Programmierung mit Threads

- Prozesse und Threads im Betriebssystem
- Typen von Threads
- Einsatzbereiche Threads
- Die Pthreads-Schnittstelle

Siehe: http://en.wikipedia.org/wiki/Thread_%28computer_science%29

Programmierung mit Threads
Die zehn wichtigsten Fragen

- Worin unterscheiden sich Threads und Prozesse?
- Was sind Threads auf Benutzerebene und auf Systemebene?
- Was ist eine hybride Thread-Realisierung?
- Warum sind Threads ein Thema beim Hochleistungsrechnen?
- Wie funktioniert Thread-Scheduling?
- Was ist ein Prioritätswechselprotokoll?
- Wann programmieren wir mit Threads?
- Welche Grundoperationen bietet die Pthreads-Schnittstelle?
- Wie funktioniert ein Mutex?
- Wie arbeitet man mit Bedingungsvariablen?

Prozesse und Threads

Traditionelle Prozesse

- Ein Prozeß ist der Ablauf eines Programms
- Aus Betriebssystemsicht
  - Prozeß ist Einheit der Ressourcenbelegungen (Speicher, Dateien, E/A-Ports)
  - Prozeß ist Einheit der Prozessorzuteilung
- Grundidee von Threads
  - Aufspaltung dieser Eigenschaften:
    Prozeß ist Einheit der Ressourcenbelegung
    Thread ist Einheit der Prozessorzuteilung

Prozesse und Threads...

