Pthreads

Andreas Dähn

Januar 2009

Orientierung und Motivation

- Pthreads sind eine Library, die es erlaubt, auf POSIX-Systemen portablen Code zu entwickeln, der Threads nutzt.
- Verglichen mit OpenMP und TBB sind Pthreads weiter unten in der Abstraktionshierarchie, wie noch gezeigt wird.

- Die Nutzung von Phtreads ist sinnvoll, wenn es auf maximale Optimierung ankommt oder keine der anderen Techniken zur Verfügung steht.

Geplanter Inhalt

- Threads und Prozesse – Einordnung
- Die Grundidee des Threads – Sinnhaftigkeit zwischen `fork()` und `exec()`
- Pthreads – Threads mit dem POSIX-Siegel
  - Einführung anhand eines Beispiels
  - Überblick über die Funktionen
- Implementierungen von Pthreads
  - Unices
  - Linux
  - Windows?
- Pthreads auf Maschinen mit echter Parallelität
- Typische Einsatzszenarien
- Quellen

1. Threads und Prozesse – kurzes Sortieren

Prozess Ein *Prozess führt ein einziges Programm aus und hat einen einzigen Kontrollfluß* [TAN92, S. 342].

- Adreßraum, globale Variablen, geöffnete Dateien, abhängige Prozesse, Uhren, Signale, Semaphore,... gehören zu genau einem Prozess.
- Interprozesskommuniktaion relativ kompliziert bzw. teuer

Thread Ein Thread ist ein „Mini-Prozess“ (auch *leichtgewichtiger Prozess* genannt)

- Besitzt eigenen Programmzähler, Stack, Registersatz, Zustand,.... (Verwaltungsinformationen)
- Threads eines Programms teilen sich einen Adressraum →Kommunikation über shared Memory einfach

Mehrere Threads per Prozess sind möglich, mehrere Prozesse pro Thread nicht.

2. Die Grundidee des Threads

- Prozesserstellung und Kontextwechsel bei Prozessen ist teuer.
- Bei eng verwobenen Aufgaben müssen entsprechend große Speicherbereiche dupliziert werden
  - Behelfslösung: Copy on write, also den Speicher erst dann tatsächlich kopieren, wenn der neue Prozess hineinschreibt.
- Threads als weitere Eltern-Kind-Beziehung: Thread — Elternprozess verhält sich ähnlich wie Prozesse — Maschine.
- Sie erlauben, *Parallelität mit sequentieller Ausführung und blockierenden Systemaufrufen kombinieren zu können* [TAN92, S. 619]
  - Blockierende Aufrufe sind für Entwickler meist übersichlicher.
  - Parallelität ist für die Ausführung meist schneller.
  - Aber die vielen gemeinsamen Variablen erforden sorgfältige Nutzung von Mutices.

2.1. Unterscheidung von User- und Kernelthreads

Für den folgenden Überblick über die im Umlauf befindlichen Implementiereungen eine kurze Erklärung dieser beiden Begriffe:

- Userthread: Wird vom Prozess allein behandelt, für den Kernel gibt es keine Threads.
- Kernelthread: Heutige Bedeutung; Ausführungsstränge innerhalb des Prozesses aus Kernelsicht.

2.2. Threads – Implementationen

- Das Konzept war da – es wurde fröhlich drauflosimplementiert (ein paar Beispiele von [DWP01]):

System	UT	KT	
MS Windows 3.1	x		Kein Speicherschutz
Mac OS 9	x	x	„Thread Manager“ f. UT; „Multiprocessing Services“ f. KT
Sun OS	x		„green threads“, spezialisiert für Nutzerinteraktion

- Daneben bringen einige Runtimeumgebungen noch Eigenes mit:
  - Die Java-Runtime implementiert Threads selbst und nutzt diese, falls das zugrundeliegende Betriebssystem kein Threading beherrscht. Somit konnte Threading im Konzept der Sprache verankert werden.
  - Die .NET-Runtime bietet noch weitere Parallelitäten wie die sog. „AppDomains“, die als logische Prozesse verwaltet werden, aber nicht an zugrundeliegende Betriebsmittel gebunden sind. Ferner werden auch Threads über einen „ThreadPool“ von der Runtime angeboten.
- ... und bei Unices brachte jede Distribution ihre eigene Implementation mit.

