Parallele Programmiermodelle III

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 13/14

Parallele Programmiermodelle III

Kommunikation über gemeinsamen Speicher

- Philosophenproblem
- Leser-Schreiber-Problem
- PThreads
- Aktive Objekte

Semaphore

Eine Semaphore ist eine Abstraktion einer Synchronisationsvariable, die die elegante Lösung einer Vielzahl von Synchronisationsproblemen erlaubt

Alle bisherigen Programme haben *aktives Warten* verwendet. Dies ist sehr ineffizient bei quasi-paralleler Abarbeitung mehrerer Prozesse auf einem Prozessor (multitasking). Die Semaphore erlaubt es Prozesse in den Wartezustand zu versetzen.

Wir verstehen eine Semaphore als abstrakten Datentyp: Datenstruktur mit Operationen, die gewisse Eigenschaften erfüllen:

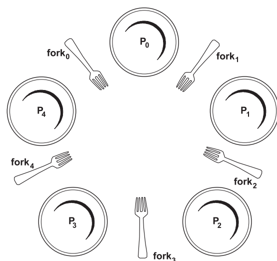
Eine Semaphore S hat einen ganzzahligen, nichtnegativen Wert *value*(S), der beim Anlegen der Semaphore mit dem Wert *init* belegt wird.

Auf einer Semaphore S sind zwei Operationen **P**(S) und **V**(S) definiert mit:

- **P**(S) erniedrigt den Wert von S um eins falls *value*(S) > 0, sonst *blockiert* der Prozess solange bis ein anderer Prozess eine **V**–Operation auf S ausführt.
- **V**(S) befreit einen anderen Prozess aus seiner **P**–Operation falls einer wartet (warten mehrere wird einer ausgewählt), ansonsten wird der Wert von S um eins erhöht. **V**–Operationen blockieren nie!

Speisende Philosophen

Komplexere Synchronisationsaufgabe: Ein Prozess benötigt exklusiven Zugriff auf mehrere Resourcen um eine Aufgabe durchführen zu können. → Überlappende kritische Abschnitte.



Fünf Philosophen sitzen an einem runden Tisch. Die Tätigkeit jedes Philosophen besteht aus den sich abwechselnden Phasen des Denkens und des Essens. Zwischen je zwei Philosophen liegt eine Gabel und in der Mitte steht ein Berg Spaghetti. Zum Essen benötigt ein Philosoph zwei Gabeln – die links und *rechts von ihm liegende.*

Speisende Philosophen

Das Problem:
Schreibe ein paralleles Programm, mit einem Prozess pro Philosoph, welches

- einer maximalen Zahl von Philosophen zu Essen erlaubt und
- das eine Verklemmung vermeidet

Grundgerüst eines Philosophen:

```
while (1)
{
    Denke;
    Nehme Gabeln;
    Esse;
    Lege Gabeln zurück;
}
```

Naive Philosophen

Programm (Naive Lösung des Philosophenproblems)

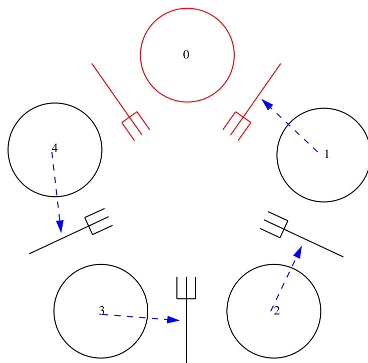
```
parallel philosphers–1
{
    const int P = 5;                              // Anzahl Philosophen
    Semaphore forks[P] = { 1 [P] };               // Gabeln

    process Philosopher [int p ∈ {0, ..., P − 1}] {
        while (1) {
            Denke;
            P(fork[p]);                           // linke Gabel
            P(fork[(p + 1) mod P]);               // rechte Gabel
            Esse;
            V(fork[p]);                           // linke Gabel
            V(fork[(p + 1) mod P]);               // rechte Gabel
        }
    }
}
```

Naive Philosophen

Philosophen sind verklemmt, falls alle zuerst die rechte Gabel nehmen!

