U6 6. Übung (Mathematik / Technomathematik)

- Besprechung Aufgabe 4 (halde)
- Besprechung Aufgabe 5 (crawl)
- Besprechung der Mini-Klausur
- POSIX-Threads
  - ➤ Motivation
  - ➤ Thread-Konzepte
  - ➤ pthread-API
  - ➤ pthread-Koordinierung
- Aufgabe 9: recgrep — rekursive, parallele Suche (letzte Aufgabe)
  - ➤ Aufgaben 6, 7 und 8 sind für die 5-ECTS-Variante von Systemprogrammierung nicht relevant!

U6-1 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

U6-2 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

U6-2 Vergleich von Thread-Konzepten (2)

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

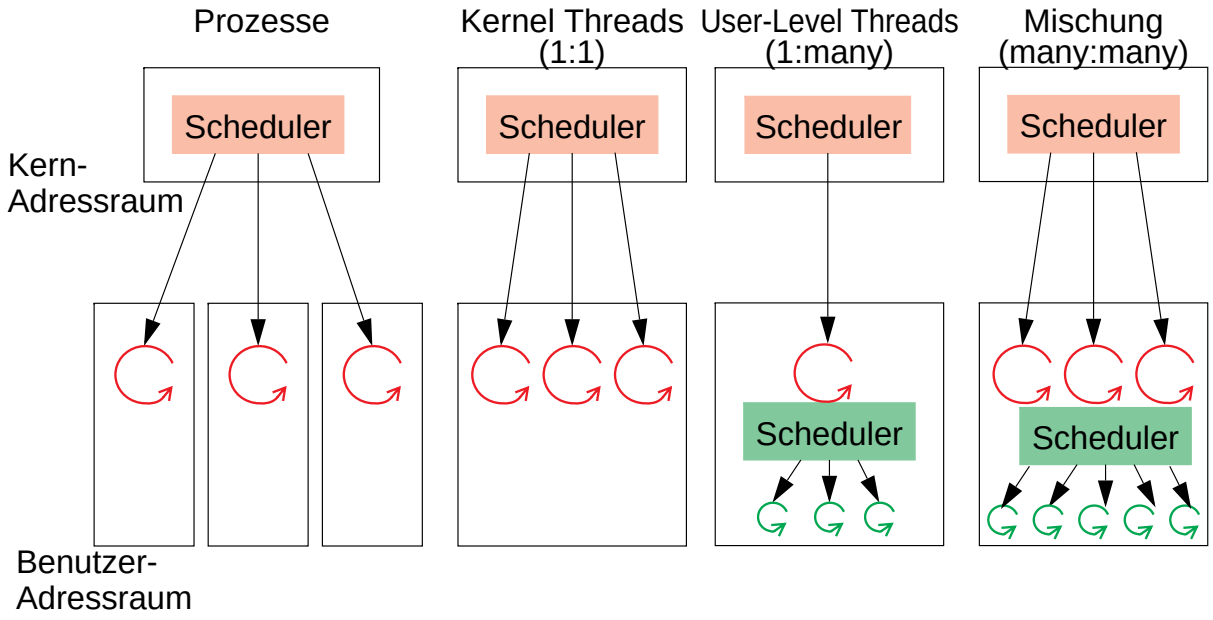
Bewertung:

- \+ 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

U6-3 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

U6-3 Thread-Konzepte in UNIX/Linux (2)



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

U6-4 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-Bibliothek zusammengefasst
  `gcc ... -pthread`

U6-4 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 wird ein Fehlercode als Ergebnis zurückgeliefert.

U6-4 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 wird ein Fehlercode als Ergebnis zurückgeliefert.

U6-5 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;
}
```

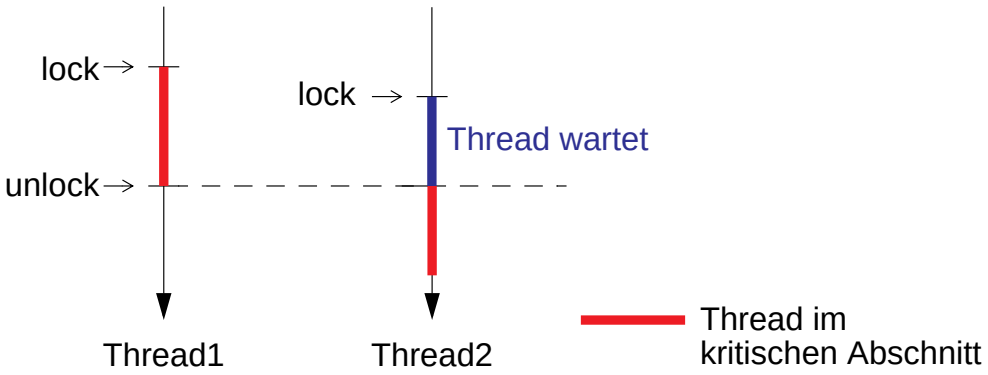
U6-6 Pthreads-Koordinierung

- 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)*

U6-6 Pthreads-Koordinierung (2)

★ **Mutexes**

- Koordinierung von kritischen Abschnitten



U6-6 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);
```


