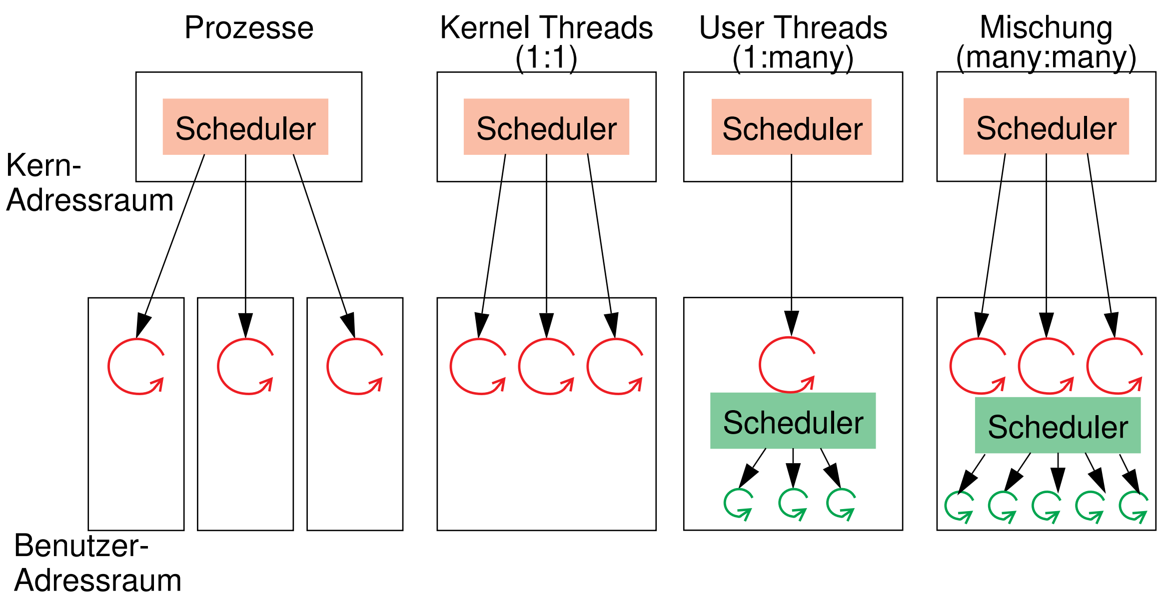
Bewertung:

- \+ 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 teuerer als bei User-level Threads

U9-4 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 "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

U9-4 Thread-Konzepte in UNIX/Linux (2)



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

U9-5 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_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

- Funktionen in pthreads-Library zusammengefasst
  `gcc ... -lpthread`

U9-5 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

U9-5 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

U9-6 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;
}
```

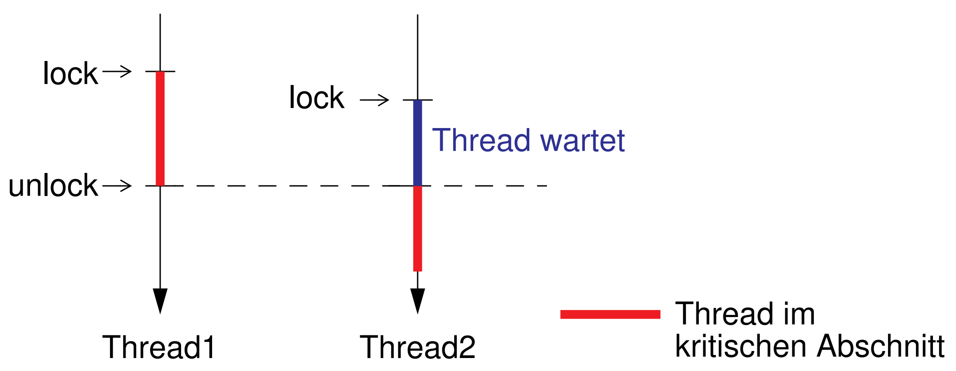
U9-7 Pthreads-Koordinierung

- 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***-Variablen
  - gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

U9-7 Pthreads-Koordinierung (2)

★ **Mutexes**

- Koordinierung von kritischen Abschnitten



U9-7 Pthreads-Koordinierung (3)

… Mutexes (2)

- Schnittstelle
  - ◆ Mutex erzeugen

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

  - ◆ Lock & unlock

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


U9-7 Pthreads-Koordinierung (4)

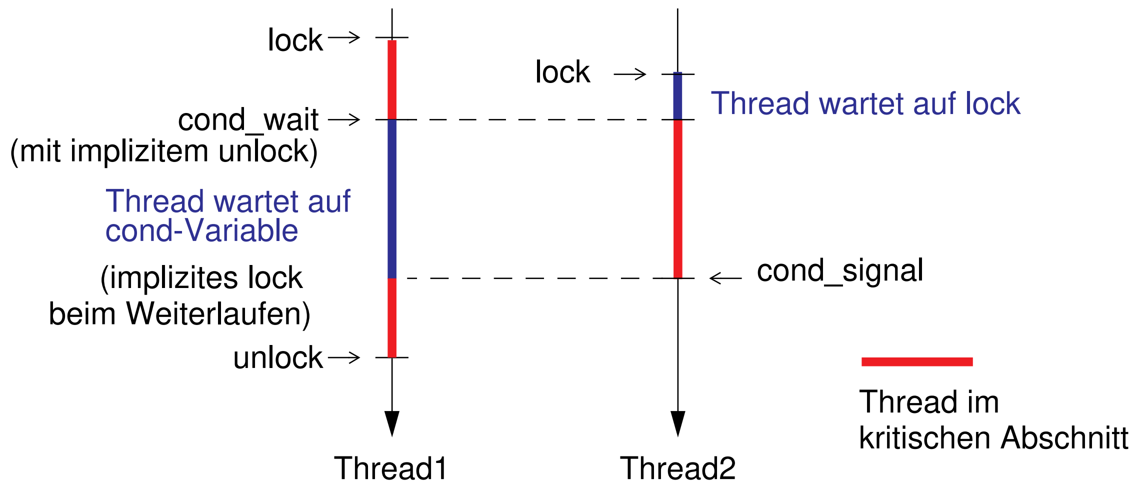
… Mutexes (3)

- 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**

U9-7 Pthreads-Koordinierung (5)

★ **Condition Variables**

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



U9-7 Pthreads-Koordinierung (6)

… 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

U9-7 Pthreads-Koordinierung (7)

... Condition Variables (3)

- Schnittstelle
  - Condition Variable erzeugen

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

  - 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_broadcast(&c1);
  pthread_mutex_unlock(&m1);
}
```


U9-7 Pthreads-Koordinierung (8)

... Condition Variables (4)

- Bei **`pthread_cond_signal`** wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher
  - ➤ evtl. Prioritätsverletzung wenn nicht der höchstpriore gewählt wird
  - ➤ Verklemmungsgefahr wenn die Threads unterschiedliche Wartebedingungen haben
- 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