System-Programmierung 6: Threads & Synchronisation

n|w

Ablauf heute

⅓ Vorlesung,

⅔ Hands-on,

Feedback.

Slides, Code & Hands-on: tmb.gr/syspr-6

Threads

Threads, wie Prozesse, erlauben einer Anwendung mehrere Aufgaben gleichzeitig zu erledigen.

Ein einzelner Prozess kann mehrere Threads haben.

Wenn ein Thread blockiert, z.B. beim Warten auf I/O, können andere Threads unterdessen weiter laufen, auf Mehrprozessorsystemen sogar echt parallel.

n|w

Speicher Layout

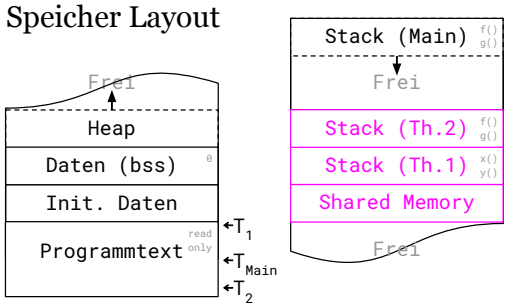
Jeder Thread führt dasselbe Programm aus, und teilt sich denselben globalen Speicher, mit initialisierten und uninitialisierten Daten-Segmenten und Heap.

Hingegen hat jeder Thread einen eigenen, "privaten" Stack für lokale Variablen und Funktionsaufrufe.

Jeder Thread führt dasselbe Programm aus, parallel zu und oft unabhängig von den anderen Threads.

n|w

Speicher Layout



n|w

Wieso Threads?

Informationen zwischen Prozessen auszutauschen ist schwierig, Parent und Child teilen keinen Speicher.

Prozesse mit *fork()* zu erzeugen ist relativ "teuer".

Threads tauschen Informationen einfach und schnell, in globalen Variablen oder auf dem Heap aus, dafür muss man auf korrekte Synchronisation achten.

Erzeugen von Threads ist relativ leichtgewichtig.

n|w

Thread Attribute

Threads teilen sich folgende Attribute *per Prozess*:

PID und Parent PID, Offene File Deskriptoren, Signal Handler, Terminal, Working Directory, CPU Zeit und konsumierte Ressourcen, Limiten für Ressourcen, …

Folgende Attribute gibt es für jeden Thread *separat*:

Thread ID, Signal Maske, *errno* Variable, Stack, …

n|w

Pthreads API

Die Pthreads API Schnittstelle definiert Datentypen:
`pthread_t, pthread_mutex_t, pthread_attr_t, …`

Und Funktionen um Threads zu erzeugen / beenden.

Die *Pthreads* Funktionen geben nicht `-1` zurück, der *return*-Wert ist `0` bei Erfolg, *+errno* bei Fehlern.

Programme, die *Pthreads* verwenden, werden mit dem *gcc* Compiler Flag *-pthread* kompiliert.

n|w

Thread kreieren

Thread kreieren mit Aufruf von *pthread_create()*:

```
int pthread_create(
  pthread_t *thread, // der neue Thread
  const pthread_attr_t *attr, // Default NULL
  void *(*start) (void *), // Start-Callback
  void *arg); // Argument für Start-Callback
```

Der Aufruf startet direkt *start(arg)* in Thread *thread*:

```
void *start(void *arg) { … } // arg casten
```

n|w

Thread beenden

Threads enden mit einem Aufruf von *pthread_exit()*:

```
void pthread_exit(void *status); // siehe Join
```

Oder die *start*-Funktion des Threads ruft *return* auf.

Oder Thread wird abgebrochen mit *pthread_cancel()*:

```
int pthread_cancel(pthread_t thread);
```

Oder beliebiger Thread ruft *exit()* auf, bzw. *main()* ruft *return* auf, worauf alle Threads sofort enden.

n|w

Thread IDs

Jeder Thread hat eine Prozess-weit eindeutige ID:

```
pthread_t pthread_self(void); // geht immer
```

z.B. für *pthread_join()*, *_detach()*, *_cancel()*, *_kill()*.

Thread IDs auf Gleichheit testen mit *pthread_equal()*:

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Der Typ *pthread_t* ist je nach Plattform verschieden.

Thread IDs bilden keinen Baum, nicht wie PIDs.

n|w

Thread Join

Auf Exit eines Threads *t* warten mit *pthread_join()*:

```
int pthread_join(pthread_t t, void **result);
```

Kommt sofort zurück, falls Thread bereits beendet.

Pro Thread ID *pthread_join()* nur einmal aufrufen, sonst ist das Verhalten des Aufrufs undefiniert.

