# Alle Fäden in der Hand

**Thread Programmierung unter Linux in C und C++ (Teil 1)**

Wolfgang Schreiner

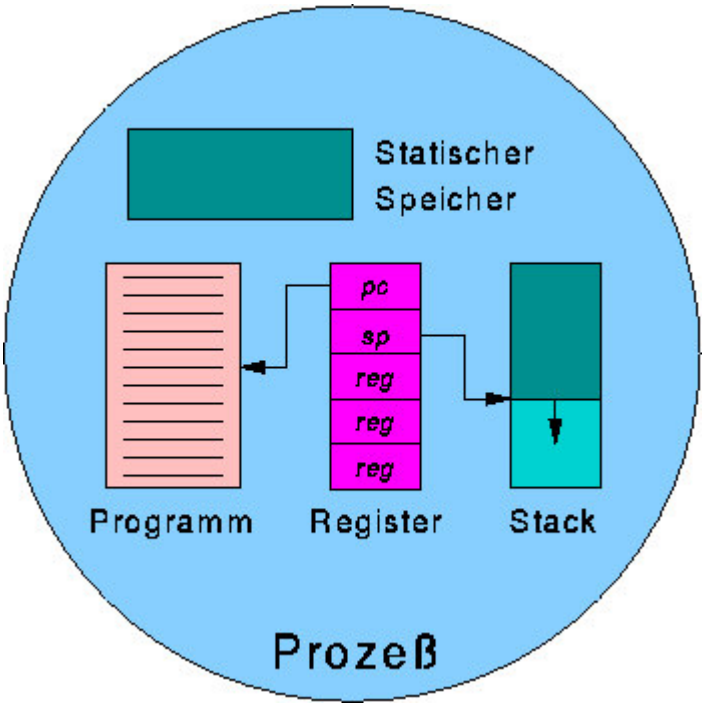
Threads (zu deutsch "Fäden") ermöglichen die gleichzeitige Ausführung von mehreren Programmpfaden innerhalb des Adreßraums eines Prozesses. Threads werden nicht nur in der parallelen Programmierung sondern vielfach auch in Client-Server Anwendungen eingesetzt. Auch unter Linux sind Thread-Pakete für C und C++ verfügbar.

Der erste Teil dieses Artikels beschreibt die Grundlagen der Thread-Programmierung, gibt einen kurzen Überblick über den POSIX Thread Standard, und verweist auf Thread-Pakete, die unter Linux verfügbar sind. Der zweite Teil wird RT++ beschreiben, eine C++ Klassenbibliothek für die parallele Programmierung auf Multiprozessor-Systemen, Workstations und Linux-PCs.

Bevor wir näher auf die Thread-Programmierung eingehen, müssen wir uns erst einmal mit den entsprechenden Betriebssystem-Grundlagen vertraut machen.

## Prozesse

Ein *Prozeß* ist ein in Ausführung begriffenes Programm. Folgendes Bild skizziert das traditionelle Prozeß-Modell von Unix und verwandten Betriebssystemen:




Ein Prozeß besteht dabei aus den folgenden Komponenten:

- Dem ausführbaren Code d.h. einer Kopie des Programms im Hauptspeicher.
- Dem statischen Speicher. Darin werden in einem C/C++ Programm alle Variablen der Speicherklassen "static" und "extern" abgelegt, insbesondere alle Variablen, die außerhalb einer Funktion (also "global") deklariert werden. Auch der Heap befindet sich im statischen Speicher und damit alle Objekte, die mittels der C Funktion `malloc` bzw. dem C++ Operator `new` erzeugt werden.
- Dem Stack. Darin werden alle dynamischen Variablen (Speicherklasse "automatic") abgelegt, insbesondere die in einer Funktion lokal deklarierten Variablen, die an eine Funktion übergebenen

