Kapitel 5

threads

Unter einem thread versteht die Folge von Anweisungen (Adressen im Programmspeicher) die einem Programm entsprechen. Im Normalfall ist dies ein gleichbleibender Strom von Anweisungen. Dies gilt bei Betrachtung vom Standpunkt eines Programms aus. Vom Processor her gilt dies nicht mehr es finden Anweisungen von verschiedenen Processen statt, dazwischen sind die context switches. Dies ist eine relativ aufwendige Operation. Die Idee ist es innerhalb eines Processes mehrere threads zuzulassen. Sie teilen sich Daten und Programmspeicher und es ist nicht der context switch zwischen Processen nötig. Sollten derartige multi thread Programme auf Mehrprocessor Rechnern laufen, haben sie einen deutlichen Geschwindigkeitsvorteil gegenüber Programmen die Parallelität mittels mehreren Processen erreichen. Man spricht bei solcher Hardware von symmetrischen Multiprocessing.

5.1 Begriffe

Zu einem thread gehören ein eigener Stack, ein Befehlszähler, Register und Flags und der Zustand. Threads teilen sich Programm und Datenspeicher. Auf diese Weise wird Parallelität mit wenig overhead erreicht. Komplizierter wird die Synchronisation. Nachfolgend wird die Erzeugung/Verwaltung von threads beschrieben. Der zugrunde liegende Standard ist Posix 1c und wurde im Juni 1995 verabschiedet. Es gibt weitere thread Implementation, z.B. unter Solaris 2. Ein Vergleich der Routinen:

	POSIX	Solaris
thread management	pthread_create	thr_create
	pthread_exit	thr_exit
	pthread_kill	thr_kill
	pthread_self	thr_self

Die Routinen werden nun genauer untersucht:

5.2 POSIX thread management

Ein POSIX thread hat eine id (ähnlich der PID eines Processes), die der thread durch den Aufruf von pthread_self herausbekommt.

```
NAME
pthread_self - return identifier of current thread
SYNOPSIS
#include <pthread.h>
pthread_t pthread_self(void);
```

Zu einem thread gehören neben id, der Stack, eine Priorität, und eine Startadresse für den Programmzähler. Threads, die jederzeit während des Processes erzeugt werden k"onnen heissen dynamisch. Die andere Variante wäre, dass die mögliche Anzahl der threads vorher festgelegt ist (z.B. = Anzahl der Processoren). Die POSIX threads sind dynamisch und werden mittels pthread_create erzeugt:

```
NAME
  pthread_create - create a new thread

SYNOPSIS
  #include <pthread.h>
  int pthread_create(pthread_t * thread, pthread_attr_t
* attr,
                 void * (*start_routine)(void *), void *
arg);
```

Damit wird ein thread erzeugt und in die ready-queue gestellt. tid ist die id des erzeugten threads. Ist der Parameter attr NULL, so erhält der neue thread die default Attribute. start_routine ist die Routine die vom neuen thread am Start aufgerufen wird. Diese Routine hat einen Parameter, nämlich arg. Dazu ein Beispiel:

```
#include <pthread.h>
int zeit=0;
void * startroutine(void * arg)
{
int i;
i = rand()%5;zeit += i;
sleep(i);
printf("%ld:1\n", (long)pthread_self());
i = rand()%5;zeit += i;
sleep(i); printf("%ld:2\n", (long)pthread_self());
i = rand()%5;zeit += i;
sleep(i); printf("%ld:3\n", (long)pthread_self());
}
main()
{
pthread_t id1,id2,id3;
void *res;
pthread_create(&id1,NULL,startroutine,NULL);
printf("thread 1 = %ld erzeugt\n", (long)id1);
pthread_create(&id2,NULL,startroutine,NULL);
printf("thread 2 = %ld erzeugt\n", (long)id2);
pthread_create(&id3,NULL,startroutine,NULL);
printf("thread 3 = %ld erzeugt\n", (long)id3);
pthread_join(id1, &res);
pthread_join(id2, &res);
pthread_join(id3, &res);
printf("zeit = %d\n",zeit);
}
```

Zum Kompilieren muss die POSIX thread library mit angegeben werden:

```
cc t1.c -lpthread
```

und die Ausgabe beii einem times a.out:

```
thread 1 = 4 erzeugt
thread 2 = 5 erzeugt
thread 3 = 6 erzeugt
5:1
5:2
4:1
6:1
6:2
5:3
4:2
6:3
4:3
zeit = 18

real 7.1
...
```