3. Pthreads – der Dschungel wird gelichtet

- POSIX, ausgeschrieben *P*ortable *O*perating *S*ystems *I*nterface *X*, war also genau das, was diesem Chaos fehlte.
- mit dem POSIX-Standard 1003.1c wird eine vereinheitlichte Schnittstelle zu Threads vorgeschlagen
  - ca. 50 Funktionen zu Threads und unterstützenden Funktionen (Mutexverwaltung, ..)
  - per Definition definiert in `pthread.h`
- Ergebnis sind POSIX Threads, kurz Pthreads.

$\longrightarrow$ Erhöhte Portabilität von Programmen auf unixartigen Systemen

- Eine gewisse Zeit verging, bis sie sich durchgesetzt haben.

4. Einführung in Pthreads am konkreten Beispiel

- Auf den kommenden Folien soll der folgende Programmcode parallelisiert werden:

```
int main(int, char**)
{
        int i, result;
        for ( i = 0; i < 10 ; i++ ) {
                result = myCalc(i, i+1);
                saveResult(i, result);
        }
}
```

- Wobei `myCalc(int, int)` eine längerdauernde Berechnung ist und `saveResult(int, int)` das Ergebnis auf einen Ausgabekanal (z.B. `STDOUT`) ablegt sowie die Teilergebnisse zusammenfasst:

```
void saveResult(int i, int result)
{
        printf("myCalc(%i, %i) \t=\t %i", i, i+1, result);
        usleep(100); // here may be some necessary action
        printf("\n");
        sum += result;
}
```

4.1. Ursprungscode

```
int
main(int argc, char** agrv)
{
        int i;
        int result;
        for ( i = 0; i < 10 ; i++ ) {
                result = myCalc(i, i+1);
                saveResult(i, result);
        }
}
```

Die Ausgangslage. Im folgenden werden Änderungen blau und rot hervorgehoben werden.

4.2. Vorbereiten auf Pthreads

```
void* act(void* param)
{
        int* i = (int*) param;
        int  result = myCalc(*i, (*i)+1);
        saveResult(*i, result);
        return NULL;
}

int
main(int argc, char** agrv)
{
        int i;
        int result;
        for ( i = 0; i < 10 ; i++ ) {
                act((void*) &i);
//              result = myCalc(*i, (*i)+1);
//              saveResult(*i, result);
        }
}
```

Beim Erstellen eines Threads kann nur eine Funktion zum Ausführen angegeben werden; entsprechend muß zusammengefasst werden.

4.3. Einfügen von Pthreads

```
#include <pthread.h>
pthread_t* thread;
int
main(int argc, char** agrv)
{
        int i;
        int result;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
#define COUNT 10
        thread=(pthread_t*) malloc(sizeof(pthread_t) * COUNT);
        for ( i = 0; i < COUNT ; i++ ) {
                pthread_create(&thread[i], &attr, act, (void*) &i);
        }
        for ( i = 0; i < COUNT ; i++)
                pthread_join(thread[i], (void*) &result);
        free(thread);
}
```

Multithreading ist implementiert... und der Code zu 90% ein anderer.

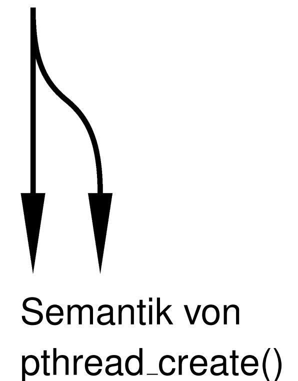
4.4. Genutzte Funktionen im Detail I

Thread erzeugen

`int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);` [MAN], wobei

- `*thread` ID des erzeugten Threads, um ihn aus dem Elternprozess zu kontrollieren
- `*attr` Attribute des erzeugten Threads, z.B. zu Scheduling, Scope oder Stack
- `(*start_routine)(void *)`, Pointer zu einer Routine, die als Thread gestartet werden soll
- `*arg`, (beliebiger) Parameter, der der Routine übergeben wirg.
- der Rückgabewert 0 ist, wenn der Aufruf erfolgreich war.
- eine mögliche aufzurufende Funktion so aussieht:

```
void doSomeFancyCalc(void* p)
{
        int* i1, i2;
        i1 = p[0]; i2 = p[1];
        p[3] = secretOp(i1, i2);
        pthread_exit(p+3);
}
```



Semantik von pthread_create()

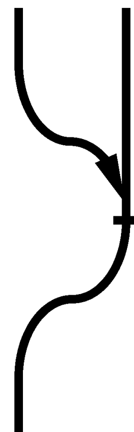
4.5. Genutzte Funtionen im Detail II

Threadende erwarten

`int pthread_join(pthread_t thread, void **value_ptr);` [MAN], wobei hier

- `thread` das Handle des jeweiligen Threads ist (erster Parameter von `pthread_create`)
- `**value_ptr`, der Wert, den der Thread an `pthread_exit` übergeben hat
- der Rückgabewert bei Erfolg 0 ist.

