POSIX-Threads in Server-Applikationen

Kurt Wagner, Peter Meerwald
{kwagner|pmeerw}@cosy.sbg.ac.at

25. Mai 1998

Zusammenfassung

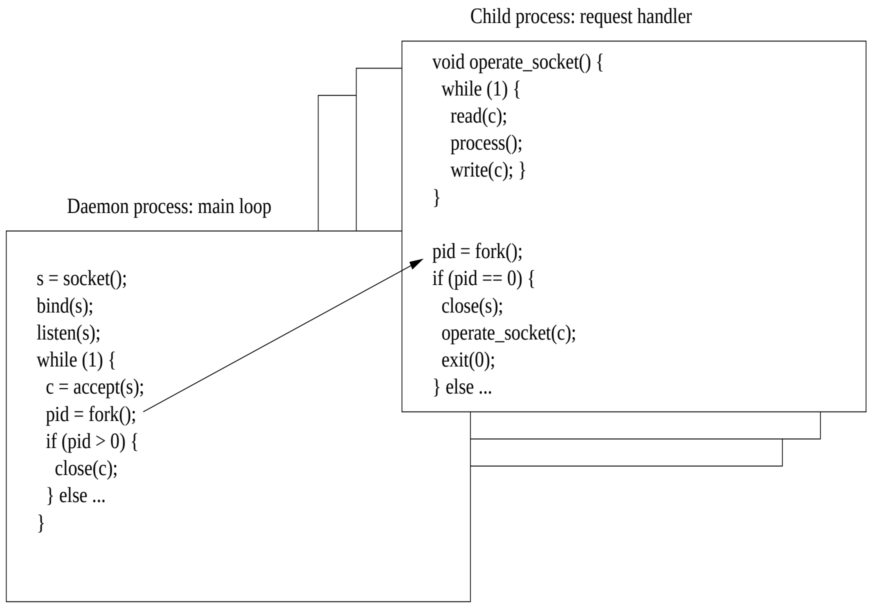
In diesem Dokument stellen wir die Problematik der Behandlung paralleler Anfragen an Server-Applikationen vor und zeigen, wie in C entsprechende Programme mit Hilfe des POSIX Thread API einfach erstellt werden können. Wir geben einen knappen Überblick der verfügbaren Thread-Implementierungen und beschreiben die wichtigsten Funktionen des POSIX Thread API. Die Code-Beispiele sollen den Vergleich mit ADA und Java dienen.

1 Einführung

1.1 Althergebrachte Nebenläufigkeit in C

fork() [2]

- `fork()` wird in vielen C Programmen unter Unix verwendet um Kind-Prozesse zu erzeugen und Nebenläufigkeit zu realisieren.
- `fork()` erzeugt eine unabhängige Kopie des laufenden Prozesses; der gesamte Speicher und alle Resourcen müssen kopiert werden.
- Die Ausführung wird in beiden Prozessen mit der nächsten Anweisung nach `fork()` fortgesetzt, der unterschiedliche Rückgabewert legt die weitere Ausführung fest.
- `socket()` erzeugt ein "handle" für eine Netzwerkverbindung, `bind()` bindet den handle an eine Netzwerkadresse, `listen()` erlaubt das Entgegennehmen von Verbindungen, `accept()` wartet auf eine eingehende Verbindung (blocking call).



Code-Beispiel: fork()ing daemon

```
void operate_socket(int c) {
  char msg[] = "hello, world\n";
  write(c, msg, sizeof(msg));
}

void main() {
  int s, pid;
  struct sockaddr_in sa;
  int v = 1;

  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
  bzero(&sa, sizeof(sa));
  sa.sin_family = AF_INET;
  sa.sin_addr.s_addr = htonl(INADDR_ANY);
  sa.sin_port = htons(PORT); /* assign port number */
  bind(s, &sa, sizeof(sa)); /* bind socket to address */
  listen(s, QUEUE); /* listen for incoming connections */

  while (1) {
    int size = sizeof(sa);
    int c = accept(s, &sa, &size); /*
blocking: new client socket/connection */

    pid = fork(); /* fork child process */
    if (pid == 0) { /* child */
      close(s);
      operate_socket(c); /* do something useful with the socket */
      close(c);
      exit(0); /* exit child program */
    } else if (pid > 0) { /* parent */
      close(c);
    } else; /* some error occured */
  }
}
```