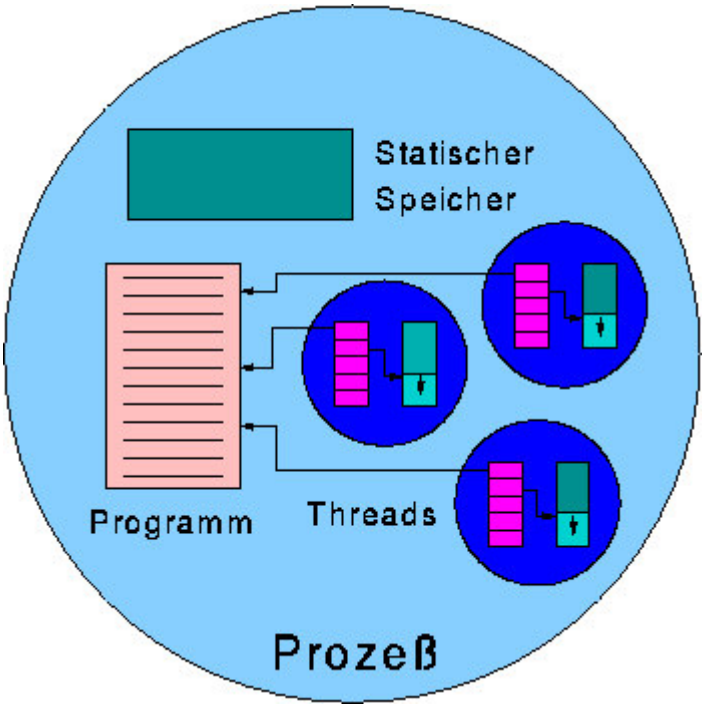
Argumente und die Zwischenergebnisse von Berechnungen.
- Dem Registersatz. Dieser enthält neben den allgemein verwendbaren Registern den *Programmzeiger* (program counter), der auf die nächste ausführbare Instruktion im Programm verweist, und den *Stackzeiger* (stack pointer), der die Grenze des belegten Stack-Bereichs angibt.

Jeder Prozeß stellt dabei einen eigenen Adreßraum dar und kann damit nur auf seine eigenen Daten zugreifen und sie verändern (eine Ausnahme davon bildet die Hauptspeicherkopie des Programmcodes, auf die ja nur lesend zugegriffen wird, und die damit von mehreren Prozessen gemeinsam benutzt werden kann).

In diesem traditionellen Modell ist mit jedem Prozeß ein einziger sequentieller Kontrollfluß verbunden, der durch den Inhalt des Programmzeigers und den in Registern, Stack und statischem Speicher enthaltenen Werten bestimmt wird. In einem Multi-Prozessor System können verschiedene Prozessoren zu gleicher Zeit verschiedene Prozesse ausführen; Prozesse stellen damit die natürlichen *Einheiten der Parallelisierung* dar. Aber auch in Systemen mit nur einem Prozessor werden durch Umschalten (context switch) von einem Prozeß zum nächsten alle Prozesse "quasi-parallel" abgearbeitet. Bei jedem Umschalten werden dabei die Register-Inhalte des alten Prozesses auf dem Stack gespeichert und die Register mit den entsprechenden Werten vom Stack des neuen Prozesses initialisiert.

# Threads

In Betriebssystemen, die Threads unterstützen, ändert sich das Prozeß-Modell wie in folgendem Bild dargestellt:




Ein Prozeß enthält einen oder mehrere Threads, von denen jeder über einen eigenen Programmzeiger, Registersatz und Stack verfügt. Threads verhalten sich damit in Bezug auf ihre Ausführung wie Prozesse, d.h. sie werden vom Betriebssystem unabhängig voneinander (und auf einem Multiprozessor-System echt parallel) ausgeführt. Im Gegensatz zu Prozessen operieren die Threads eines Prozesses aber im gleichen Adreßraum; sie teilen sich also den statischen Speicher und greifen damit auf die gleichen globalen Variablen zu. In gleicher Art und Weise teilen sich threads alle anderen Prozeß-spezifischen Informationen wie geöffnete Dateien, Signale, Funktionen zur Signal-Behandlung, etc.