`pthread_join` wartet, bis der referenzierte Thread seine Arbeit einstellt. Entsprechend muss sichergestellt sein, dass der übergebene Thread das auch tatsächlich tut...



Semantik von pthread_join()

4.6. Das Ergebnis des Codes

```
[ad001@glas /usr/home/ad001/uni/0809/HS/MPC/example]$ make
cc -O2 -fno-strict-aliasing -pipe -g -march=prescott -c main.c
cc -o ttest calc.o main.o -lpthread
[ad001@glas /usr/home/ad001/uni/0809/HS/MPC/example]$ ./ttest
myCalc(1, 2)    =        0myCalc(3, 4)   =        200myCalc(5, 6)        =
myCalc(4, 5)    =        762myCalc(6, 7)        =        5168myCalc(8, 9)

myCalc(7, 8)    =        10782
myCalc(2, 3)    =        32myCalc(10, 11)       =        59952
myCalc(15, 16)  =        430202myCalc(16, 17)   =        589698myCalc(17, 18)




myCalc(19, 20)  =        1368192


Die Summe ist 4082462.
```

Beim Ausgabekanal ist wohl noch ein Mutex notwendig ...

4.7. Einfügen eines Mutex für das Ausgabemedium

```
pthread_t* thread;
pthread_mutex_t mux;
void* act(void* param) {
        int i = *((int*) param);
        int  result = myCalc(i, i+1);
        pthread_mutex_lock(&mux);
        saveResult(i, result);
        pthread_mutex_unlock(&mux);
        free(iptr); return NULL;
}
int main(int argc, char** agrv) {
        int i, result;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_mutex_init(&mux, NULL);
#define COUNT 20
        thread=(pthread_t*) malloc(sizeof(pthread_t) * COUNT);
        int* iptr;
        for ( i = 0; i < COUNT ; i++ ) {
                iptr = (int*) malloc(sizeof(int)); *iptr = i;
                pthread_create(&thread[i], &attr, act, (void*) iptr);
        }
        for ( i = 0; i < COUNT ; i++)
                pthread_join(thread[i], (void*) &result);
        free(thread);
        printf("Die Summe ist %i.\n", sum);
}
```

4.8. Genutzte Funtionen im Detail III

Mutex sperren `int pthread_mutex_lock(pthread_mutex_t *mutex);` [MAN], wobei hier

- `*mutex` das Handle des angeforderten Mutex ist. Dieser muß zuvor mit `pthread_mutex_init()` initialisiert worden sein.
- der Rückgabewert bei Erfolg 0 ist.

`pthread_mutex_lock` sperrt den angegebenen Mutex. Ist dieser gerade gesperrt, wird der Prozess schlafen gelegt, bis der Mutex frei wird. Im Hintergrund ist eine atomare `test-and-set`-Anweisung notwendig.

4.9. Genutzte Funtionen im Detail IV

Mutex freigeben `int pthread_mutex_unlock(pthread_mutex_t *mutex);` [MAN], wobei hier

- `*mutex` das Handle des freizugebenden Mutex ist.
- der Rückgabewert bei Erfolg 0 ist.

`pthread_mutex_unlock` gibt die zuvor erworbene Sperre eines Mutex wieder auf.