Warum Threads? Viele I/O-Aufgaben lassen sich natürlich als parallel laufende Unterroutinen formulieren, allerdings ist das Konzept von `fork()` in mehrerer Hinsicht problematisch:

Daten und Resourcen (ausgenommen Stack und Register) stehen allen Threads zur Verfügung, dadurch entfällt (aufwendige) Interprozesskommunikation (z. B. via shared memory, pipes, ...). Kontext-Wechsel zwischen Threads innerhalb eines Prozesses sind nicht so aufwendig wie Wechsel zwischen Prozessen, da Threads keine eigene Resourcen besitzen (lightweight processes).[3]

1.2 POSIX-Standard

Neben den "echten" IEEE POSIX 1003.1c-1995 Threads (auch als PThreads bezeichnet) gibt es die DCE Threads, die auf Draft 4 des POSIX Standards basieren, und die Unix International (UI) Threads, die hauptsächlich auf Solaris- und UnixWare-Systemen implementiert sind (ein Vorläufer der POSIX-Threads).[3]

Der POSIX-Thread-Standard (.1c)[1] ist Teil des ANSI/IEEE 1003.1-1996 Standards. Ein weiterer Teil (.1b) beschäftigt sich mit Semaphore und real-time Eigenschaften. [7]

Thread-Modelle

- POSIX-Threads
- lightweight processes: SUN; `lwp_xxx()`
- UI Threads: `thr_xxx()`
- C Threads (Digital UNIX, Mach)
- Win32 Threads
- OS/2 Threads
- andere, durch div. libraries realisiert

Verfügbarkeit IEEE- und ISO-Standards sind nicht frei im Netz erhältlich sondern müssen kostenpflichtig unter folgender Adresse bestellt werden:

> IEEE Service Center, Publication Sales, P. O. Box 1331, 445 Hoes Lane, Piscataway, NJ 08854-1331, USA, publication SH 943252-NYF. Kosten: $ 120.

User-space threads keine Unterstützung durch das Betriebssystem, daher können user-space threads auch nicht in Multiprozessor-Umgebungen verteilt werden.

Kernel-supported threads in reine Kernel-Thread-Systemen übernimmt das Betriebssystem alleine das Scheduling, es gibt aber auch hybride Systeme.

Library-Implementierung müssen Kernel-Implementierungen nicht unterlegen sein. Vor allem hybride Systeme vereinen die Vorteile von kernel- und library-basierter Implementierung (wenig kernel-calls, einfaches scheduling). [6]

MT-safe ein Stück Code, das in multithreaded Programmen sicher ist und einen ausreichenden Grad an Nebenläufigkeit erlaubt wird als MT-safe (multithreaded-safe) bezeichnet.

[1] während der Standardisierung als 1003.4a bezeichnet

Code, der auf globale Daten zugreift, kann durch Einschließen in Mutex-Semaphore abgesichert werden; dadurch wird aber Nebenläufigkeit verhindert und man bezeichnet solchen Code als nicht MT-safe.

Thread-Implementierungen

Betriebssystem	Implementierung	POSIX API	Anmerkungen
Digital Unix 4.0	mixed	.1c	als patch
OpenVMS 7.0	mixed	.1c	als patch
AIX 4.2	kernel	draft 7	
HP/UX 10.20	kernel	.1c	als patch
OS/2	kernel		EMX
Linux 2.x	kernel	.1c	`clone()`, basiert auf LinuxThread
FreeBSD 3.x	kernel	.1c	
Win32	kernel	DCE	DCE auf Win32 Threads
Irix 6.2	mixed	.1c	`sproc()`
Solaris 2.5+	mixed	.1c	`thr_xxx()`, `lwp_xxx()`

siehe auch [3]