Während verschiedene Prozesse durch Speicherschutz-Mechanismen voreinander abgeschottet sind und so der Absturz eines Prozesses einen anderen Prozeß nicht betrifft, sind die Threads eines Prozesses in

keiner Weise voreinander geschützt. Dafür aber können sie über den gemeinsamen Speicher sehr effizient (ohne den Umweg über das Betriebssystem) miteinander kommunizieren und so kooperieren. Da auch die Erzeugung von Threads im allgemeinen sehr effizient ist, eignen sich Threads daher hervorragend für die *fein-körnige Parallelisierung* von Anwendungen, wo jede Thread nur einen sehr geringen Teil der gesamten Arbeit erfüllt.

Threads werden daher auch manchmal als *leicht-gewichtige Prozesse* (LWP, light-weight processes) bezeichnet.

# Der Einsatz von Threads

Threads wurden erfunden, um durch Ausnutzung von Parallelität zu verhindern, daß Prozesse, die gewisse Dienstleistungen zur Verfügung stellen, durch Ausführung dieser Leistung auf längere Zeit blockiert sind [6].

Ein gutes Beispiel dafür sind *Client-Server Anwendungen*, bei denen ein Server-Prozeß auf die Anforderungen eines oder mehrerer Klienten reagieren muß. Für einen Datei- oder Datenbank-Server kann es sich dabei notwendig sein, laufend einen Strom von Anfragen von Hunderten von Klienten zu beantworten.

Der klassische sequentielle Rahmen für einen solchen Server sieht folgendermaßen aus:

```
server()
{
  while (1)
    {
      r = receive_request();
      a = process_request(r);
      answer_request(r, a);
    }
}
```

Diese Lösung besitzt allerdings den Nachteil, daß eine einzige komplexe Anfrage den Server auf längere Zeit blockieren kann und damit die Beantwortung aller später eingelangten Anfragen ungebührlich lange verzögert wird.

Eine entsprechende Lösung unter dem Einsatz von Threads würde dagegen den Server folgendermaßen realisieren:

```
dispatcher()
{
   while (1)
   {
     r = receive_request();
     start_thread(worker, r);
   }
}
worker(r)
{
  a = process_request(r);
  answer_request(r, a);
}
```

Der Server wird dabei zum *Dispatcher* (Zuteiler), der zur Bearbeitung jeder Anforderung eine neue Thread erzeugt. Der Server ist damit nur so lange blockiert, wie zum Empfang der Anforderung und zur Erzeugung der Thread notwendig sind.

Diese Lösung hat allerdings den Nachteil, daß nach Beantwortung einer Anfrage die jeweilige Thread terminiert und für spätere Anfragen Threads neu erzeugt werden müssen. Diese immerwährende Folge von Erzeugen neuer Threads und Termination alter Threads wird durch folgende Struktur vermieden:

```
R requests[N];
dispatcher()
{
  T threads[N];
  create_threads(threads);
  while (1)
  {
    r = receive_request();
    p = idle_thread();
    requests[p] = r;
    wake_thread(threads[p]);
  }
}
thread(p)
{
  while (1)
  {
    sleep_tread();
    a = process_request(requests[p]);
    answer_request(r, a);
  }
}
```

Der Dispatcher erzeugt zu Beginn einen Pool von Threads, von denen jede zur Bearbeitung einer Vielzahl von Anforderungen herangezogen werden kann. Sobald eine Thread eine Anforderung erledigt hat, legt sie sich schlafen und wird vom Dispatcher aufgeweckt, wenn eine neue Anforderung zur Bearbeitung ansteht. Die Kommunikation zwischen Dispatcher und Thread erfolgt dabei über eine globales Feld `requests`, in das der Dispatcher die jeweils zu beantwortende Anforderung schreibt und die von den Threads gelesen wird.