Falls es kein Join gibt für einen Thread, wird er zu einem Zombie-Thread, wie Prozesse ohne *wait()*.

n|w

Thread Detach

Per Default sind Threads *joinable*, mit Detach sagt man dem System, dass kein Join passieren wird:
`int pthread_detach(pthread_t thread);`

Nach Ablauf des Threads räumt das System alles weg.

Ein solcher Thread kann nicht mehr gejoint werden.

Bei *exit()* werden auch solche Threads weggeräumt.

Hands-on, 30': Threads pthreads.!c

Schreiben Sie ein Programm *my_pthreads.c* welches einen Thread erzeugt, und mit *pthread_exit()* endet.

Geben Sie Thread ID, Argument und Resultat aus.

Wie würde man mehrere Argumente übergeben?

Hands-on, 15': Self Join self_join.!c

Schreiben Sie ein Programm *my_self_join.c* welches seine eigene Thread ID liest und einen Join macht:
`pthread_join(pthread_self(), NULL);`

Was, bzw. welcher Fehler passiert dabei auf Linux?

Pthreads und *errno* errno.sh, eintr.c

Als globale Variable wäre *errno* nicht *Thread-safe*, deshalb ist *errno* in *Pthreads* mit Makros definiert.

Pthreads definiert *errno* lokal zum laufenden Thread:
`# define errno (*__errno_location())`

Die Zuweisung z.B. *errno* = *EINTR* sieht dann so aus:
`*__errno_location() = 4; // nach Präprozessor`

So hat jeder Thread eine eigene *errno* Variable.

Threads und Prozesse im Vergleich

Austausch von Daten ist einfacher zwischen Threads.

Erzeugen & Kontext-Switch ist schneller bei Threads.

Aufgerufene Funktionen müssen *Thread-safe* sein.

Bugs, z.B. Seg. Fault, in einem Thread betrifft alle, Prozesse hingegen sind besser voneinander isoliert.

Threads müssen sich Prozess-Ressourcen teilen.

Thread Synchronization

Zwei Mechanismen um Threads zu Synchronisieren:

Ein *Mutex* schützt den Zugriff auf geteilte Ressourcen.

Zustandsvariablen (condition variables) erlauben es Threads, sich gegenseitig über Zustandsänderungen einer gemeinsam genutzten Ressource zu informieren.

Mutex und Critical Section

Threads teilen sich globale Variablen, das ist bequem, man muss aber auch aufpassen beim Zugriff darauf.

Critical Section nennt man einen Programmteil, der auf gemeinsam genutzte Variablen zugreift und nur atomar ausgeführt werden darf, ein Thread aufs Mal.

Ein *Mutex* garantiert den gegenseitigen Ausschluss, das ist wichtig, um Race Conditions zu verhindern.

n|w

Race Condition Beispiel thread_incr.c[TLPI]

Hier ein einfaches Beispiel einer Race Condition:

```
$ ./thread_incr 1000 # glob = 2000 (erwartet)
$ ./thread_incr 1000000 # z.B. glob = 1254665
```

Sogar mit *glob++* ist die Instruktion nicht atomar:

```
$ objdump -S --disassemble thread_incr
10894: e5933000 ldr r3, [r3]
10898: e2833001 add r3, r3, #1
```

Hier hilft ein *Mutex*: thread_incr_mutex.c[TLPI]

n|w

Mutex Sperre

Die *Mutex Sperre* (Lock) garantiert *mu*tual *ex*clusion.

Eine Mutex Sperre hat zwei Zustände, *zu* und *offen*.

Zu jedem Zeitpunkt hat genau ein Thread die Sperre.

Diesen Thread nennt man auch *Besitzer* des Mutex.

Weitere Versuche, die Sperre zu bekommen, werden blockiert, oder es gibt eine Fehlermeldung.

n|w

Mutex Ablauf

Pro geteilte Ressource braucht es eine Mutex Sperre:

```
Thread A                Thread B
lock mutex M                 |
      | (granted)       lock mutex M
      |             (blocked) |
access resource R            |
unlock mutex M ---> (granted) |
                       access resource R
(Freiwillig, "fair play")    unlock mutex M
```

n|w

Mutex Funktionen in *Pthread*

Mutex Variablen haben den Typ *pthread_mutex_t*, und müssen vor dem Gebrauch initialisiert werden:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Um eine Mutex Sperre (Lock) zu-/aufzuschliessen:

```
int pthread_mutex_lock(pthread_mutex_t *m);
int pthread_mutex_unlock(pthread_mutex_t *m);
```

Falls die Mutex Sperre zu ist, blockiert *_lock()*.

n|w

Verhalten von Mutex Funktionen

Wenn ein Thread versucht einen Mutex zu sperren, den er bereits gesperrt hat, gibt es einen *Deadlock*.

```
err = thread_mutex_lock(m); // 0, Success
err = thread_mutex_lock(m); // Deadlock (!)
```

_unlock() eines offenen/fremden Mutex => Fehler.