4.10. Zeilen im fertigen Programm, die sich mit Pthreads befassen:

```
pthread_t* thread;
pthread_mutex_t mux;
void* act(void* param) {
        int i = *((int*) param);
        int  result = myCalc(i, i+1);
        pthread_mutex_lock(&mux);
        saveResult(i, result);
        pthread_mutex_unlock(&mux);
        free(iptr); return NULL;
}
int main(int argc, char** agrv) {
        int i, result;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_mutex_init(&mux, NULL);
#define COUNT 20
        thread=(pthread_t*) malloc(sizeof(pthread_t) * COUNT);
        int* iptr;
        for ( i = 0; i < COUNT ; i++ ) {
                iptr = (int*) malloc(sizeof(int)); *iptr = i;
                pthread_create(&thread[i], &attr, act, (void*) iptr);
        }
        for ( i = 0; i < COUNT ; i++)
                pthread_join(thread[i], (void*) &result);
        free(thread);
        printf("Die Summe ist %i.\n", sum);
}
```

4.11. Fazit des Beispiels

- Dadurch, dass Pthreads auf relativ niedrigem Level ansetzen, sind entsprechend viele Modifikationen am Programmcode notwendig.
- Threading an sich benötigt bereits relativ viel Umsichtigkeit beim Programmieren:
  - die Übergabe von Schleifenvariablen ist nicht sicher (da die Schleife ja nach dem Starten des Threads weiterläuft)
  - gewisse Betriebsmittel (hier ein Ausgabekanal) müssen abgesichert werden, dies erfordert noch entsprechende Mutices.

5. Die Pthreads-Funktionen

- Im Folgenden ein kurzer Überblick über die Funktionengruppen, die von Pthreads bereitgestellt werden:
  - Grundlagen: Thread erstellen, verlassen, darauf warten, ...
  - erweiterte Grundlagen: Selbstkenntnis, Start- und Stoproutinen
  - Kommunikationsmittel (Konditionsvariablen, Mutices)
  - Tuning am Scheduler
  - und viele Helfer.

- Wirklich detaillierte Beschreibungen *jeder* Funktion finden sich in Manpages auf allen gängigen Systemen.

5.1. Grundlegende Funktionen

- `pthread_create()` einen neuen Thread erstellen.
- `pthread_exit()` einen Thread beenden (den aufrufenden Thread).
- `pthread_cancel()` einen Thread beenden (einen anderen Thread).
- `pthread_testcancel()` testen, ob ein Thread sich beenden lassen moechte.
  - `pthread_setcancelstate()` die entsprechende Eigenschaft setzen
  - `pthread_getcancelstate()` und auslesen.
- `pthread_join()` auf einen Thread warten.
- `pthread_kill()` und `pthread_sigmask()` Kommunikation über Signale.
- `pthread_detach()` der Implementation von Pthreads sagen, dass der angegebene Thread seinen Speicher nicht mehr braucht.

5.2. erweiterte Grundlagen: Selbstkenntnis, Start- und Stoproutinen

- `pthread_self()` ID des eigenen Threads auslesen.
- `pthread_equal()` zwei Threads anhand ihrer Handles auf Gleichheit testen.
- `pthread_cleanup_push()` eine Routine vor Threadende als Aufräumroutine eintragen.
- `pthread_cleanup_pop()` und umgekehrt wieder austragen.

5.3. Kommunikationsmittel

- Pthreads bieten zwei unterschiedliche Techniken an:
  - Konditionsvariablen

    und
  - Mutices.

5.3.1. Konditionsvariablen

- Beim Anlegen von K.Variablen kann man Attribute mit angeben. Diese werden mit den Funktionen `pthread_condattr_(init|destroy|getpshared|setpshared)()` angelegt bzw. modifiziert.
- `pthread_cond_init()` Konditionsvariable initialisieren.
- `pthread_cond_destroy()` Konditionsvariable freigeben.
- `pthread_cond_signal()` aktiviert die Konditionsvariable und gibt
- `pthread_cond_wait()` –Aufrufe frei.
- `pthread_cond_timedwait()` wie einfaches `wait`, bricht aber nach vorgegebener Zeit ab.
- `pthread_cond_broadcast()` wie `signal()` aber für alle wartenden Threads.

5.3.2. Mutices

- Es gibt, wie bei Konditionsvariablen, Attribute (die sich ähnlich verhalten)
- `pthread_mutex_init()` um einen Mutex zu initialisieren.
- `pthread_mutex_destroy()` um ihn wieder zu zerstören.
- `pthread_mutex_lock()` um ihn zu locken. Falls der Mutex gerade gelockt ist, wird der aufrufende Prozess schlafen gelegt und geweckt, sobald der Mutex frei ist. Es handelt sich also um kein „busy waiting".
- `pthread_mutex_trylock()` prüft, ob ein Mutex gelockt werden kann. Wenn ja, wird er gelockt, wenn nein, ein Errorcode zurückgegeben aber nicht blockiert.
- `pthread_mutex_unlock()` um einen gelockten Mutex wieder freizugeben.