# POSIX Threads

Viele Betriebssysteme implementieren Threads direkt im System-Kern. Dazu gehören sowohl klassische Betriebssysteme mit monolithischer Struktur (z.B. Solaris) als auch moderne Systeme auf Basis eines Micro-Kerns (z.B. Mach). Während die meisten dieser Thread-Implementieren sich konzeptuell ähnlich sind, unterscheiden sich sich doch wesentlich in der Programmierschnittstelle und in anderen subtilen Einzelheiten. Auf Basis dieser system-spezifischen Threads geschriebene Anwendungen sind damit in keiner Weise portabel.

Die wesentlichste Bestrebung zur Standardisierung hat das IEEE mit dem POSIX-Standard P1003.1c "Portable Operating System Interface for Computer Environments, Threads Extensions" unternommen (als *POSIX Threads* oder *Pthreads* bekannt). Trotz der 1995 erfolgten Verabschiedung dieses Standards ist nicht zu erwarten, daß alle Hersteller auf ihre eigenen Programmier-Schnittstellen mit ihren spezifischen Eigenheiten und Erweiterungen verzichten. Zumindest als Alternative aber bieten die meisten eine weitgehend POSIX-kompatible Schnittstelle auf ihre Implementierungen an. Beispielsweise unterstützt Sun Microsystems neben den eigenen Solaris Threads auch POSIX Threads; die Gemeinsamkeiten und Unterschiede zwischen beiden Schnittstellen sind in der "Threads Page" [2] dokumentiert.

Im folgendes zeigen wir ein kleines Beispiel, das die wesentlichsten Punkte des POSIX Thread Modells beschreibt:

```
#include <pthread.h>   /* Posix 1003.1c threads */
...

pthread_mutex_t mutex; /* Wechselseitiger Ausschluss */
pthread_cond_t  cond;  /* Bedingungsvariable */
pthread_key_t key;     /* Thread-specifischer Datenschluessel */
int counter;           /* Thread-Zaehler */

void print(void)
{
  char *string = (char*)pthread_getspecific(key);
```

```
  printf("%s", string);
}

void *thread(void *arg)
{
  int number;        /* 0 = hello, 1 = world */

  pthread_mutex_lock(&mutex);
  number = counter; counter++;
  pthread_mutex_unlock(&mutex);

  pthread_setspecific(key, ((char**)arg)[number]);
  if (number)
  {
    print();
    pthread_cond_signal(&cond);
  }
  else
  {
    pthread_cond_wait(&cond);
    print();
  }

  return(0);
}

int main()
{
  char* strings[] = {"hello ", "world\n"};
  pthread_t thread1. thread2;

  pthread_init();

  mutex = PTHREAD_MUTEX_INITIALIZER;
  cond = PTHREAD_COND_INITIALIZER;
  pthread_key_create(&key, NULL);
  number = 0;

  pthread_create(&thread1, NULL, thread, strings);
  pthread_create(&thread2, NULL, thread, strings);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  return(0);
}
```

Dieses Programm druckt "hello world" (zugegebenermaßen auf etwas eigentümliche Weise) aus. Dazu verwendet es zwei Threads, die mittels einer globalen *counter* Variable bestimmen, wer für das "hello" und wer für das "world" zuständig ist. Die Ursprungs-Thread wartet auf die Termination der beiden Threads und beendet dann das Programm.

In diesem Beispiel werden bereits eine Vielzahl wesentlicher Konzepte der Thread Programmierung und des POSIX Thread Standards veranschaulicht:

- Die Funktion

  `pthread_create(`*`thread`*`, `*`attribute`*`, `*`fun`*`, `*`arg`*`)`

  erzeugt eine neue *thread*, die *fun*(*arg*) ausführt. *attribute* ist `NULL` oder legt verschiedene Thread Attribute (wie das verwendete Scheduling Verfahren) fest.
- Die Funktion

