

Übungen zu Systemprogrammierung 1

Ü5 – Threads und Koordinierung

Sommersemester 2019

Simon Ruderich, Dustin Nguyen, Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



6.1 Threads

6.2 Koordinierung

6.3 Aufgabe 5: mach

6.4 Gelerntes anwenden

6.1 Threads

6.2 Koordinierung

6.3 Aufgabe 5: mach

6.4 Gelerntes anwenden



- UNIX-Prozesskonzept (Ausführungsumgebung mit einem Aktivitätsträger) für viele heutige Anwendungen unzureichend
 - keine parallelen Abläufe innerhalb eines logischen Adressraums auf Multiprozessorsystemen
 - typische UNIX-Server-Implementierungen benutzen die fork-Operation, um einen Server-Prozess für jeden Client zu erzeugen
 - Verbrauch unnötig vieler System-Ressourcen
 - zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adressraums nützlich
- Lösung: bei Bedarf weitere Aktivitätsträger in einem UNIX-Prozess erzeugen



- Faden vs. Thread
 - Vorlesung verwendet Begriff „Faden“:
„konkreter Strang: roter Faden in einem Programm“
 - Übung verwendet Begriff „Thread“:
siehe folgende Folien
- Thread ist keine direkte Übersetzung von Faden,
genaue Definition von Faden ist schwierig
- Für die Übung ist nur Thread relevant



Federgewichtige Prozesse (User-Threads)

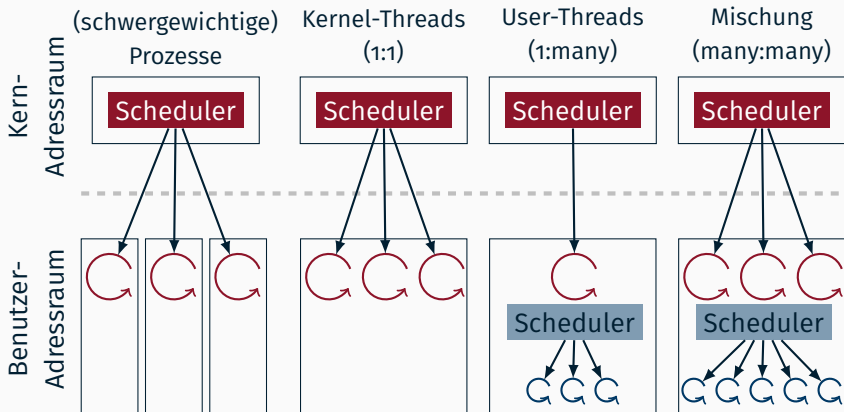
- Realisierung auf Anwendungsebene
- Systemkern sieht nur **einen** Kontrollfluss
- + Erzeugung von Threads extrem billig
- Systemkern hat kein Wissen über diese Threads
 - in Multiprozessorsystemen keine parallelen Abläufe möglich
 - wird **ein** User-Thread blockiert, sind **alle** User-Threads blockiert
 - Scheduling zwischen den Threads schwierig

Leichtgewichtige Prozesse (Kernel-Threads)

- + Gruppe von Threads nutzt gemeinsam die Betriebsmittel eines Prozesses
- + jeder Thread ist als eigener Aktivitätsträger dem Betriebssystemkern bekannt
- Kosten für Erzeugung erheblich geringer als bei Prozessen, aber erheblich teurer als bei User-Threads

Umschaltungskosten (“Gewichtsklasse”)

Kosten für Umschaltung zwischen Threads hängt maßgeblich von der Anzahl der Adressraumwechsel ab.





■ POSIX-Thread erzeugen

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

- thread Thread-ID (Ausgabeparameter)
- attr Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.
- Nach der Erzeugung führt der Thread die Funktion `start_routine` mit Parameter `arg` aus
- 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.

■ Eigene Thread-ID ermitteln

```
pthread_t pthread_self(void);
```

- Die Funktion kann nie fehlschlagen.



- Thread beenden (bei Rücksprung aus `start_routine` oder):
`void pthread_exit(void *retval);`
 - Der Thread wird beendet und `retval` wird als Rückgabewert zurück geliefert (siehe `pthread_join(3)`)
- Auf Thread warten, Ressourcen freigeben und Rückgabewert 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 Rückgabewert der Thread-Funktion kann nicht abgefragt werden.



```
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 *) i);
    for(int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
    ...
}

static void *mult(void *cp) {
    int i = (int) cp;
    double sum = 0;
    for(int j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return NULL;
}
```

- Casts zwischen `int` und Zeiger (bei Parameterübergabe für `pthread_create()`) problematisch – **nicht zu Hause nachmachen!**



- Generischer Ansatz mit Hilfe einer Struktur für die Argumente

```
struct param {  
    int index;  
};
```

- Für jeden Thread eine eigene Argumenten-Struktur anlegen
 - Speicher je nach Situation auf dem Heap oder dem Stack allozieren

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



```
static void *mult(void *arg) {
    struct param *par = arg;

    double sum = 0;
    for(int j = 0; j < 100; j++) {
        sum += a[par->index][j] * b[j];
    }
    c[par->index] = sum;
    return NULL;
}
```

- Zugriff auf den threadspezifischen Parametersatz über
(gecasteten) Parameter (`void *arg` → `struct param *par`)



6.1 Threads

6.2 Koordinierung

6.3 Aufgabe 5: mach

6.4 Gelerntes anwenden



Was macht das Programm? Welches Problem kann auftreten?