Dabei wird die Routine pthread_join verwendet:

```
NAME
pthread_join, thr_join - wait for thread termination

SYNOPSIS
POSIX
cc [ flag ... ] file ... -lpthread [ library ... ]

#include <pthread.h>
int pthread_join(pthread_t target_thread, void
**status);
```

die auf das Ende eines threads wartet. Der erste Parameter ist die id, des threads auf den gewartet wird, der zweite Parameter ist die Rückgabe der Startroutine. Man sieht auch, dass während des sleep Befehls die anderen threads arbeiten, ferner sieht man auch den gemeinsamen Datenbereich mit der globalen Variable zeit. Um wirklich den Vorteil des Multithreadings auf Mehrprocessorsystemen zu sehen, muss natürlich in den threads gerechnet werden:

```
#include <pthread.h>
int gg;
void * startroutine(void * arg)
{
int i;
for (i=1;i<100000000;i++)gg*=i;
}
main()
{
pthread_t id1,id2,id3;
void *res;
pthread_create(&id1,NULL,startroutine,NULL);
printf("thread 1 = %ld erzeugt\n", (long)id1);
pthread_create(&id2,NULL,startroutine,NULL);
printf("thread 2 = %ld erzeugt\n", (long)id2);
pthread_create(&id3,NULL,startroutine,NULL);
printf("thread 3 = %ld erzeugt\n", (long)id3);
pthread_join(id1, &res);
pthread_join(id2, &res);
pthread_join(id3, &res);
}
```

und nun z.B. auf einem 3 Prozessor Rechner:

```
thread 1 = 1026 erzeugt
thread 2 = 2051 erzeugt
thread 3 = 3076 erzeugt


real 0m0.532s
user 0m1.570s
sys 0m0.000s
```

Eine weitere Routine ist pthread_exit:

```
NAME
pthread_exit - Terminates the calling thread.

LIBRARY
DECthreads POSIX 1003.1c Library (libpthread.so)

SYNOPSIS
#include <pthread.h>
void pthread_exit( void *value_ptr);
```

damit wird der aktuelle thread beendet, der Parameter ist der Wert, den ein eventuell wartendes pthread_join erhält. Nützlich ist dies, wenn man direkt aus einer Unterrou-

tine der startroutine heraus den thread beenden will, innerhalb der startroutine genügt ein return. Ein weiterer Unterschied zwischen exit handler und return ist das Ausführen sog. exit handler, was bei return nicht passiert. Beim Rückgabewert muss man aufpassen, es muss natürlich ein auch ausserhalb gültiger Speicherbereich sein. Ferner muss man bei den Routinen, die innerhalb der threads verwendet werden aufpassen, dass sie multi thread safe sind.

5.3 POSIX thread Attribute

Da Threads ein relativ neuer Bestandteil des UNIX Systems (und anderer POSIX kompatibler Systeme) ist wurde ein objektorientierter Ansatz gewählt. Ein Attribut Objekt kann erzeugt werden, variert und zerstört werden. Ein Attribut Objekt kann auch mehreren Threads zugeordnet werden, sodass eine Änderung an diesem Objekt alle Threads modifiziert. Das Attribut Objekt ist vom Typ pthread_attr_t. Nachfolgend eine Tabelle der Attribute:

Eigenschaft	Funktion
Erzeugen	pthread_attr_init
Löschen	pthread_attr_destroy
Stackgrösse	pthread_attr_setstacksize
	pthread_attr_getstacksize
Stackadresse	pthread_attr_setstackaddr
	pthread_attr_getstackaddr
Detachstate	pthread_attr_setdetachstate
	pthread_attr_getdetachstate
Scope	pthread_attr_setscope
	pthread_attr_getscope
Vererbung der	pthread_attr_getinheritsched
Scheduling Parameter	pthread_attr_setinheritsched
Scheduling Methode	pthread_attr_setschedpolicy
	pthread_attr_getschedpolicy
Scheduling Parameter	pthread_attr_setschedparam
	pthread_attr_getschedparam

Die ersten beiden Funktion sind zum Erzeugen und Löschen:

```
NAME
pthread_attr_init, pthread_attr_destroy

SYNOPSIS
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

init erzeugt ein Attribut Objekt mit den default Parametern. destroy macht das entsprechende Objekt ungültig, d.h. eine weitere Verwendung führt zu einem Fehler. Die anderen Funktionen haben immer zwei Parameter, als erstes das Objekt und nachfolgende eine Struktur für die entsprechenden Parameter, so z.B.