Wenn mehrere Threads auf dieselbe Sperre warten, kommt ein beliebiger Thread als nächstes dran.

n|w

Nicht-blockierende Mutex Funktionen

Die *_tryock()* und *_timedlock* Calls blockieren nicht:
`int pthread_mutex_trylock(pthread_mutex_t *m);`
gibt *EBUSY* zurück, falls Mutex *m* schon gesperrt ist.

Bei *_timedlock()* kommt nach Timeout *ETIMEDOUT*:

```
const struct timespec *t = {1, 0}; // 1s + 0ns
int pthread_mutex_timedlock(
  pthread_mutex_t *m,
  const struct timespec *t);
```

n|w

Zustandsvariablen

Eine *Zustandsvariable* (condition variable) erlaubt einem Thread, andere Threads über Änderungen des Zustands einer gemeinsam genutzten Ressource zu informieren, z.B. bei Verfügbarkeit eines Resultats.

Eine Zustandsvariable wird immer zusammen mit einem Mutex verwendet, der die Variable schützt.

n|w

Zustandsvariablen in *Pthread*

Zustandsvariablen haben den Typ *pthread_cond_t*, und müssen vor dem Gebrauch initialisiert werden:
`pthread_cond_t c = PTHREAD_COND_INITIALIZER;`

Eine Zustandsänderung abwarten bzw. signalisieren:

```
int pthread_cond_wait(pthread_cond_t *c,
pthread_mutex_t *m); // m muss locked sein
int pthread_cond_broadcast(pthread_cond_t *c);
int pthread_cond_signal(pthread_cond_t *c);
```

n|w

Zustandsänderung abwarten

Die *_wait()* Funktion wartet auf Eintreten von *cond*:
`int err = thread_cond_wait(cond, mutex);`

Im *wait()* wird der Mutex entsperrt, der Aufrufende Thread danach bis zum Eintreten des Zustands *cond* suspendiert, und erst dann *mutex* wieder gesperrt.

Der Aufrufende Thread läuft so nur noch wenn nötig.

Von *_wait()* gibt es eine *_timedwait()* Variante.

n|w

Zustandsänderung signalisieren

Wenn *_signal()* auf eine Zustandsvariable aufgerufen wird, wird zufällig ein wartender Thread geweckt.

Der *_broadcast()* Call weckt alle wartenden Threads.

Wenn alle Threads dieselbe Aufgabe haben, wird eher *_signal()* verwendet, sonst ist *_broadcast()* besser.

Der Name *_signal()* hat nichts zu tun mit Signals.

n|w

CPU schonen prod_no_condvar.c[TLPI]

Hier ein Beispiel ohne (bzw. mit) Zustandsvariable:

```
$ time prod_no_condvar 3 3 3 # user time (!)
```

Das Programm ist korrekt, verschwendet aber CPU-Zyklen, indem es den Mutex immer wieder (un)lockt, obwohl die Bedingung *avail* > 0 noch nicht erfüllt ist.

Die Producer-Threads wissen genau, wann *avail* > 0. Hier hilft eine Zustandsvariable: prod_condvar.c[TLPI]

n|w

Hands-on, 30': Producer/Consumer

Das Producer/Consumer Problem ist ein Klassiker der parallelen Programmierung - studieren Sie die Version (von Sun) mit zwei Zustandsvariablen: docs.oracle.com/cd/E19455-01/806-5257/sync-31

Und, falls Zeit bleibt, hier eine detaillierte Version: http://www.cs.fsu.edu/~baker/realtime/restricted/notes/prodcons.html mit einem zirkulären Buffer.

n|w

Selbststudium, 3h: Vorbereitung

Repetieren Sie Slides & Hands-on der Lektionen 1-6.

Das obligatorische Assessment dauert 2 Stunden.

Eine (mehrseitige) C-Referenzkarte ist erlaubt.

Die Karte soll keinen Beispielcode enthalten.

Das Assessment ist am 05.11., im Raum 1.045 (!)

n|w

Feedback?

Gerne im Slack oder an thomas.amberg@fhnw.ch

Programmierfragen am besten schriftlich.

Sprechstunde auf Voranmeldung :)

Slides, Code & Hands-on: tmb.gr/syspr-6