UNIX-Schnittstelle

2. Threads

Alle Beispiel-Quellen mittels SVN unter:

https://svn.informatik.hu-berlin.de/svn/unix-2014/Threads

2.1. Vorbemerkungen

Was ist ein Thread?

Einzelner, sequentieller Steuerungsfluss in einem Programm. Die meisten klassischen Programme bestehen aus einem Thread (single thread).

Neu: Multithread Unterstützung: Das Betriebssystem erlaubt, dass ein Programm mehrere Threads enthält (multithread process) Dabei wird kein vollständiger neuer UNIX-Prozess benutzt. Heap, Code, Static-Data werden von den Threads gemeinsam benutzt. Stack und Register besitzt jeder Thread für sich.

Single Threaded Process

```
+------------------------------------------+
| Prozess                                  |
|                            +--------+    |
|                            |Register|    |
|                            +--------+    |
| +- - - - - - - - - - - - - - - - - - +   |
| | Memory                            |   |
| | +------+                +-------+  |   |
| | | Heap |                |       |  |   |
| | +------+                |       |  |   |
| | |Static|                | Stack |  |   |
| | +------+                |       |  |   |
| | | Code |                |       |  |   |
| | +------+                +-------+  |   |
| +- - - - - - - - - - - - - - - - - - +   |
+------------------------------------------+
```


Mulithreaded Process

```
+----------------------------------------------------+
| Prozess     +----------+----------+----------+     |
|             | Thread 1 | Thread 2 | Thread 3 |     |
|             |+--------+|+--------+|+--------+|     |
|             ||Register|||Register|||Register||     |
|             |+--------+|+--------+|+--------+|     |
| +- - - - - -|- - - - - |- - - - - |- - - - - |-+   |
| | Memory    |          |          |          | |   |
| | +------+  |+--------+|+--------+|+--------+| |   |
| | | Heap |  ||        |||        |||        || |   |
| | +------+  ||        |||        |||        || |   |
| | |Static|  || Stack  ||| Stack  ||| Stack  || |   |
| | +------+  ||        |||        |||        || |   |
| | | Code |  ||        |||        |||        || |   |
| | +------+  |+--------+|+--------+|+--------+| |   |
| |           +----------+----------+----------+ |   |
| +- - - - - - - - - - - - - - - - - - - - - - - +   |
+----------------------------------------------------+
```

```
Beispiel:

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

main()
{
    do_one_thing(&r1);      do_anaother_thing(&r2);   /*parallel*/

                do_wrap_up(r1,r2)
}

Herkömliche Realisierungsmöglichkeiten:

  Threads/Simple/simple.c     - in einem Prozess

  Threads/Simple/simple_processes.c  - in mehreren Prozessen
```



```
Implementationen - etwas Geschichte
-----------------------------------

DEC-UNIX  (1993)
           DECthreads  nicht offengelegt
           cma         Bibliothek für Digital Proprietary Interface
                       auf DECthread-Basis

SOLARIS   (1993)
           LW-Prozesse     - Systemcalls, offengelegt
           Solaris Threads - ähnlich POSIX 1003.4a

PTHREAD (POSIX 1003.4a)
           ab DEC-UNIX 4.0E
           ab Solaris 2.6: POSIX 1003.4a
           ab Linux 2.0 - libpthread
                   bis kernel 2.4 - PTL (Emulation in mehreren Prozessen)
                   ab Kern 2.6 - NPTL (PTL mittels LD_ASSUME_KERNEL=2.4.1)
```

```
Allgemeine Architektur für Threadinterface
------------------------------------------

traditioneller Prozess        Prozess mit Multithreads
                   Proc 1       Proc 2

  User                 {T}        {T} {T} {T}   {T}  Bounded Thread
                        |          |   |   |     |
                        |          +-+-+-+-+     |
                        |            |   |       |
                        o            o   o       o
                        +------------+-+-+-------+
                                       |
  Kernel                               *
                                       |
                                +---+--+--+
                                |   |     |
                               [x] [x]   [x]

{T} - Thread,  o - Execution resource (LWP,DECThread),  [x] - Prozessor
```



```
Arbeitsmodelle mit Threads
-------------------------

1. Boss/Worker Modell

   Ein Thread (Boss) übernimmt die Steuerung und übergibt weiteren
   Threads (Worker) Teilaufgaben. Die Worker melden den Arbeitszustand
   an den Boss.
   Ableitung davon ist das Work Queue Model, bei dem der Boss die Aufgaben
   in einer Workqueue plaziert und die Worker die Aufträge entnehmen.

2. Work Crew Modell

   Mehrere Threads erledigen zusammen eine Aufgabe.

                Prozess
   +---------+----------+---------+
   |         | Thread 1 |         |
   |         +----------+         |
   |  Setup  | Thread 2 | Cleanup |
   |         +----------+         |
   |         | Thread 3 |         |
   +---------+----------+---------+
```