Einfache Lösung des Deadlockproblems: Vermeide zyklische Abhängigkeiten, z. B. dadurch, dass der Philosoph 0 seine Gabeln in der anderen Reihenfolge links/rechts nimmt.

Diese Lösung führt eventuell nicht zu maximaler Parallelität:



Schlaue Philosophen

Nehme Gabeln nur wenn *beide* frei sind

Kritischer Abschnitt: nur einer kann Gabeln manipulieren

Drei Zustände eines Philosophen: denkend, hungrig, essend

Programm (Lösung des Philosophenproblems)

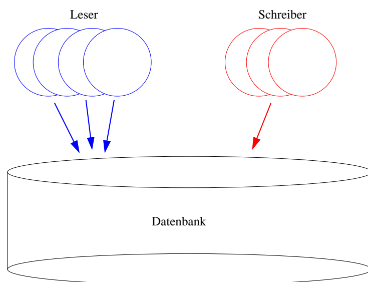
```
parallel philosphers–2
{
    const int P = 5;                          // Anzahl Philosophen
    const int think=0, hungry=1, eat=2;
    Semaphore mutex=1;
    Semaphore s[P] = { 0 [P] };              // essender Philosoph
    int state[P] = { think [P] };            // Zustand


}
```

Schlaue Philosophen

Programm (Lösung des Philosophenproblems)

```
parallel process
{
    Philosopher [int p ∈ {0, ..., P − 1}] {
        void test (int i) {
            int r=(i + P − 1) mod P, l=(i + 1) mod P;
            if (state[i]==hungry ∧ state[l]≠eat ∧ state[r]≠eat)
            {
                state[i] = eat;
                V(s[i]);
            }
        }

        while (1) {
            Denke;
            P(mutex);                          // Gabeln nehmen
            state[p] = hungry;
            test(p);
            V(mutex);
            P(s[p]);
            Esse;
            P(mutex);                          // Gabeln weglegen
            state[p] = think;
            test((p + P − 1) mod P);
            test((p + 1) mod P);
            V(mutex);
        }
    }
}
```

Leser/Schreiber Problem



Zwei Klassen von Prozessen, Leser und Schreiber, greifen auf eine gemeinsame Datenbank zu. Leser führen Transaktionen aus, die die Datenbank nicht verändern. Schreiber verändern die Datenbank und benötigen exklusiven Zugriff. Falls kein Schreiber Zugriff hat können beliebig viele Leser gleichzeitig zugreifen.

Probleme:

- Verklemmungsfreie Koordination der Prozesse
- Fairness: Schließliches Eintreten der Schreiber

Naive Leser/Schreiber

Zwei Semaphore:

- *rw*: Wer hat Zugriff auf Datenbank die Leser/der Schreiber
- *mutexR*: Absicherung des Schreiberzählers *nr*

Programm (Leser–Schreiber–Problem, erste Lösung)

```
parallel readers–writers–1
{
    const int m = 8, n = 4;              // Anzahl Leser und Schreiber
    Semaphore rw=1;                      // Zugriff auf Datenbank
    Semaphore mutexR=1;                  // Anzahl Leser absichern
    int nr=0;                            // Anzahl zugreifender Leser

    process Reader [int i ∈ {0, ..., m − 1}] {
        while (1) {
            P(mutexR);                   // Zugriff Leserzähler
            nr = nr+1;                   // Ein Leser mehr
            if (nr==1) P(rw);            // Erster wartet auf DB
            V(mutexR);                   // nächster Leser kann rein
            lese Datenbank;
            P(mutexR);                   // Zugriff Leserzähler
            nr = nr-1;                   // Ein Leser mehr
            if (nr==0) V(rw);            // Letzter gibt DB frei
            V(mutexR);                   // nächster Leser kann rein
        }
    }

}
```

Naive Leser/Schreiber

Programm (Leser–Schreiber–Problem, erste Lösung cont.)

parallel process
{
  Writer [**int** $j \in \{0, \ldots, n-1\}$] {
    while *(1)* {
      P*(rw);*   *// Zugriff auf DB*
      schreibe Datenbank;
      V*(rw);*   *// gebe DB frei*
    }
  }
}