- `pthread_once()` nicht diret zu Mutices gehörig, passt aber an diese Stelle: Es wird sichergestellt, dass die übergebene Funktion nur genau einmal aufgerufen wird.

5.3.3. Einschub: Reenteranz

- Bezeichnet die Fähigkeit einer Routine, dass sie zugleich von mehr als einem Prozess/Thread ausgeführt werden kann
- z. B. Signalhandler oder – deutlich trivialer – Rekursion:

```
void myproc(void) {
 pthread_mutex_lock(mymutex);
 if (something)
  myproc();
 ...
```

- Das Verhalten von Pthreads ist *undefiniert*!

5.4. Scheduler-Attribute

- Die Scheduler-Optionen müssen nicht unbedingt vom Betriebssystem unterstützt werden.
- Wenn Sie unterstützt werden, ist es eine Art „Finetuning“.
  - Genau dieses Feintunig ist bei Mehrprozessormaschinen wichtig.
- `pthread_getschedparam();`
- `pthread_setschedparam();`
- `pthread_attr_getschedpolicy();`
- `pthread_attr_setschedpolicy();`

6. Implementierungen von Pthreads

- Damit Pthreads effizient nutzbar sind, ist es vorteilhaft, wenn sie im Kernel implementiert sind. Entsprechend bedeutet Portabilität, dass die Systeme Phtreads unterstützen müssen.

- Wie sieht es also mit der Unterstützung von Ptheads auf einigen verbreiteten Platformen aus?
  - Linux?
  - Mac OS?
  - Windows?

6.1. Implementierungen I

- Linux [LAM07]
  - Die ursprüngliche *LinuxThreads*-Library war nicht POSIX-Kompatibel und hatte bekannte Probleme.
  - *Next Generation POSIX Threads (NGPT)* sollte der Nachfolger werden, wäre aber auch nicht 100%ig Pthreads-kompatibel gewesen.
  - *Native POSIX Thread Library (NPTL)* ist 2003 entwickelt worden und wurde (zusammen mit entsprechenden Änderungen am Kernel) schnell zum Quasi-Standard auch auf Linux.

6.2. Implementierungen II

- Mac OS?
  - Mac OS X basiert auf Darwin.
  - Darwin ist Unix-Kompatibel, entsprechend werden Pthreads nativ unterstützt.

6.3. Implementierungen III

- Microsoft Windows
  - Honestly, who cares about POSIX? Windows is based on the Win32-API, not on POSIX.

  - Es gibt allerdings Projekte wie „Pthreads Win32“, die an einer Portierung arbeiten.

7. Pthreads auf Einprozessormaschinen

- „Simulierte Parallelität“
  - Nicht der tatsächlichen Abarbeitung gegenüber
  - Nur dem Nutzer gegenüber scheinbare Nebenläufigkeit.

- Trotzdem Gewinn in der Programmausführung:
  - Blockierende Aufrufe (Netzwerk-IO, Dateizugriffe, ...) halten die Programmoberfläche bzw. den restlichen Programmablauf nicht an
  - Abtrennen: Aufwändige Berechnungen können ebenfalls in den Hintergrund geschoben werden.

8. Pthreads auf Mehrprozessormaschinen

- Wirklich rechenaufwändige Aufgaben können in einen eigenen Thread ausgelagert werden *und nun auf einer eigenen CPU laufen.*
- Engpass sind CPU-Caches bei datenlastigen Operationen durch beide Threads.

- Wichtig sind die Scheduler-Eigenschaften
  - Priority: Generelle Festlegung, welcher von 2 Threads bei unterschiedlicher Priorität die CPU erhält.
  - Policy: Welcher Algorithmus soll verwendet werden, um bei 2 Threads mit gleicher Priorität zu entscheiden, welcher die CPU erhält.

9. Typische Einsatzszenarien

- Verbreitete Programmiermodelle im Multithreading
  - Boss-Worker
  - Peer-Modell
  - Pipeline-Modell
- Threadpools vs. Wegwerfthreads

9.1. Programmiermodelle: Boss-Worker

- Ein Master-Thread (der *Boss*) nimmt Arbeitsaufträge z.B. aus dem Netzwerk an.
- Für jede Aufgabe wird ein Thread (der *Worker*) angelegt, der sie bearbeitet und das Ergebnis zurückliefert.
- Das Ergebnis wird zum Auftraggeber (Netzwerk, etc.) zurückgeliefert und der Thread zerstört.