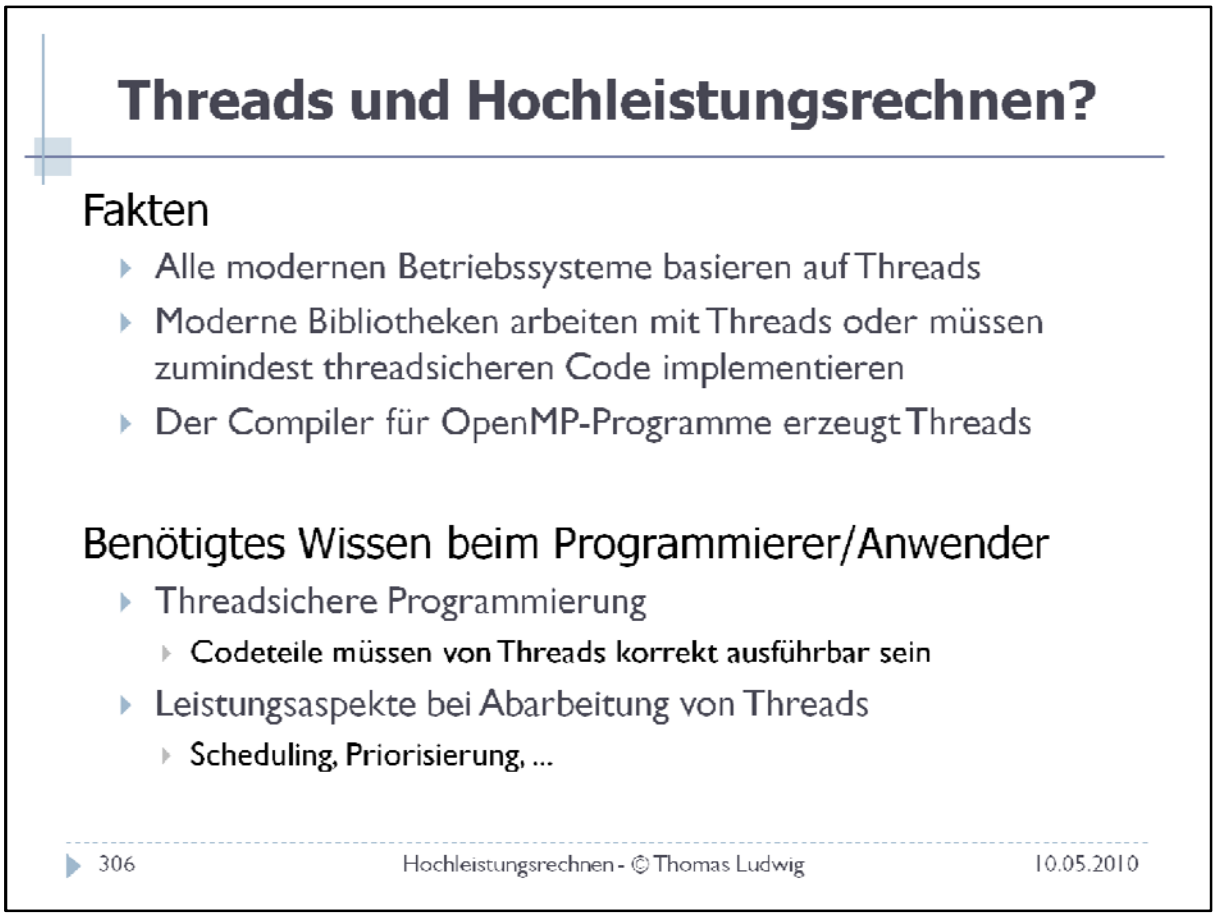
Eigenschaften von Threads

- Threads eines Prozesses haben gemeinsame Ressourcen (Speicher, Dateien, ...)
  - Einfache, effiziente Kooperation möglich
  - Aber: kein wechselseitiger Schutz
- Parameter zur Erzeugung eines Threads
  - Zeiger auf Programmcode (typisch auf Funktion)
  - Kellergröße (feste maximale Größe)
- Einheit der Prozessorzuteilung
- Geringer Verwaltungsaufwand
- Schneller Wechsel (innerhalb desselben Prozesses)

Prozesse und Threads...

Bedeutung von Threads

- Verringerung der Antwortzeit von Servern
  - Unterbrechbarkeit langer Anfragen
- Durchsatzsteigerung
  - Überlappung blockierender Systemaufrufe
- Behandlung asynchroner Ereignisse
- Realzeitanwendungen
  - Hochpriore Threads für zeitkritische Aufgaben
- Basis für Parallelverarbeitung auf SMPs
- Strukturierung von Programmen



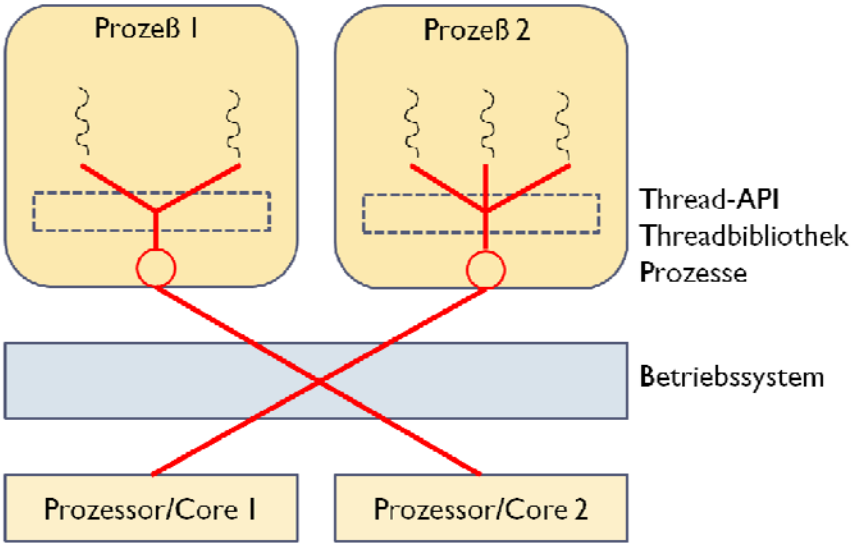
Siehe: http://en.wikipedia.org/wiki/Threadsafe

„Thread safety is a key challenge in multi-threaded programming. It was once only a concern of operating system programmers but since the late 1990s has become a commonplace issue. In a multi-threaded program, several threads execute simultaneously in a shared address space. Every thread has access to virtually all the memory of every other thread. Thus the flow of control and the sequence of accesses to data often have little relation to what would be reasonably expected by looking at the text of the program, violating the principle of least astonishment. Thread safety is a property aimed at minimizing surprising behavior by re-establishing some of the correspondences between the actual flow of control and the text of the program." [http://en.wikipedia.org/wiki/Threadsafe] (Siehe auch: http://en.wikipedia.org/wiki/Principle_of_least_astonishment)

„A subroutine is reentrant, and thus thread-safe, if 1) the only variables it uses are from the stack, 2) execution depends only on the arguments passed in, and 3) the only subroutines it calls have the same properties. Such a sub-routine is sometimes called a "pure function", and is much like a mathematical function."

Arten von Threads

Threads auf Benutzerebene (user threads)



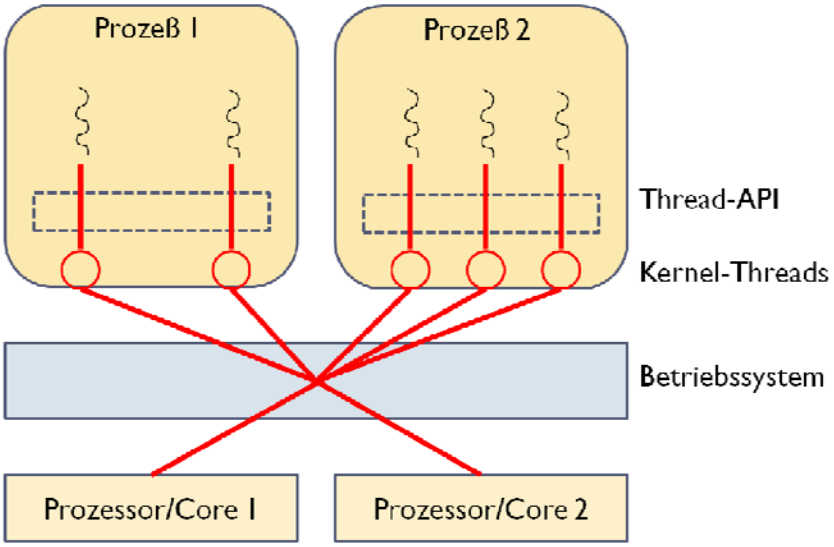
Arten von Threads...