```
NAME
pthread_attr_setschedpolicy pthread_attr_getschedpolicy
pthread_attr_setschedparam pthread_attr_getschedparam

SYNOPSIS
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr,
          int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t
*attr,
          int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr,
          const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t
*attr,
          struct sched_param *param);
```

Nun eine Beschreibung der einzelnen Attribute:

- detachstate
  Damit kann eingestellt werden ob auf den thread mit join gewartet werden kann: PTHREAD_CREATE_JOINABLE oder nicht
  PTHREAD_CREATE_DETACHED. Default ist ersteres. Nachteil beim join, ist, dass von einem beendeten thread mehr Informationen (Rückgabewert und das Thread Objekt) im Speicher bleiben, bis der join erfolgt.

- schedpolicy
  Damit kann die Scheduling Methode für die Threads eingestellt werden, es gibt SCHED_OTHER (default) , SCHED_FIFO , SCHED_RR (round robin). Es kann sein, dass die letzten beiden nur für super user Processe (PUID == 0) zur Verfügung stehen (linux). Bei Linux ist dies, da dies realtime scheduling Methoden sind, die auf kernel level konkurrieren. Dies zeigt, dass es sich bei den LINUX Threads um kernel threads handelt. Anders ist dies bei user level threads, dann findet ein eigenes scheduling innerhalb des processes statt, und es gibt keine Notwendigkeit Einschränkungen bei den Zugriffsrechten zu machen. Die Scheduling Methode wird z.B. bei IRIX exakt beschrieben. Das Scheduling erfolgt immer anhand von Prioritäten, der Unterschied zwischen Fifo und Round Robin, ist, dass bei Round Robin automatisch nach einer gewissen Zeit die Priorität herab gesetzt wird.

- schedparam
  Dies ist im wesentlichen die Priority für das Scheduling. Default ist 0.

- inheritsched
  Damit kann eingestellt werden ob die gleiche Scheduling Methode beibehalten wird
  (PTHREAD _INHERIT_SCHED) oder aber eine eigene (kann auch die default Methode sein) verwendet wird
  (PTHREAD_EXPLICIT_SCHED). Default ist Plattformabhängig.

- scope
  Damit wird eingestellt ob nur innerhalb des Processes (PTHREAD_SCOPE_PROCESS) oder innerhalb des gesamten Systems (PTHREAD_SCOPE_SYSTEM) um CPU Zeit gekämpft wird. Default ist Plattformabhängig da dies z.B mit ein Unterschied zwischen kernel threads und user threads ist. LINUX = PTHREAD_SCOPE_SYSTEM, IRIX = PTHREAD_SCOPE_PROCESS.

- stacksize,stackaddr
  Wird ein eigener Stack vor dem Erzeugen des Threads mit diesen beiden Parametern zur Verfügung gestellt, so wird dieser auch nicht am Ende automatisch gelöscht. Default ist Plattformabhängig. Nützlich ist dies wenn man genau weiss wie gross der Stack sein muss.

Noch ein Beispiel, wie man eigenes Scheduling erreicht. Das Beispiel ist an eine IRIX 2-Processor Maschine angepasst.

```
#include <pthread.h>
#include <sched.h>

#define ANZ 4
#define LOOP 20000000
void * startroutine(void * arg)
{
int i,j;
for (j=1;j<=3;j++) {
for (i=0;i<LOOP;i++);
printf("%ld:%d\n", (long)pthread_self(),j);}
}

pthread_t id[ANZ*2];
pthread_attr_t a[ANZ*2];

startthread(int i,int m) {
  struct sched_param param;
  pthread_attr_init(a+i);

  pthread_attr_setinheritsched(a+i, PTHREAD_EXPLICIT_SCHED);
  if (m < 0)
    param.sched_priority = sched_get_priority_min(SCHED_OTHER);
  else
    param.sched_priority = sched_get_priority_max(SCHED_OTHER);
  pthread_attr_setschedparam(a+i, &param);


  pthread_create(id+i,a+i,startroutine,NULL);
  printf("%s thread %d = %ld erzeugt\n", (m < 0 ? "min":
"max"), i,(long)id[i]);
}

main() {
void *res;
int i;
for (i=0;i<ANZ*2;i++)
if (i%2)
  startthread(i,1);
else
  startthread(i,-1);

for (i=0;i<ANZ*2;i++)
pthread_join(id[i], &res);
}
```

Es wurden einige Anpassungen an IRIX vorgenommen um die Werte für min und max bei der Priorität herauszubekommen, angepasst wurde das Programm an die 4 Prozessoren, die man interaktiv auf der 12 Prozessormaschine btrzxe zur Verfügung

hat.