Verfügbarkeit von Thread-libraries

Bibliothek	Version	Autor	URL
LinuxThreads	0.8	X. Leroy	http://pauillac.inria.fr/~xleroy/linuxthreads
PCthreads	1.0	M. T. Peterson	http://www.aa.net/~mtp/PCthreads.html
DCEthreads	1.0	M. T. Peterson	http://www.aa.net/~mtp/
PThreads	1.6b4	C. Provenzano	http://www.mit.edu:8001/people/proven/
PART	3.5	F. Müller	ftp://ftp.cs.fsu.edu/pub/PART/PTHREADS/

siehe auch [3]

Überblick API - welche Bereicht deckt das API ab siehe [5]

- Initialization
- Thread management
- Synchronization
- Thread-specific data
- Thread scheduling
- Prozesskontrolle
- Signals
- Thread Cancellation
- Reentrant functions

Notwendige Änderungen an der Library Einige C-Library-Funktionen sind nicht MT-safe, zum Beispiel `strtok()`, da die Funktionen über einen eigenen Zustand verfügt (static local data). Der Zustand muß somit für jeden Thread separat gespeichert werden. Der Suffix `_r` kennzeichnet die MT-safe Variante einer Funktion: zum Beispiel `strtok_r()` oder `gethostbyname_r()`.

Andere Funktionen erhalten in Threads eine andere Bedeutung, zum Beispiel `sleep()`. `sleep()` suspendiert den aufrufenden Threads für die angegebene Zeitspanne; um den gesamten Prozess zu suspendieren ist ein Aufruf von `pthread_delay_np()` erforderlich.

Auch die Behandlung von signals ändert sich durch Multithreading erheblich. [5], [6]

2 Beispiele

Voraussetzungen

- Kompilieren mit -D_REENTRANT bzw. -DTHREAD_SAFE
- Linken mit -lpthread (POSIX-Threads) bzw. -lthreads (UI-Threads)

2.1 Erzeugen und Beenden eines Threads

`int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`

`pthread_create()` creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function `start_routine()` passing it `arg` as first argument. The new thread terminates either explicitly, by calling `pthread_exit()`, or implicitly, by returning from the `start_routine()` function. The latter case is equivalent to calling `pthread_exit()` with the result returned by `start_routine()` as exit code.

The `attr` argument specifies thread attributes to be applied to the new thread. In the future thread attributes may be more interesting, but for now, most applications can get simply using the default by passing `pthreads_attr_default` in the threads attribute parameter position.

`int pthread_join(pthread_t th, void **thread_return);`

`pthread_join()` suspends the execution of the calling thread until the thread identified by `th` terminates, either by calling `pthread_exit()` or by being cancelled.

`void pthread_exit(void *retval);`

`pthread_exit()` terminates the execution of the calling thread. The `retval` argument is the return value of the thread. It can be consulted from another thread using `pthread_join()`.

Code-Beispiel: tierwelt.c

```
void giblaut(void *arg) {
 char *message = arg;
  printf("%s ", message);
}

void main() {
  pthread_t hund, katz;
  char *message1 = "wau";
  char *message2 = "miau";

  pthread_create(&hund, NULL, giblaut, message1);
  pthread_create(&katz, NULL, giblaut, message2);

  pthread_join(hund, NULL);
  pthread_join(katz, NULL);
}
```

Vergleich mit Ada

- Ada-Tasks starten gleichzeitig mit dem Hauptprogramm - Posix-Threads werden explizit im Hauptprogramm ins Leben gerufen.
- In Ada eigener Body für jeden Task - PThread kann jede Prozedur werden.
- Ada-Hauptprogramm terminiert erst, wenn auch alle Child-Tasks terminiert sind - das C-Hauptprogramm muß mittels `pthread_join()` zum Warten gezwungen werden.

2.2 Verwenden einer Mutex-Semaphore

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

`pthread_mutex_init()` initializes the mutex object pointed to by `mutex` according to the mutex attributes specified in `mutexattr`. If `mutexattr` is NULL, default attributes are used instead.

`int pthread_mutex_lock(pthread_mutex_t *mutex));`

`pthread_mutex_lock()` locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and `pthread_mutex_lock()` returns immediately. If the mutex is already locked by another thread, `pthread_mutex_lock()` suspends the calling thread until the mutex is unlocked.

`int pthread_mutex_unlock(pthread_mutex_t *mutex);`

`pthread_mutex_unlock()` unlocks the given mutex.