3. Pipelining Modell

Beim Pipelining Modell wird die Aufgabe in mehrere nacheinander auszuführende Teilaufgaben zerlegt. Für jede Teilaufgabe wird ein Thread bereitgestellt, der die Eingangswerte vom vorhergehenden Thread übernimmt und die Ergebnisse an den nachfolgenden Thread übergibt.

```
             Prozess
+----------+----------+----------+
| Thread 1 | Thread 2 | Thread 3 |
+----------+----------+----------+
```

4. Kombination der Modelle 1.-3.

Kombinationen der Modelle 1.-3. sind für komplizierte Aufgaben üblich. Der Programmierer hat dabei alle Freiheiten.

Arbeiten mit Threads
----

1. Programmkomplexität
   Die Benutzung von Treads ist genau zu überlegen und nur dort einzusetzen wo sinnvoll. Oft wird durch den Verzicht auf Threads ein Geschwindigkeitsvorteil erreicht. Auch Threads erzeugen einen Overhead. Die Lesbarkeit von Programmen kann aber durch Threads erhöht werden.

2. Syncronisationsprobleme
   Zwei Threads greifen auf die gleiche Variable zu.
   Code Locking (ein Lock-Punkt im Prozess)
   Data Locking (kritische Variable durch Mutex-Variable oder Semaphore geschützt), flexibler als Code Locking

3. Deadlocks
   Gegenseitiges warten von Threads
   eigene Deadlocks oder rekursive Deadlocks
   Sheduling Deadlocks durch Prioritätsinversion (Threads blockieren sich durch überzogene Prioritätsforderungen gegenseitig)

4. Benutzung von nicht reenteranter Software
   Threads benutzen gleichzeitig Bibliotheksroutinen, die nicht reenterant sind.

5. Faustregeln für die Benutzung von Locks
   - Keine Locks über I/O-Operationen
   - Keine Locks beim Ruf von nicht reenteranten Funktionen
   - Keine überzogenen Prozessoranforderungen während Locks
   - Benutzung von multiplen Locks nur in gleicher Art und Weise

Thread-Operationen - eine Übersicht

Präfix für verschiede Threadbibliotheken:
SOLARIS-Pthread, Linux - pthread_
SOLARIS-Threads - thr_

1. Starten von Threads
   - pthread_create (SOLARIS, LINUX)
   - thr_create
   - thr_min_stack

2. Beenden von Threads
   - pthread_exit (SOLARIS, LINUX)
   - thr_exit

3. Abbrechen von Threads
   - pthread_cancel (SOLARIS, LINUX)

4. Warten auf das Ende von Threads
   - pthread_join (SOLARIS, LINUX)
   - thr_join

5. Freigeben von Threads (nach Abbruch oder Beendigung)
   - pthread_detach (SOLARIS, LINUX)

6. Holen eigenen Thread-Ident
   - pthread_self (SOLARIS, LINUX)
   - thr_self

7. Manipulation der Priorität von Threads

   Zahl der gleichzeitig aktiven Threads (Näherungswert)
   - thr_setconcurrency, thr_getconcurrency

   Setzen/Holen der Priorität eines Threads
   - pthread_getschedparam pthread_setschedparam (SOLARIS, LINUX)
   - thr_setprio thr_getprio

   Freigeben des eigenen Prozessors für ein Thread mit gleicher Priorität
   - pthread_yield (Linux)
   - thr_yield (Solaris)