```
min thread 0 = 65537 erzeugt max thread 1 = 65538 erzeugt
min thread 2 = 65539 erzeugt max thread 3 = 65540 erzeugt
65540:1 65538:1
min thread 4 = 65541 erzeugt max thread 5 = 65542 erzeugt
min thread 6 = 65543 erzeugt max thread 7 = 65544 erzeugt
65538:2 65544:1 65537:1 65540:2 65541:1 65539:1 65543:1 65542:1
65538:3 65544:2 65537:2 65540:3 65543:2 65539:2 65541:2 65542:2
65544:3 65537:3 65542:3 65543:3 65539:3 65541:3
```

5.4 Mutexes

Mit den Mutexes (Mutual Exclusions) kann man sensitive Daten, auf die mehrere Threads zugreifen, schützen. Ohne dieses Hilfsmittel geht dies z.B. auch mit Semaphoren. Das Prinzip ist denkbar einfach. Die Pthreads-Bibliothek stellt Mutex-Variablen bereitet, die Threads sperren bzw. entsperren können. Zwischen dem Sperr- und Entsperrvorgang werden dann Änderungen an den sensitiven Daten durchgeführt. Versucht ein Thread eine schon gesperrte Variable zu sperren, so wird er scheitern und muss so lange warten, bis der Mutex wieder frei ist. Dadurch ist es nicht möglich, dass zwei Threads gleichzeitig eine Mutex Variable sperren. Betrachten wir die wichtigsten Befehle für Mutexes:

```
NAME
pthread_mutex_init

SYNOPSIS
#include <pthread.h>

pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex,
            const pthread_mutexattr_t *mutexattr);
```

Eine Mutex-Variable wird entweder durch pthread_mutex_init oder gleich bei der Definition mit Hilfe von PTHREAD_MUTEX_INITIALIZER initialisiert. Nach erledigter Arbeit kann man per pthread_mutex_destroy die Mutex-Variable freigeben, sie aus dem Speicher entfernen.

```
NAME
pthread_mutex_destroy

SYNOPSIS
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Wie bei Semaphoren gibt es die Funktionen um die resource zu schützen, sie heissen lock, trylock und unlock.

```
NAME
pthread_mutex_lock,pthread_mutex_trylock
pthread_mutex_unlock

SYNOPSIS
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Der Vollständigkeit halber seien zwei ganz ähnliche Funktionen für Variablen, mit denen man die Attribute der Mutexes steuern kann, erwähnt:

```
NAME
pthread_mutexattr_destroy,pthread_mutexattr_init

SYNOPSIS
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t
*attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

Achtung! Nicht alle Implementationen enthalten Attributsvariablen für Mutexes, z.B. existieren diese unter AIX nicht. Das nächste Beispiel veranschaulicht den Einsatz von Mutexes. Es werden zwei Threads gestartet, der erste wird zu einer global definierten Variable sum 1000000 mal eine eins hinzuaddieren, der zweite abziehen. Da die Threads gleichzeitig laufen, kann es passieren, dass sie im gleichen Moment den Variablenwert zu verändern versuchen. Die Konsequenzen sind klar: Es wird sich nur einer der beiden Methoden durchsetzen, uns geht also eine Veränderung verloren! Um das zu verhindern, werden wir den Zugriff auf die Variable mit Hilfe einer Mutex schützen.

```
#include <pthread.h>
/* Hier wird die Mutex-Variable definiert */
pthread_mutex_t mutex_sum = PTHREAD_MUTEX_INITIALIZER;
int sum = 100;
void *minus(void *arg)
{
int i = 0;
for(i=0; i < 1000000; i++)
   {
   pthread_mutex_lock(&mutex_sum);
   random(); //damits etwas länger dauert
   sum = sum - 1;
   pthread_mutex_unlock(&mutex_sum);
   }
}
void *plus(void *arg)
{
int i = 0;
for(i=0; i < 1000000; i++)
   {
   pthread_mutex_lock(&mutex_sum);
   random();
   sum = sum + 1;
   pthread_mutex_unlock(&mutex_sum);
   }
}
// ...
int main() {
// Definition der Variablen
pthread_t thread[2];
pthread_attr_t attr;
int i, status, err;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

/* Threads werden getstartet - auf Fehlerabfrage wurde
verzichtet */
pthread_create(&thread[0], &attr, minus, (void *)
NULL);
pthread_create(&thread[1], &attr, plus, (void *) NULL);
/* Warte auf alle Threads */
for(i=0; i < 2; i++) {
err = pthread_join(thread[i], (void **)&status);
if (err) printf("No join...\n");
}
printf("Summe : %d\n",sum); }
```

Die Ausgabe liefert das erwartete Ergebnis : Summe : 100 Lässt man die Mutex-Variablen weg, so kann das Ergebnis haarsträubende Werte annehmen. Da die Threads

ständig in einen Wartezustand gesetzt werden, sobald sie auf eine Mutex-Variable warten müssen, wird sich die Laufzeit des Programms umso mehr verlängern je öfter ein Thread warten muss. Daher ist es ratsam die Zahl der lock-Aufrufe zu minimieren. Im obigen Beispiel hätte man die Aufrufe z.B. ausserhalb der Schleife setzen können.