`int pthread_mutex_destroy(pthread_mutex_t *mutex);`

`pthread_mutex_destroy()` destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance.

Code-Beispiel Leser/Schreiber kommunizieren über einen gemeinsamen Puffer, der genau ein Zeichen aufnehmen kann.

```
char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex;

main() {
  pthread_t reader;

  pthread_mutex_init(&mutex, NULL);
  pthread_create(&reader, NULL, reader_function ,NULL);
  writer_function();
}

void writer_function(void) {
  while (1) {
    pthread_mutex_lock(&mutex);
    if (buffer_has_item == 0) {
      buffer = make_new_item();
      buffer_has_item = 1;
    }
    pthread_mutex_unlock(&mutex);
  }
}

void reader_function(void) {
  while (1) {
    pthread_mutex_lock(&mutex);
    if (buffer_has_item == 1) {
      consume_item(buffer);
```

```
      buffer_has_item = 0;
    }
    pthread_mutex_unlock(&mutex);
  }
}
```

2.3 Verwenden einer Condition-Variable

A condition (short for "condition variable") is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

`int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`

`pthread_cond_init()` initializes the condition variable `cond`, using the condition attributes specified in `cond_attr`, or default attributes if `cond_attr` is NULL.

`int pthread_cond_signal(pthread_cond_t *cond);`

`pthread_cond_signal()` restarts one of the threads that are waiting on the condition variable `cond`. If no threads are waiting on `cond`, nothing happens. If several threads are waiting on `cond`, exactly one is restarted, but it is not specified which.

`int pthread_cond_broadcast(pthread_cond_t *cond);`

`pthread_cond_broadcast()` restarts all the threads that are waiting on the condition variable `cond`. Nothing happens if no threads are waiting on `cond`.

`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`

`pthread_cond_wait()` atomically unlocks the `mutex` (as per `pthread_unlock_mutex()`) and waits for the condition variable `cond` to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The `mutex` must be locked by the calling thread on entrance to `pthread_cond_wait()`. Before returning to the calling thread, `pthread_cond_wait()` re-acquires `mutex` (as per `pthread_lock_mutex()`).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition variable.

`int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`

`pthread_cond_timedwait()` atomically unlocks `mutex` and waits on `cond`, as `pthread_cond_wait()` does, but it also bounds the duration of the wait. If `cond` has not been signaled within the amount of time specified by `abstime`, the mutex `mutex` is re-acquired and `pthread_cond_timedwait()` returns the error ETIMEDOUT.

Code Beispiel: Semaphore mittels Condition-Variable

```
struct sem_t {
  pthread_mutex_t lock;
  pthread_cond_t cond;
  int count;
};

void sem_init(sem_t *s, int init) {
  pthread_mutex_init(&s->lock, NULL);
```

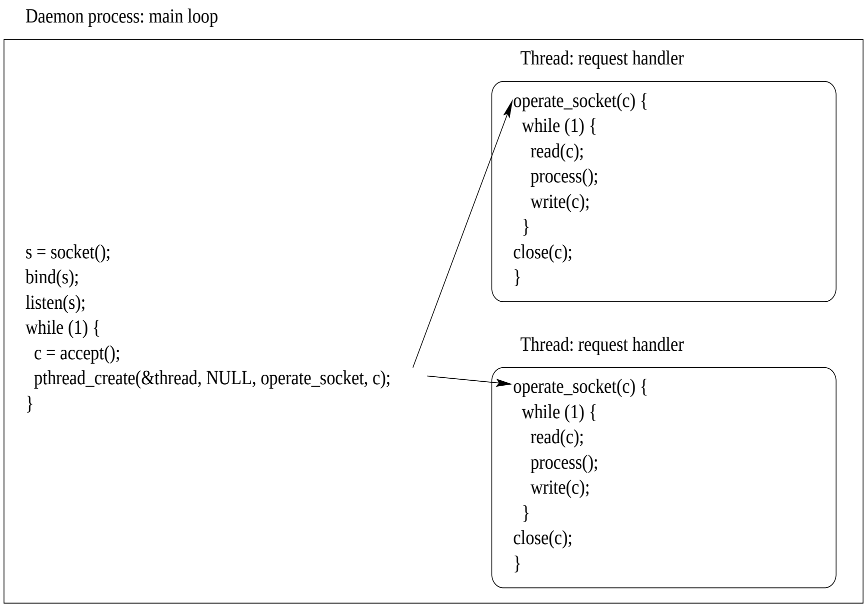
```
  pthread_cond_init(&s->lock, NULL);
  s-> count = (init < 0 ? 0 : init);
}

void sem_down(sem_t *s) {
  pthread_mutex_lock(&s->lock);
  while (s->count == 0)
    pthread_cond_wait(&s->cond, &s->lock);
  s->count--;
  pthread_mutex_unlock(&s->lock);
}

void sem_up(sem_t *s) {
  int contention;

  pthread_mutex_lock(&s->lock);
  contention = (s->lock++ == 0);
  pthread_mutex_unlock(&s->lock);
  if(contention)
    pthread_cond_signal(&s->cond);
}
```