   Anhalten und starten von Threads
   - thr_suspend, thr_continue

8. Initialisierung

   Einmaliges Abarbeitung einer Initialisierungsroutine
   - pthread_once

```
9. Manipulation von Attributen für Threads

     Thread-Attribute-Objekte verwalten
     SOLARIS/LINUX:  pthread_attr_init, pthread_attr_destroy

     Wert für Guardsize im Attributobjekt
       pthread_attr_getguardsize_np  pthread_attr_getguardsize
       pthread_attr_setguardsize_np  pthread_attr_setguardsize

     Art der Prioritätsvererbung im Attributeobjekt
       SOL/LINUX:   pthread_attr_getinheritsched
                    pthread_attr_setinheritsched

     Wert für Priorität im Attributeobjekt
       pthread_attr_getprio   cma_attr_get_priority
       pthread_attr_setprio   cma_attr_set_priority

     Art des Schedulingtypes im Attributeobjekt
       pthread_attr_getsched   cma_attr_get_sched
       pthread_attr_setsched   cma_attr_set_sched
       SOLARIS/LINUX: pthread_attr_getschedparam,
                      pthread_attr_setschedparam,
              pthread_attr_getschedpolicy,pthread_attr_setschedpolicy,
                      pthread_attr_getscope, pthread_attr_setscope

     Wert für Stackgrösse im Attribeobjekt
       SOL/LINUX: pthread_attr_getstacksize, pthread_attr_setstacksize
                pthread_attr_getstackaddr, pthread_attr_setstackaddr
```



```
2.2. Thread Bibliotheksrufe (SOLARIS/LINUX/POSIX)
=================================================
Include-File: thread.h  (pthread.h für POSIX-Threads)

   typedef unsigned int thread_t;
   typedef unsigned int thread_key_t;
   size_t               thr_min_stack(void);
   #define      THR_MIN_STACK   thr_min_stack()
   /*  thread flags (one word bit mask) */
   #define      THR_BOUND               0x00000001
   #define      THR_NEW_LWP             0x00000002
   #define      THR_DETACHED            0x00000040
   #define      THR_SUSPENDED           0x00000080
   #define      THR_DAEMON              0x00000100
   thread_t     thr_self();
   void         thr_yield(void);
   ...
Bemerkungen zur Compilierung:

SOLARIS/LINUX:
     pthreads:

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

SOLARIS:
     thread:

       cc [ flag ... ] file ...  -lthread [ library ... ]
```

1.Starten von Threads

```
int pthread_create( pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine)(void *), void * arg);
int thr_create( void *stack_base,  size_t stack_size,
                     void *(*start_routine)(void *), void *arg,
                     long flags, thread_t *new_thread_ID);
```

Starten eines neun Threads in dem aktuellen Prozess. Es wird mit der Abarbeitung der Routine start_routine begonnen. Es kann für die Routine ein Parameter arg angegeben werden.
Der Stack kann spezifiziert werden, wenn keine Werte angegeben werden (NULL,0), werden Standardwerte genommen (aus Heap THR_MIN_STACK Bytes). Wenn der thr_create erfolgreich war und new_thread != NULL wird in *new_thread der Identifier des neuen Threads abgelegt.
Default-Attribute: NULL (SOLARIS/LINUX)
Flags:

```
   THR_SUSPENDED - neuer Thread gestoppt, gestartet mit thr_continue()
   THR_DETACHED  - Thread wird nach  thr_exit sofort gestrichen
                     (Rückkehrwerte nicht nutzbar)
      THR_BOUND     - Eigener LWP
      THR_NEW_LP    - Zahle der LWP's im Pool um 1 erhöhen
      THR_DAEMON    - Dämon-Thread, wird nach dem letzten  thr_exit
                      eines Nicht-Dämon-Thread automatisch beendet
```

Ruckkehrwerte:

```
       0     - OK
     EAGAIN  - Systemlimits (LWP) erschöpft
     ENOMEM  - kein Speicher
     EINVAL  - unzulässige Stackwerte
```


2. Beenden von Threads

```
void pthread_exit(void *value_ptr);
void thr_exit(void *status);
```

Beenden des aktuellen Threads. Hat der Thread den Status DETACHED, werden alle thread-spezifischen Objekte einschliesslich des Exit-Status (value_ptr) gelöscht. Andernfalls werden der Thread-ID und der Exit-Status gespeichert bis ein anderer Thread sie mit thr_join() abfordert.
Rückkehrwert: keiner