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

int main(int argc, char *argv[]) {
    pthread_t tids[100];
    struct param args[100];

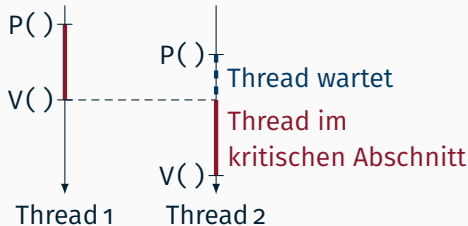
    for(int i = 0; i < 100; i++) {
        args[i].index = i;
        pthread_create(&tids[i], NULL, sumRow, &args[i]);
    }
    for(int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
}

static void *sumRow(void *arg) {
    struct param *par = arg;
    double localSum = 0;
    for(int j = 0; j < 100; j++)
        localSum += a[par->index][j];
    sum += localSum;
    return NULL;
}
```

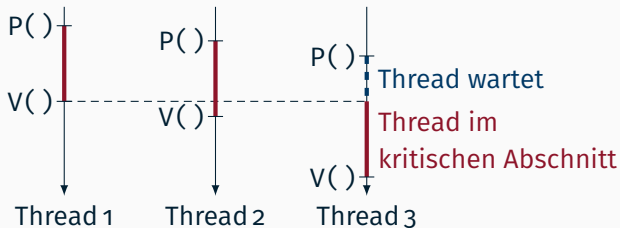


- Zur Koordinierung von Threads können Semaphore verwendet werden
- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
 - Implementierung durch den Systemkern
 - komplexe Datenstrukturen, aufwändig zu programmieren
 - für die Koordinierung von Threads viel zu teuer
- Stattdessen Verwendung einer eigenen Semaphorimplementierung mit atomaren P()- und V()-Operationen
 - Datenstruktur mit (atomarer) Zählervariable
 - P() dekrementiert Zähler und blockiert Aufrufer, falls Zähler ≤ 0
 - V() inkrementiert Zähler und weckt ggf. wartende Threads
 - Mehr Details: s. Vorlesung B | VI.1, Seite 22f.

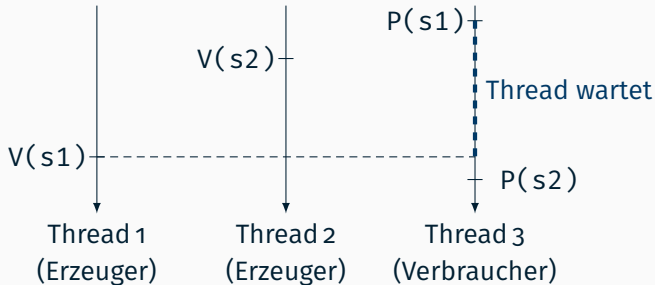
- Spezialfall des zählenden Semaphors: Binärer Semaphor
 - Initialisierung des Semaphors mit 1
- Beispiel: Schreibender Zugriff auf ein gemeinsames Datum



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



- Benachrichtigung eines anderen Threads über ein Ereignis
- Beispiel: Bereitstellen von Zwischenergebnissen
 - Initialisierung des Semaphors mit 0





- Semaphor erzeugen
`SEM *semCreate(int initVal);`
- P/V-Operationen
`void P(SEM *sem);`
`void V(SEM *sem);`
- Semaphor zerstören
`void semDestroy(SEM *sem);`
- Semaphor-Modul und zugehörige Headerdatei befinden sich im pub-Verzeichnis.
- Semaphor-Funktionen bekannt machen mit `#include "sem.h"`



6.1 Threads

6.2 Koordinierung

6.3 Aufgabe 5: mach

6.4 Gelerntes anwenden

Funktionsweise der `mach`

Gruppen von Befehlen aus einer Eingabedatei lesen und parallel ausführen

- Maximale Parallelität beachten
- Jeweils am Ende der Gruppe auf Beendigung aller Prozesse warten

Beispiel-Aufruf

```
user@host:~$ ./mach <anzahl threads> <mach-datei>  
user@host:~$ ./mach 2 machfile
```

machfile

```
gcc -c file1.c  
gcc -c file2.c  
gcc -c file3.c  
gcc -c sem.c
```

```
gcc -o main file1.o file2.o file3.o sem.o
```



6.1 Threads

6.2 Koordinierung

6.3 Aufgabe 5: mach

6.4 Gelerntes anwenden



„Aufgabenstellung 1“

Beispiel von Folie 13 mit Hilfe eines Semaphors korrekt synchronisieren