  `pthread_join(`*`thread`*`, `*`status`*`)`

  wartet auf die Termination von *thread*. *status* ist `NULL` oder liefert zusätzliche Information über die terminierte Thread.
- Die global deklarierte Variable

```
pthread_mutex_t mutex;
```

ist ein *Mutex* (mutual exclusion, wechselseitiger Ausschluß). Ein Mutex kann zum Schutz einer kritischer Region eingesetzt werden, die zu jedem Zeitpunkt nur von einer Thread betreten werden darf. Durch die Operationen

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

wird der Mutex gesperrt bzw. freigegeben. In obigem Beispiel dient der Mutex zum Schutz der globalen Variable *counter*, die von beiden Threads gelesen und um eins erhöht wird. Die Thread, die dabei schneller zum Zug kommt, druckt das "hello", die langsamere Thread das "world".

- Die Bedingungsvariable

```
pthread_cond_t cond;
```

wird zur Synchronisation der beiden Threads untereinander eingesetzt. Dabei führt die "world" Thread die Operation

```
pthread_cond_wait(cond);
```

die die Thread so lange blockiert bis die "hello" Thread mittels

```
pthread_cond_signal(cond);
```

die entsprechende Bedingung gesetzt hat. Damit ist sichergestellt, daß das "world" erst nach dem "hello" gedruckt wird.

- Mittels der Operation

```
pthread_key_create(key, destructor);
```

wird ein Schlüssel *key* für *thread-spezifische Daten* definiert (wobei *destructor* `NULL` oder eine benutzer-definierte Destruktor-Funktion ist) . Jede Thread kann nun mittels

```
pthread_setspecific(key, val);
```

einen Wert *val* an *key* binden und mittels

```
pthread_getspecific(key)
```

den an *key* gebundenen Wert lesen. Dieser Wert ist damit *global* innerhalb aller Funktionen, die innerhalb der *gleichen* Thread ausgeführt werden (während statisch deklarierte Variablen global für *alle* Threads sind)! In obigem Beispiel wird mittels einer thread-spezifischen Variable festgelegt, welche Zeichenkette die `print` Funktion ausdrucken soll.

Neben den angeführten Operationen bietet der POSIX Thread Standard noch vielerlei andere Möglichkeiten, auf die hier nicht näher eingegangen werden kann (für nähere Details siehe zum Beispiel die "Threads Page" [2]).

# Threads in Linux

Eine direkte Unterstützung des Betriebssystems für Threads durch Implementierung im System-Kern hat gewisse Vorteile. Vor allem ermöglicht sie es, System-Aufrufe (wie das Lesen von Dateien) so zu implementieren, daß gegebenenfalls nur die aufrufende Thread aber *nicht* der gesamte Prozeß (und damit alle anderen Threads dieses Prozesses) blockiert wird. Damit kann jede Thread die gesamte Funktionalität, die das Betriebssystem zur Verfügung stellt, unabhängig und gleichzeitig mit anderen Threads nutzen.

Ansonsten aber ist die Unterstützung des Betriebssystems *nicht* notwendig, um Threads zu implementieren. Tatsächlich bietet es eine Reihe von Vorteilen, Threads auf der Benutzer-Ebene, d.h. ohne Wissen und Unterstützung des Betriebssystems, zu realisieren:

- An erster Stelle steht die wesentlich größere System-Unabhängigkeit. Der einzige Teil eines Thread-Pakets, der nicht in einer portablen Hochsprache (beispielsweise C oder C++) geschrieben werden kann, ist die Initialisierung einer neuen Thread und das Umschalten von einer Thread zur nächsten (d.h. dem Sichern der aktuellen Registerinhalte auf dem Stack der alten Thread und dem Laden der Register vom Stack der neuen Thread). Einzig und allein diese Aufgaben erfordern prozessor-spezifische Assembler-Routinen.
- Weiters erhält man damit eine wesentlich größere Flexibilität. Viele Betriebssysteme legen ihren Thread Implementierungen unangenehme Einschränkungen auf. So ist zum Beispiel die Maximalanzahl der (von allen Benutzern!) zu verwendenden Threads oft eine feste Konstante, die ohne Neuübersetzung des Betriebssystems nicht vergrößert werden kann. Auch sind die Algorithmen zur Zuteilung des Prozessors an Threads (scheduling) oft direkt im Kern fixiert und können nicht vom Benutzer geändert werden. In Thread-Paketen auf Benutzer-Ebene können diese und andere Konfigurationsparameter jederzeit leicht geändert werden.
- Vor allem aber sind Threads, die im Adreßraum des Benutzers implementiert werden, im allgemeinen *effizienter* als System-Threads, da sie bei der Erzeugung und bei Umschalten von einer Thread zur nächsten keinen Aufruf des Betriebssystem-Kerns (und damit keinen Software-Interrupt) benötigen.