2.4 Das Produzent/Konsument-Problem

```
/* The classic producer-consumer example.
   Illustrates mutexes and conditions.
   All integers between 0 and 9999 should be printed exactly twice,
   once to the right of the arrow and once to the left. */

#include <stdio.h>
#include <pthread.h>

#define BUFFER_SIZE 16
/* Circular buffer of integers. */
struct prodcons {
  int buffer[BUFFER_SIZE];      /* the actual data */
  pthread_mutex_t lock;         /* mutex
ensuring exclusive access to buffer */
  int readpos, writepos;        /* positions
for reading and writing */
  pthread_cond_t notempty;      /* signaled
when buffer is not empty */
  pthread_cond_t notfull;       /* signaled when buffer is not full */
};

/* Initialize a buffer */
void init(struct prodcons * b) {
  pthread_mutex_init(&b->lock, NULL);
  pthread_cond_init(&b->notempty, NULL);
  pthread_cond_init(&b->notfull, NULL);
  b->readpos = b->writepos = 0;
}

/* Store an integer in the buffer */
void put(struct prodcons * b, int data) {
  pthread_mutex_lock(&b->lock);
  /* Wait until buffer is not full */
  while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) {
    pthread_cond_wait(&b->notfull, &b->lock);
    /* pthread_cond_wait reacquired b->lock before returning */
  }
  /* Write the data and advance write pointer */
  b->buffer[b->writepos] = data;
  b->writepos++;
  if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
  /* Signal that the buffer is now not empty */
  pthread_cond_signal(&b->notempty);
```

```
  pthread_mutex_unlock(&b->lock);
}

/* Read and remove an integer from the buffer */
int get(struct prodcons * b) {
  int data;
  pthread_mutex_lock(&b->lock);
  /* Wait until buffer is not empty */
  while (b->writepos == b->readpos)
    pthread_cond_wait(&b->notempty, &b->lock);
  /* Read the data and advance read pointer */
  data = b->buffer[b->readpos];
  b->readpos++;
  if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
  /* Signal that the buffer is now not full */
  pthread_cond_signal(&b->notfull);
  pthread_mutex_unlock(&b->lock);
  return data;
}

/* A test program: one thread inserts integers from 1 to 10000,
   the other reads them and prints them. */

#define OVER (-1)
struct prodcons buffer;

void * producer(void * data) {
  int n;
  for (n = 0; n < 10000; n++) {
    printf("%d --->\n", n);
    put(&buffer, n);
  }
  put(&buffer, OVER);
  return NULL;
}

void * consumer(void * data) {
  int d;
  while (1) {
    d = get(&buffer);
    if (d == OVER) break;
    printf("---> %d\n", d);
  }
  return NULL;
}

int main() {
  pthread_t th_a, th_b;
  void * retval;

  init(&buffer);
  /* Create the threads */
  pthread_create(&th_a, NULL, producer, 0);
  pthread_create(&th_b, NULL, consumer, 0);
  /* Wait until producer and consumer finish. */
  pthread_join(th_a, &retval);
  pthread_join(th_b, &retval);
  return 0;
}
```