U6-6 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

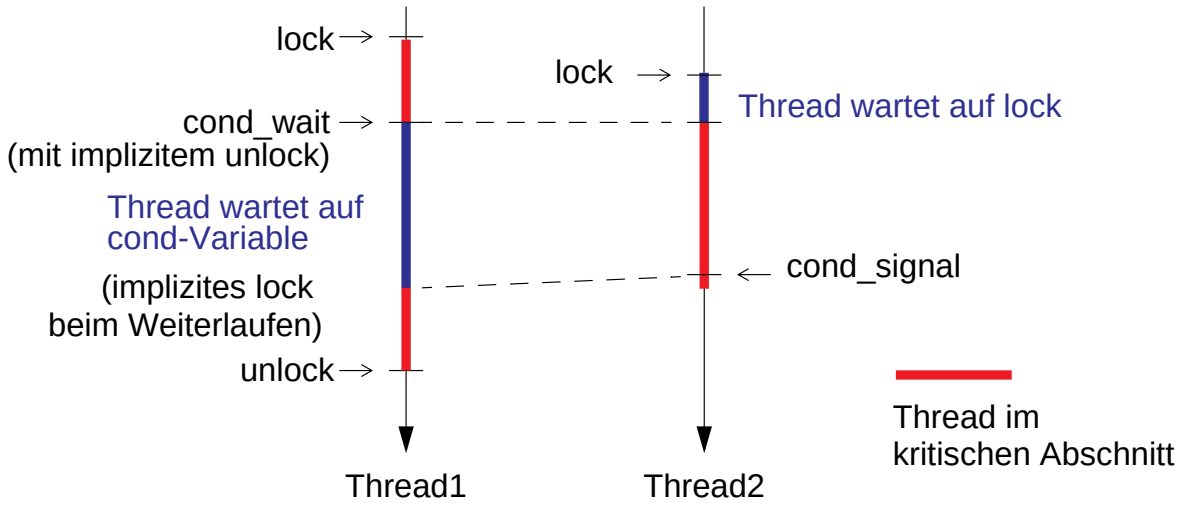
➥ Lösung: Mutex in Verbindung mit sleep/wakeup-Mechanismus

➥ **Condition Variables**

U6-6 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



U6-6 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

U6-6 Pthreads-Koordinierung (7)

… Condition Variables (3)

- Schnittstelle
  - ◆ Condition Variable erzeugen

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

  - ◆ Beispiel: zählender Semaphor

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);
}
```


U6-6 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

U6-7 Aufgabe 9: recgrep

- Mehrfädige Suche
- Hauptfaden sucht Dateien und öffnet sie
  - ➤ crawl-Lösung zu 99 % verwendbar
  - ➤ statt Dateinamen auszugeben wird die Datei geöffnet und an den Suchmechanismus übergeben
- Arbeiterthreads suchen in den geöffneten Dateien
- Auftragsübergabe über einen Ringpuffer
  - ◆ Hauptfaden (Schreiber) fügt Namen und Filedeskriptoren der geöffneten Dateien in den Ringpuffer ein
  - ◆ Arbeiterthreads (Leser) entnehmen die Dateiinformationen aus dem Ringpuffer, bearbeiten die Anfrage und schließen die Datei
  - ◆ danach entnimmt ein freier Arbeiterthread die nächsten Dateiinformationen

1 Semaphor-Modul

- Synchronisation von Überlauf/Unterlauf des Ringpuffers
- Zählende P/V-Semaphoren zur Synchronisation von POSIX-Threads

2 Ringpuffer-Modul

- Ringpuffer zur Verwaltung von int-Werten
- Randbedingungen: ein Produzent, mehrere Konsumenten
- blockierende Synchronisation bei Überlauf/Unterlauf
- Synchronisation von gleichzeitigen Zugriffen der Arbeiterthreads

3 Koordinierung eines Ringpuffers

- Eines der klassischen Koordinierungsprobleme in der Informatik
- Puffer fester Größe
  - mehrere Prozesse/Threads lesen und beschreiben den Puffer
  - beispielsweise Erzeuger und Verbraucher (Erzeuger-Verbraucher-Problem) (z.B. Erzeuger liest einen Katalog; Verbraucher zählt Zeilen; Gesamtanwendung zählt Einträge in einem Katalog)
  - UNIX-Pipe ist solch ein Puffer
- Problem
  - Koordinierung von Leser und Schreiber
    - gegenseitiger Ausschluss beim Pufferzugriff
    - Blockierung des Lesers bei leerem Puffer
    - Blockierung des Schreibers bei vollem Puffer

3 Koordinierung eines Ringpuffers (2)

- Implementierung mit zählenden Semaphoren
  - zwei Funktionen zum Zugriff auf den Puffer
    - **`put`** stellt Zeichen in den Puffer
    - **`get`** liest ein Zeichen vom Puffer
  - Puffer wird durch ein Feld implementiert, das als Ringpuffer wirkt
    - zwei Integer-Variablen enthalten Feldindizes auf den Anfang und das Ende des Ringpuffers
  - ein Semaphor für den gegenseitigen Ausschluss
  - je einen Semaphor für das Blockieren an den Bedingungen „Puffer voll“ und „Puffer leer“
    - Semaphor **`full`** zählt wieviele Zeichen noch in den Puffer passen
    - Semaphor **`empty`** zählt wieviele Zeichen im Puffer sind

3 Koordinierung eines Ringpuffers (3)

```
char buffer[N];
int inslot= 0, outslot= 0;
semaphor mutex= 1, empty= 0, full= N;
```

```
void put( char c )
{
    P( &full );
    P( &mutex );
    buffer[inslot]= c;
    if( ++inslot >= N )
        inslot= 0;
    V( &mutex );
    V( &empty );
}
```

```
char get( void )
{
    char c;

    P( &empty );
    P( &mutex );
    c= buffer[outslot];
    if( ++outslot >= N )
        outslot= 0;
    V( &mutex );
    V( &full );
    return c;
}
```