3. Warten auf das Ende von Threads

```
int pthread_join(pthread_t thread, void **value_ptr);
int thr_join(thread_t wait_for, thread_t *departed, void **status);
```

thr_join blockiert den aktuelle Thread bis der spezifizierte Thread (wait_for) beendet wird. Der Thread muss im aktuellen Prozess sein und nicht den Status DETACHED haben. Wenn wait_for den Wert (thread_t)0 hat, wartet thr_join auf das Ende eines beliebigen nicht DETACHED Thread des aktuellen Prozesses. Wenn departed != NULL ist, enthält *departed nach erfolgreicher Abarbeitung von thr_join den ID des beendeten Threads. **value_ptr enhält den Exit-Status des beendeten Threads, wenn value_ptr!=NULL ist.
Rückkehrwerte:

```
0       - Ok
ESRCH   - ungültiger Parameter wait_for
EDEADLK - wait_for spezifiziert den aktuellen Thread (Deadlock)
```

Einfaches Beispiel mit Threads

```
   Threads/Simple/simple_threads.c
```


4. Holen der eigenen Thread-Ident

```
pthread_t pthread_self(void);
thread_t thr_self(void);
```

Holen des eigenen Thread-ID.

Rückkehrwert:
eigener Thread-ID

5. Initialisierung von Threads

```
int pthread_once(pthread_once_t *once_control,
                 void  (*init_routine) (void));
```

Hiermit können sich mehrer Threads so synchronisieren, dass eine Funktion init_routine nur einmal abgearbeitet wird. Die Synchronisation erfolgt über eine Datenstruktur, die durch once_control adressiert wird.

Rückerwert:
immer 0

Beispiel

Einmalig Initialisierung

```
Threads/Simple_once/once.c
```

6. Manipulation der Priorität von Threads

Zahl der gleichzeitig aktiven Threads (Näherungswert)

```
int thr_getconcurrency(void)
int thr_setconcurrency(int new_level)
```

thr_getconcurrency ermittelt einen Näherungswert für die Zahl der gleichzeitig aktiven Threads (LWP's). Das System bestimmt selbst wieviele LWP's für eine bestimmte Anzahl von ungebundenen Threads notwendig sind. Mit thr_setconcurrency kann der Programmierer die Zahl der LWP's ändern. Der übergebene Wert new_level dient dem System als Hinweis.

Rückkehrwert:
- thr_getconcurrency - Wert
- thr_setconcurrency:
  - 0 - Ok
  - EAGAIN - Systemressourcen überschritten
  - EINVAL - negativer Wert

Setzen/Holen der Priorität eines Threads

```
    int thr_getprio(thread_t target_thread, int *priority);
    int thr_setprio(thread_t target_thread, int priority);
 SOLARIS/LINUX:
    int pthread_getschedparam(pthread_t target_thread,
                int *policy, struct sched_param *param);
    int pthread_setschedparam(pthread_t target_thread,
          int  policy, const struct sched_param *param);

    struct sched_param {
                  int __sched_priority;
    };
```

pthread_getschedparam, thr_getprio speichert die aktuelle Priorität des Threads thread nach param.__sched_priority .
pthread_setschedparam, thr_setprio setzt die aktuelle Priorität des Threads thread auf den Wert param.__sched_priority .

Rückkehrwert:
- 0 - OK
- ESRCH - Thread exitiert nicht im aktuellen Prozess
- EINVEL - unzulässiger Prioritätswert

Freigeben des eigenen Prozessors

```
void pthread_yield();
void thr_yield();
```

thr_yield übergibt die Steuerung an einen anderen Thread mit gleicher oder höherer Priorität.

Anhalten und starten von Threads

```
int thr_suspend(thread_t target_thread);
int thr_continue(thread_t target_thread);
```

thr_suspend stoppt sofort die Arbeit des durch target_thread spezifizierten Threads. thr_continue aktiviert den durch thr_suspend gestoppten. bzw. durch thr_create() noch nicht gestarteten Thread target_thread wieder. thr_suspend hat auf einen gestoppten Thread keine Wirkung. thr_continue hat auf einen arbeitenden Thread keine Wirkung.

Beispiel Copy (mit SUN-Threads)

Nur für Solaris!!!!!!

Kopieren von Standard-Eingabe nach Standard-Ausgabe

Threads/My/copy.c

Wirkung von fork

```
Threads/Fork/fork.c   - fork in einem Thread
Threads/Fork/fork1.c  - fork1 (Solaris)
Threads/Fork/fork2.c  - fork
Threads/Fork/fork3.c  - forkall (Solaris)
Threads/Fork/vfork.c  - vfork, unsicher
```