3 Implementierung einer einfachen Server-Applikation

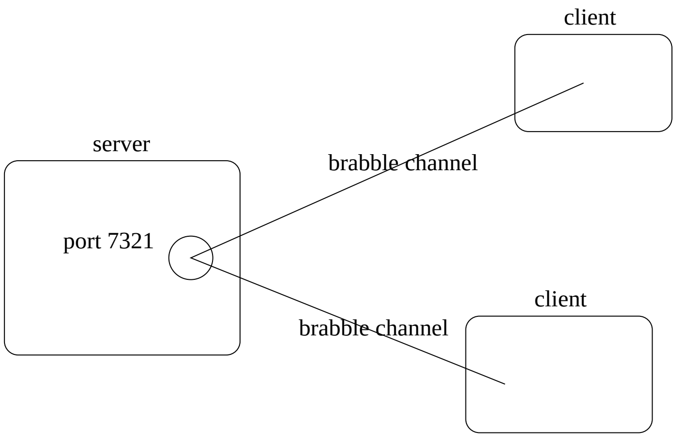


Code-Beispiel: threading daemon

```
#define _REENTRANT
#include <pthread.h>

void operate_socket(int c) {
  char msg[] = "hello, world\n";
  write(c, msg, sizeof(msg));
  close(c);
}

main() {
  int s, pid;
  int v = 1;
  struct sockaddr_in sa;
  pthread_t t;

  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
  bzero(&sa, sizeof(sa));
  sa.sin_family = AF_INET;
  sa.sin_addr.s_addr = htonl(INADDR_ANY);
  sa.sin_port = htons(PORT); /* assign port number */
  bind(s, &sa, sizeof(sa)); /* bind socket to address */
  listen(s, QUEUE); /* listen for incoming connections */

  while (1) {
    int size = sizeof(sa);
    int c = accept(s, &sa, &size); /*
blocking: new client socket/connection */
    pthread_create(&t, NULL, operate_socket,
c); /* create new thread */
  }
}
```

3.1 Architektur eines einfachen IRC-Servers

Ein C Programm soll in einer Server-Applikation ein einfaches Protokoll implementieren, über das Nachrichten zwischen beliebig vielen Clients ausgetauscht werden können. [1]



Ein Server-Thread registriert neu hinzukommende Clients und starten für jede neue Verbindung einen weiteren Threads. Diese Client-Threads bearbeiten die eingehenden Anfragen synchron und senden Antworten an eventuell mehrere Clients.

Der Server verwaltet die bestehenden Verbindung und Kommunikationskanäle in mehreren Listenstrukturen. Da alle Threads diese Strukturen sowohl lesen als auch schreiben können, muß der Zugriff jeweils exklusiv erfolgen - dazu werden die Listen durch Mutex-Semaphore geschützt.

3.2 Probleme

- Verfügbarkeit: Linux hängt in `fgets(stream, ...)` im zweiten Thread; Besserung bringt voraussichtlich die glibc2 in den neuesten Linux-Distributionen.
- SOLARIS: streams nicht MT-safe; `fprintf(stream, ...)` hängt, wenn auf stream gleichzeitig gelesen wird.
- Granularität der Mutex-Semaphore: Sperren zu großer Strukturen verhindert Nebenläufigkeit, "feines" Sperren mühsam.

3.3 Aktuelle Entwicklungen

- glibc2 für Linux
- Apache 1.3b7
- Squid 1.2b20

Literatur

[1] Meerwald, P., "The Brabble Chat Protocol (BCP)", Mai 1998.

[2] Stevens, R., "Advanced TCP Programming", Addison-Wesley, 1997.

[3] O'Sullivan, B., "Answers to frequently asked questions for comp.programming.threads", May 1997.

[4] Wagner, T., Towsley, D., “Getting Started With POSIX Threads”, University of Massachusetts at Amherst, July 1995.

[5] Müller, F., “Pthreads Library Interface”, Florida State University, April 1996.

[6] Müller, F. “A Library Implementation of POSIX Threads under UNIX”, Florida State University, January 1993.

[7] Kuhn, M. “A Vision for Linux 2.2 - POSIX.1b Compatibility and Real-Time Support”, Universität Erlangen, Juli 1997.