Lösung ist nicht fair: Schreiber können verhungern

Faire Leser/Schreiber

Bearbeite wartende Prozesse nach FCFS in einer Warteschlange

Variable:

- *nr*, *nw*: Anzahl der *aktiven* Leser/Schreiber ($nw \leq 1$)
- *dr*, *dw*: Anzahl *wartender* Leser/Schreiber
- *buf*, *front*, *rear*: Warteschlange
- Semaphore *e*: Absichern des Zustandes/der Warteschlange
- Semaphore *r*, *w*: Warten der Leser/Schreiber

Programm (Leser–Schreiber–Problem, faire Lösung)

```
parallel readers–writers–2
{
    const int m = 8, n = 4;              // Anzahl Leser und Schreiber
    int nr=0, nw=0, dr=0, dw=0;          // Zustand
    Semaphore e=1;                       // Zugriff auf Warteschlange
    Semaphore r=0;                       // Verzögern der Leser
    Semaphore w=0;                       // Verzögern der Schreiber
    const int reader=1, writer=2;        // Marken
    int buf[n + m];                      // Wer wartet?
    int front=0, rear=0;                 // Zeiger



}
```

Faire Leser/Schreiber

Programm (Leser–Schreiber–Problem, faire Lösung cont1.)

```
parallel readers–writers–2 cont1.
{
    int wake_up (void)                              // darf genau einer ausführen
    {
        if (nw==0 ∧ dr>0 ∧ buf[rear]==reader)
        {
            dr = dr-1;
            rear = (rear+1) mod (n + m);
            V(r);
            return 1;                               // habe einen Leser geweckt
        }
        if (nw==0 ∧ nr==0 ∧ dw>0 ∧ buf[rear]==writer)
        {
            dw = dw-1;
            rear = (rear+1) mod (n + m);
            V(w);
            return 1;                               // habe einen Schreiber geweckt
        }
        return 0;                                   // habe keinen geweckt
    }

}
```

Faire Leser/Schreiber

Programm (Leser–Schreiber–Problem, faire Lösung cont2.)

```
parallel readers–writers–2 cont2.
{

    process Reader [int i ∈ {0, ..., m − 1}]
    {
        while (1)
        {
            P(e);                               // will Zustand verändern
            if(nw>0 ∨ dw>0)
            {
                buf[front] = reader;            // in Warteschlange
                front = (front+1) mod (n + m);
                dr = dr+1;
                V(e);                           // Zustand freigeben
                P(r);                           // warte bis Leser dran sind
                                                // hier ist e = 0 !
            }
            nr = nr+1;                          // hier ist nur einer
            if (wake_up()==0)                   // kann einer geweckt werden?
                V(e);                           // nein, setze e = 1

            lese Datenbank;

            P(e);                               // will Zustand verändern
            nr = nr-1;
            if (wake_up()==0)                   // kann einer geweckt werden?
                V(e);                           // nein, setze e = 1
        }
    }
```

Faire Leser/Schreiber

Programm (Leser–Schreiber–Problem, faire Lösung cont3.)

```
parallel readers–writers–2 cont3.
{

    process Writer [int j ∈ {0, ..., n − 1}]
    {
        while (1)
        {
            P(e);                                  // will Zustand verändern
            if(nr>0 ∨ nw>0)
            {
                buf[front] = writer;               // in Warteschlange
                front = (front+1) mod (n + m);
                dw = dw+1;
                V(e);                              // Zustand freigeben
                P(w);                              // warte bis an der Reihe
                                                   // hier ist e = 0 !
            }
            nw = nw+1;                             // hier ist nur einer
            V(e);                                  // hier braucht keiner geweckt werden

            schreibe Datenbank;                    // exklusiver Zugriff

            P(e);                                  // will Zustand verändern
            nw = nw-1;
            if (wake_up()==0)                      // kann einer geweckt werden?
                V(e);                              // nein, setze e = 1
        }
    }
}
```

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“)
- Standard Dokument ist kostenpflichtig
- 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
- `pthread_cond_`
  Bedingungsvariablen (condition variables)
- `pthread_cond_attr_`
  Attribute für Bedingungsvariablen