- Vorteile
  - relativ einfache, intuitive Programmierung
  - keine Systemlast durch mehr wartende Threads als notwendig: Im Standardfall „wartet“ nur ein Thread auf eingehende Aufträge, andere existieren nur dann, wenn sie auch arbeiten.
- Nachteile
  - Bei jedem neuen Auftrag Overhead für das Erstellen eines neuen Threads.

9.2. Programmiermodelle: Peer-Modell

- Vom ersten Thread (der beim Programmstart ja vorhanden ist) wird eine Zahl von Threads erzeugt, die der Anzahl an z. B. Eingabekanälen entspricht.
- Danach benimmt sich dieser erste Thread genauso, wie die anderen und arbeitet „seine“ Aufgabe ab.

- Vorteile
  - Es entsteht kaum Overhead für eine dynamische Verteilung von Aufgaben an Arbeitsthreads
  - Jeder Thread ist alleine für eine Aufgabe verantwortlich, von der Annahme bis zum Zurücksenden – einfachere Implementierung
- Nachteile
  - Genaue Kenntnis der Aufgaben zum Zeitpunkt der Implementierung ist erforderlich
  - Geringe Anpassungsfähigkeit

9.3. Programmiermodelle: Pipeline-Modell

- Voraussetzung: Die Aufgabe kann in mehrere hintereinanderausführbare Teilaufgaben unterteilt werden.
- Jeder Thread arbeitet eine Stufe ab
- Der Thread bekommt seine Eingabe von der vorherigen Stufe und gibt sein Ergebnis an die nachfolgende Stufe als deren Eingabe weiter.

- Vorteile
  - Sehr effiziente Bearbeitung einer Aufgabe bei sinnvoller Teilung möglich (z. B. Video-Pipeline)
- Nachteile
  - Alle Stufen müssen ungefähr gleich schnell sein – die Geschwindigkeit der ganzen Pipeline ist so schnell, wie die langsamste Stufe.
  - Die Aufgabe muss sich entsprechend effizient teilen lassen.

9.4. Thread-Pools

- Anstatt (z. B. bei einem Boss-Worker-Modell) Threads ständig neu zu erzeugen und zu zerstören werden, diese als Vorrat gehalten.
- Zu Programmbeginn wird eine bestimmte Anzahl von Threads erzeugt, die dann *sofort* zur Verfügng stehen
  - Umsetzung: Thread erstellen, mit Synchronisationsmittel in Wartezustand bringen und zu geeigneter Zeit aufwecken.

- Vorteile
  - Keine Wartezeiten für Threaderzeugung
  - Threads ggf. für verschiedene Aufgaben nutzbar
- Nachteile
  - Möglicherweise unnötige Last, da 90% der Threads nur warten und nie eine Aufgabe bekommen.

  → Kann aber durch geeignete Programmierung abgefangen werden.
- Wird vielfältig genutzt (z. B. Datenbankserver, Webserver, . . .)

10. Quellen

NBF96 Bradford Nichols, Dick Buttlar and Jaqueline Prolux Farrell: „Pthreads Programming“, 1. Auflage. O’Reilly & Assoc, Inc. 1996. ISBN 1-57592-115-1.

TAN92 Andrew S. Tannenbaum: „Moderne Betriebssysteme“ in der Übersetzung von Radermacher und Baumgarten, 2. Auflage. Hanser 1992. ISBN 3-446-18402-3.

LAM07 Adrien Lamothe: „Pthreads Programming: A Hands-on Introduction“. Vortrag auf der Open Source Convention 2007.
`http://conferences.oreillynet.com/presentations/os2007/os_lamothe.pdf`

LUK09 Peter Luksch: „Cluster Computing“, Folien zu den Vorlesungen zum Thema Pthreads. Universität Rostock, Wintersemester 2008/2009.

DWP01 Artikel der deutschsprachigen Wikipedia zu „Thread (Informatik)“ am 05.01.2009.

MAN Manpages zu den Pthread-Funktionen auf FreeBSD-current

Fragen, Anregungen, Kritik... →Kontakt

Per

Mail an `ad001@uni-rostock.de`.

IRC im euIRC im Channel #uni-rostock als direkte Frage oder Query an `ad001`.

oder einfach jetzt stellen :)