PThreads

Prozesse und Threads

Ein Unix-Prozess hat

- IDs (process,user,group)
- Umgebungsvariablen
- Verzeichnis
- Programmcode
- Register, Stack, Heap
- Dateideskriptoren, Signale
- message queues, pipes, shared memory Segmente
- Shared libraries

Jeder Prozess besitzt seinen eigenen Adressraum

Threads existieren innerhalb eines Prozesses

Threads teilen sich einen Adressraum

Ein Thread besteht aus

- ID
- Stack pointer
- Register
- Scheduling Eigenschaften
- Signale

Erzeugungs- und Umschaltzeiten sind kürzer

„Parallele Funktion“

Pthreads

- Jeder Hersteller hatte eine eigene Implementierung von Threads oder „light weight processes“
- 1995: IEEE POSIX 1003.1c Standard (es gibt mehrere “drafts“)
- Definiert Threads in portabler Weise
- Besteht aus C Datentypen und Funktionen
- Header file `pthread.h`
- Bibliotheksname nicht genormt. In Linux `-lpthread`
- Übersetzen in Linux: `gcc <file> -lpthread`

Pthreads Übersicht

Alle Namen beginnen mit `pthread_`

- `pthread_` Thread Verwaltung und sonstige Routinen
- `pthread_attr_` Thread Attributobjekte
- `pthread_mutex_` Alles was mit Mutexvariablen zu tun hat
- `pthread_mutex_attr_` Attribute für Mutexvariablen

Weitere Befehle werden wir später kennen lernen.

Erzeugen von Threads

- `pthread_t` : Datentyp für einen Thread.

- *Opaquer Typ: Datentyp wird in der Bibliothek definiert und wird von deren Funktionen bearbeitet. Inhalt ist implementierungsabhängig.*
- `int pthread_create(thread,attr,start_routine,arg)` : Startet die Funktion `start_routine` als Thread.
  - `thread` : Zeiger auf eine `pthread_t` Struktur. Dient zum identifizieren des Threads.
  - `attr` : Threadattribute besprechen wir später. Default ist `NULL`.
  - `start_routine` Zeiger auf eine Funktion vom Typ `void* func (void*);`
  - `arg` : `void*`-Zeiger der der Funktion als Argument mitgegeben wird.
  - Rückgabewert größer Null zeigt Fehler an.
- Threads können weitere Threads starten, maximale Zahl von Threads ist implementierungsabhängig

Beenden von Threads

- Es gibt folgende Möglichkeiten einen Thread zu beenden:
  - Der Thread beendet seine `start_routine()`
  - Der Thread ruft `pthread_exit()`
  - Der Thread wird von einem anderen Thread mittels `pthread_cancel()` beendet
  - Der Prozess wird durch `exit()` oder durch das Ender der `main()`-Funktion beendet
- `pthread_exit(void* status)`
  - Beendet den rufenden Thread. Zeiger wird gespeichert und kann mit `pthread_join` (s.u.) abgefragt werden (Rückgabe von Ergebnissen).
  - Falls `main()` diese Routine ruft so laufen existierende Threads weiter und der Prozess wird nicht beendet.
  - Schließt keine geöffneten Dateien!

Warten auf Threads

- Peer Modell: Mehrere gleichberechtigte Threads bearbeiten eine Aufgabe. Programm wird beendet wenn alle Threads fertig sind
- Erfordert Warten eines Threads bis alle anderen beendet sind
- Dies ist eine Form der Synchronisation
- `int pthread_join(pthread_t thread, void **status);`
  - Wartet bis der angegebene Thread sich beendet
  - Der Thread kann mittel `pthread_exit()` einen `void*`-Zeiger zurückgeben,
  - Gibt man `NULL` als Statusparameter, so verzichtet man auf den Rückgabewert