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 unten. 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 Ende 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 Fehler:

```
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“

Join/Detach

- Ein Thread im Zustand `PTHREAD_CREATE_JOINABLE` gibt seine Resourcen erst frei, wenn `pthread_join` ausgeführt wird.
- Ein Thread im Zustand `PTHREAD_CREATE_DETACHED` gibt seine Resourcen frei sobald er beendet wird. In diesem Fall ist `pthread_join` nicht erlaubt.
- Default ist `PTHREAD_CREATE_JOINABLE`, das implementieren aber nicht alle Bibliotheken.
- Deshalb besser:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
int rc = pthread_create(&t,&attr,(void*(*)(void*))func,NULL);
....
pthread_join(&t,NULL);
pthread_attr_destroy(&attr);
```

- Gibt Beispiel für die Verwendung von Attributen

Mutex Variablen

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

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

  Mutex Variable ist im Zustand frei
- Versuche in den kritischen Abschnitt einzutreten (blockierend):

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

Bedingungsvariablen: 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

Bedingungsvariablen: 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 unbedingt erfüllt sein
- Mit einer Bedingungsvariablen sollte man nur genau ein Lock verwenden

Bedingungsvariablen: 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.

Bedingungsvariablen: Ping-Pong Beispiel

```
#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 cont. I

```
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 cont. II

```
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 cont. III

```
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 Funktion, 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
- STL: Allokieren ist threadsicher, Zugriff mehrerer Threads auf einen Container muss vom Benutzer abgesichert werden.

Threads und OO

- Offensichtlich sind Pthreads relativ unpraktisch zu programmieren
- Mutexes, Bedingungsvariablen, Flags und Semaphore sollten objektorientiert realisiert werden. Umständliche `init/destroy` Aufrufe können in Konstruktoren/Destruktoren versteckt werden
- Threads werden in Aktive Objekte umgesetzt
- Ein Aktives Objekt „läuft“ unabhängig von anderen Objekten

Aktive Objekte

```
class ActiveObject
{
public:
  //! constructor
  ActiveObject ();

  //! destructor waits for thread to complete
  ~ActiveObject ();

  //! action to be defined by derived class
  virtual void action () = 0;

protected:
  //! use this method as last call in constructor of derived class
  void start ();

  //! use this method as first call in destructor of derived class
  void stop ();

private:
   ...
};
```

Aktive Objekte cont. I

```
#include<iostream>
#include"threadtools.hh"

Flag arrived_flag,continue_flag;

int main (int argc, char *argv[])
{
  Producer prod; // starte prod als aktives Objekt
  Consumer con;  // starte con als aktives Objekt

  return(0);
} // warte auf bis prod und con fertig sind
```

Aktive Objekte cont. II

```
class Producer : public ActiveObject
{
public:
  // constructor takes any arguments the thread might need
  Producer () {
    this->start();
  }

  // execute action
  virtual void action () {
    for (int i=0; i<100; i++) {
      std::cout « "ping" « std::endl;
      arrived_flag.signal();
      continue_flag.wait();
    }
  }

  // destructor waits for end of action
  ~Producer () {
    this->stop();
  }
};
```

Aktive Objekte cont. III

```
class Consumer : public ActiveObject
{
public:
  // constructor takes any arguments the thread might need
  Consumer () {
    this->start();
  }

  // execute action
  virtual void action () {
    for (int i=0; i<100; i++) {
      arrived_flag.wait();
      std::cout « "pong" « std::endl;
      continue_flag.signal();
    }
  }

  // destructor waits for end of action
  ~Consumer () {
    this->stop();
  }
};
```

Links

1. PThreads tutorial vom LLNL
   http://www.llnl.gov/computing/tutorials/pthreads/
2. LinuxThreads Library
   http://pauillac.inria.fr/∼xleroy/linuxthreads/
3. Thread safety of GNU standard library
   http://gcc.gnu.org/onlinedocs/libstdc++/17_intro/howto.html#3
4. Resourcen zu Pthreads Funktionen
   http://as400bks.rochester.ibm.com/iseries/v5r1/ic2924/index.htm?info/apis/rzah4mst.htm