Es gibt eine Reihe von Thread-Implementierungen, die zur Gänze auf Benutzer-Ebene implementiert wurden. Für Linux sind dabei insbesondere die folgenden Pakete frei verfügbar:

- Eine am Massachusetts Institute for Technology (MIT) entwickelte Implementierung der POSIX Threads ist auch für ix86 Architekturen erhältlich [4].
- Eine für SPARC Architekturen unter Solaris entwickelte Pthreads Implementierung der Florida State University (FSU) wurde auf ix86 Architekturen unter Linux portiert [3].
- Das QuickThreads Paket [1] der Washington University ist zwar selbst kein Thread Paket, stellt aber die für die Entwicklung eigener Thread-Pakete notwendigen Assembler-Routinen für eine Vielzahl von Prozessor-Typen (unter anderem für die ix86 Welt) zur Verfügung.

Während Threads auf der Basis des POSIX Standards eine gewisse Portabilität gewährleisten, bieten sie den Nachteil, daß sie sich auf einer sehr niedrigen Abstraktions-Stufe bewegen. Sie spiegeln deutlich ihre Herkunft (von einer Betriebssystem-Schnittstelle) wider und sind damit für Zwecke der System-Programmierung hervorragend geeignet. Für die Entwicklung *paralleler Anwendungen* allerdings würde man sich oft ein bequemeres Programmiermodell wünschen. Der zweite Teil dieses Artikels wird von RT++ berichten, einem in C++ entwickelten Thread-Paket, das ein deutlich höheres Abstraktionsniveau anbietet [5].

# Literatur und Software

**[1]**

David Keppel. Tools and Techniques for Building Fast Portable Thread Packages, 1993. Software (QuickThreads) ftp://ftp.cs.washington.edu/pub/qt-002.tar.gz, Dokumentation ftp://ftp.cs.washington.edu/tr/1993/05/UW-CSE-93-05-06.PS.Z.

**[2]**

Sun Microsystems. The Threads Page, 1996. http://www.sun.com/sunsoft/Developer-products/sig/threads/.

**[3]**

Frank Mueller. A Library Implementation of POSIX Threads under UNIX for the SPARC/ix86, 1995. Software (FSU Pthreads) ftp://ftp.cs.fsu.edu/pub/PART/PTHREADS/pthreads.tar.Z, Dokumentation ftp://ftp.cs.fsu.edu/pub/PART/publications/pthreads_usenix93.ps.gz.

**[4]**

Chris Provenzano. A POSIX Threads Implementation, 1993. Software (MIT Pthreads) ftp://sipb.mit.edu/pub/pthreads.

**[5]**

Wolfgang Schreiner. RT++ -- Higher Order Threads for C++, 1996. Software und Dokumentation http://www.risc.uni-linz.ac.at/people/schreine/rt++/main.html.

**[6]**

Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1995.

---

*Wolfgang Schreiner ist am Research Institute for Symbolic Computation (RISC-Linz) der Johannes Kepler Universität Linz tätig und beschäftigt sich mit Parallelem und Verteilten Rechnen für Symbolische Anwendungen. Zu erreichen ist er unter Wolfgang.Schreiner@risc.uni-linz.ac.at.*