Thread Management Beispiel

```
#include <pthread.h>     /* for threads     */

void* prod (int *i) {    /* Producer thread */
  int count=0;
  while (count<100000) count++;
}

void* con (int *j) { /* Consumer thread */
  int count=0;
  while (count<1000000) count++;
}

int main (int argc, char *argv[]) { /* main program */
  pthread_t thread_p, thread_c; int i,j;

  i = 1;
  pthread_create(&thread_p, NULL, (void*(*)(void*)) prod, (void *) &i);
  j = 1;
  pthread_create(&thread_c, NULL, (void*(*)(void*)) con,  (void *) &j);

  pthread_join(thread_p, NULL); pthread_join(thread_c, NULL);
  return 0;
}
```

Übergeben von Argumenten

- Übergeben von mehreren Argumenten erfordert Definition eines eigenen Datentyps:

  ```
  struct argtype {int rank; int a,b; double c;};
  struct argtype args[P];
  pthread_t threads[P];

  for (i=0; i<P; i++) {
    args[i].rank=i; args[i].a=...
      pthread_create(threads+i,NULL,(void*(*)(void*)) prod,
                     (void *)args+i);
  }
  ```

- Folgendes Beispiel enthält zwei Fallstricke:

  ```
  pthread_t threads[P];
  for (i=0; i<P; i++) {
    pthread_create(threads+i,NULL,(void*(*)(void*)) prod,&i);
  ```

  - Inhalt von `i` ist möglicherweise verändert bevor Thread liest
  - Falls `i` eine Stackvariable ist exisitiert diese möglicherweise nicht mehr

Thread Identifiers

- `pthread_t pthread_self(void);` Liefert die eigene Thread-ID
- `int pthread_equal(pthread_t t1, pthread_t t2);` Liefert wahr (Wert>0) falls die zwei IDs identisch sind
- Konzept des „opaque data type“

Mutex Variablen

- Mutex Variablen realisieren den wechselseitigen Ausschluss innerhalb der Pthreads Bibliothek
- Erzeugen und initialisieren einer Mutex Variable

  ```
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex,NULL);
  ```

  Mutex Variable ist nach Initialisierung im Zustand frei
- Eintritt in den kritischen Abschnitt:

  ```
  pthread_mutex_lock(&mutex);
  ```

  Diese Funktion blockiert solange bis man drin ist.
- Verlasse kritischen Abschnitt

  ```
  pthread_mutex_unlock(&mutex);
  ```
- Gebe Resourcen der Mutex Variable wieder frei

  ```
  pthread_mutex_destroy(&mutex);
  ```

Bedingungsvariablen

- Bedingungsvariablen erlauben das *inaktive* Warten eines Prozesses bis eine gewisse Bedingung eingetreten ist
- Einfachstes Beispiel: Flaggenvariablen (siehe Beispiel unten)
- Zu einer Bedingungssynchronisation gehören *drei* Dinge:
  - Eine Variable vom Typ `pthread_cond_t`, die das inaktive Warten realisiert
  - Eine Variable vom Typ `pthread_mutex_t`, die den wechselseitigen Ausschluss beim Ändern der Bedingung realisiert
  - Eine globale Variable, deren Wert die Berechnung der Bedingung erlaubt

Erzeugen/Löschen

- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);` initialisiert eine Bedingungsvariable

  Im einfachsten Fall: `pthread_cond_init(&cond,NULL)`
- `int pthread_cond_destroy(pthread_cond_t *cond);` gibt die Resourcen einer Bedingungsvariablen wieder frei

Wait

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);` blockiert den aufrufenden Thread bis für die Bedingungsvariable die Funktion `pthread_signal()` aufgerufen wird
- Beim Aufruf von `pthread_wait()` muss der Thread auch im Besitz des Locks sein
- `pthread_wait()` verlässt das Lock und wartet auf das Signal in atomarer Weise
- Nach der Rückkehr aus `pthread_wait()` ist der Thread wieder im Besitz des Locks
- Nach Rückkehr muss die Bedingung nicht unbeding erfüllt sein
- Mit einer Bedingungsvariablen sollte man nur genau ein Lock verwenden

Signal

- `int pthread_cond_signal(pthread_cond_t *cond);` Weckt einen Prozess der auf der Bedingungsvariablen ein `pthread_wait()` ausgeführt hat. Falls keiner wartet hat die Funktion keinen Effekt.
- Beim Aufruf sollte der Prozess im Besitz des zugehörigen Locks sein
- Nach dem Aufruf sollte das Lock freigegeben werden. Erst die Freigabe des Locks erlaubt es dem wartenden Prozess aus der `pthread_wait()` Funktion zurückzukehren
- `int pthread_cond_broadcast(pthread_cond_t *cond);` weckt *alle* Threads die auf der Bedingungsvariablen ein `pthread_wait()` ausgeführt haben. Diese bewerben sich dann um das Lock.

Beispiel I

```
#include<stdio.h>
#include<pthread.h>     /* for threads     */

int arrived_flag=0,continue_flag=0;
pthread_mutex_t arrived_mutex, continue_mutex;
pthread_cond_t arrived_cond, continue_cond;

pthread_attr_t attr;

int main (int argc, char *argv[])
{
  pthread_t thread_p, thread_c;

  pthread_mutex_init(&arrived_mutex,NULL);
  pthread_cond_init(&arrived_cond,NULL);
  pthread_mutex_init(&continue_mutex,NULL);
  pthread_cond_init(&continue_cond,NULL);
```

Beispiel II

```
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  pthread_create(&thread_p,&attr,(void*(*)(void*)) prod,NULL);
  pthread_create(&thread_c,&attr,(void*(*)(void*)) con ,NULL);

  pthread_join(thread_p, NULL);
  pthread_join(thread_c, NULL);

  pthread_attr_destroy(&attr);

  pthread_cond_destroy(&arrived_cond);
  pthread_mutex_destroy(&arrived_mutex);
  pthread_cond_destroy(&continue_cond);
  pthread_mutex_destroy(&continue_mutex);

  return(0);
}
```

Beispiel III

```
void prod (void* p) /* Producer thread */
{
  int i;
  for (i=0; i<100; i++) {
    printf("ping\n");

    pthread_mutex_lock(&arrived_mutex);
    arrived_flag = 1;
    pthread_cond_signal(&arrived_cond);
    pthread_mutex_unlock(&arrived_mutex);
```

```
    pthread_mutex_lock(&continue_mutex);
    while (continue_flag==0)
    pthread_cond_wait(&continue_cond,&continue_mutex);
    continue_flag = 0;
    pthread_mutex_unlock(&continue_mutex);
  }
}
```

Beispiel IV

```
void con (void* p) /* Consumer thread */
{
  int i;
  for (i=0; i<100; i++) {
    pthread_mutex_lock(&arrived_mutex);
    while (arrived_flag==0)
      pthread_cond_wait(&arrived_cond,&arrived_mutex);
    arrived_flag = 0;
    pthread_mutex_unlock(&arrived_mutex);

    printf("pong\n");

    pthread_mutex_lock(&continue_mutex);
    continue_flag = 1;
    pthread_cond_signal(&continue_cond);
    pthread_mutex_unlock(&continue_mutex);
  }
}
```

Thread Safety

- Darunter versteht man ob eine Funktion/Bibliothek von mehreren Threads gleichzeitig genutzt werden kann.
- Eine Funktion ist *reentrant* falls sie von mehreren Threads gleichzeitig gerufen werden kann.
- Eine Funktionen, die keine globalen Variablen benutzt ist reentrant
- Das Laufzeitsystem muss gemeinsam benutzte Resourcen (z.B. den Stack) unter wechselseitigem Ausschluss bearbeiten
- Der GNU C Compiler muss beim Übersetzen mit einem geeigneten Thread-Modell konfiguriert werden. Mit `gcc -v` erfährt man das Thread-Modell

Links

1 PThreads tutorial vom LLNL `https://computing.llnl.gov/tutorials/pthreads/`

2 Noch ein Tutorial `http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`