Threads auf Benutzerebene...

- Schnelle Thread-Erzeugung und Kontextwechsel
  - Keine Betriebssystemaufrufe notwendig
- Relativ gute Portierbarkeit
  - Aber: Problem mit nicht-wiedereintrittsfähigen Laufzeitbibliotheken
- Geringe Belastung des BS-Kerns
- Keine echte Parallelität innerhalb eines Prozesses
- Blockierung des Prozesses bei blockierendem Systemaufruf
- Keine Koordination zwischen BS- und Thread-Scheduler

Arten on Threads...

Threads auf Systemebene (kernel threads)



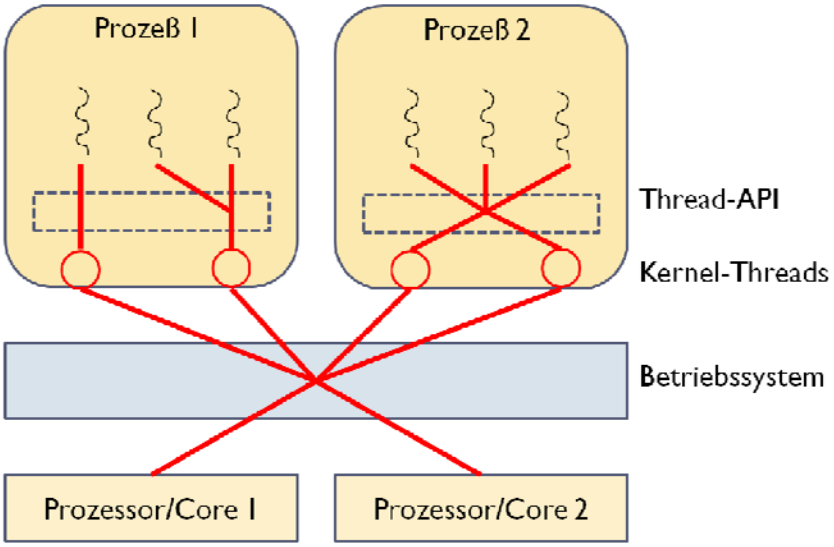
Arten von Threads...

Threads auf Systemebene...

- Langsamer als User Threads wg. Systemaufrufen
  - Z.B. Zeit zur Erzeugung auf SparcStation2
    U´Thread: 50µs / K´Thread: 350µs / Prozeß 1700µs
- Unterschiedliche Schnittstellen und Semantiken bei unterschiedlichen Schnittstellen
  - Aber POSIX-Standard IEEE P1003.1c: Pthreads
- Parallelität auch innerhalb eines Prozesses
- Keine Probleme mit Blockierungen und Wiedereintrittsfähigkeit (Reentrancy)

Arten von Threads...

Hybride Thread-Realisierung



Arten von Threads...

Hybride Thread-Realisierung

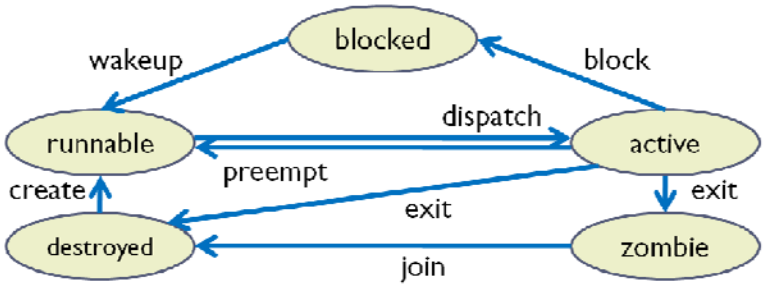
- Anzahl der Kernel-Threads von der Thread-Bibliothek bestimmt

  Abhängig von Anzahl User-Threads und Prozessoren
- Anzahl Kernel-Threads und Zuordnung von User-Thread zu Kernel-Thread steuerbar
  - Z.B. eins-zu-eins-Zuordnung wenn zeitkritisch
- Anzahl von Kernel-Threads bestimmt
  - Maximalen Parallelitätsgrad
  - Max. Zahl gleichzeitig blockierender Systemaufrufe

Thread-Scheduling

- Scheduling: Zuteilung rechenbereiter Threads an Prozessoren
- Präemptives Scheduling: rechnender Thread kann unterbrochen werden
- Threadzustände

  (POSIX-Implementierungsmodell)



Thread-Scheduling...

Schedulingverfahren

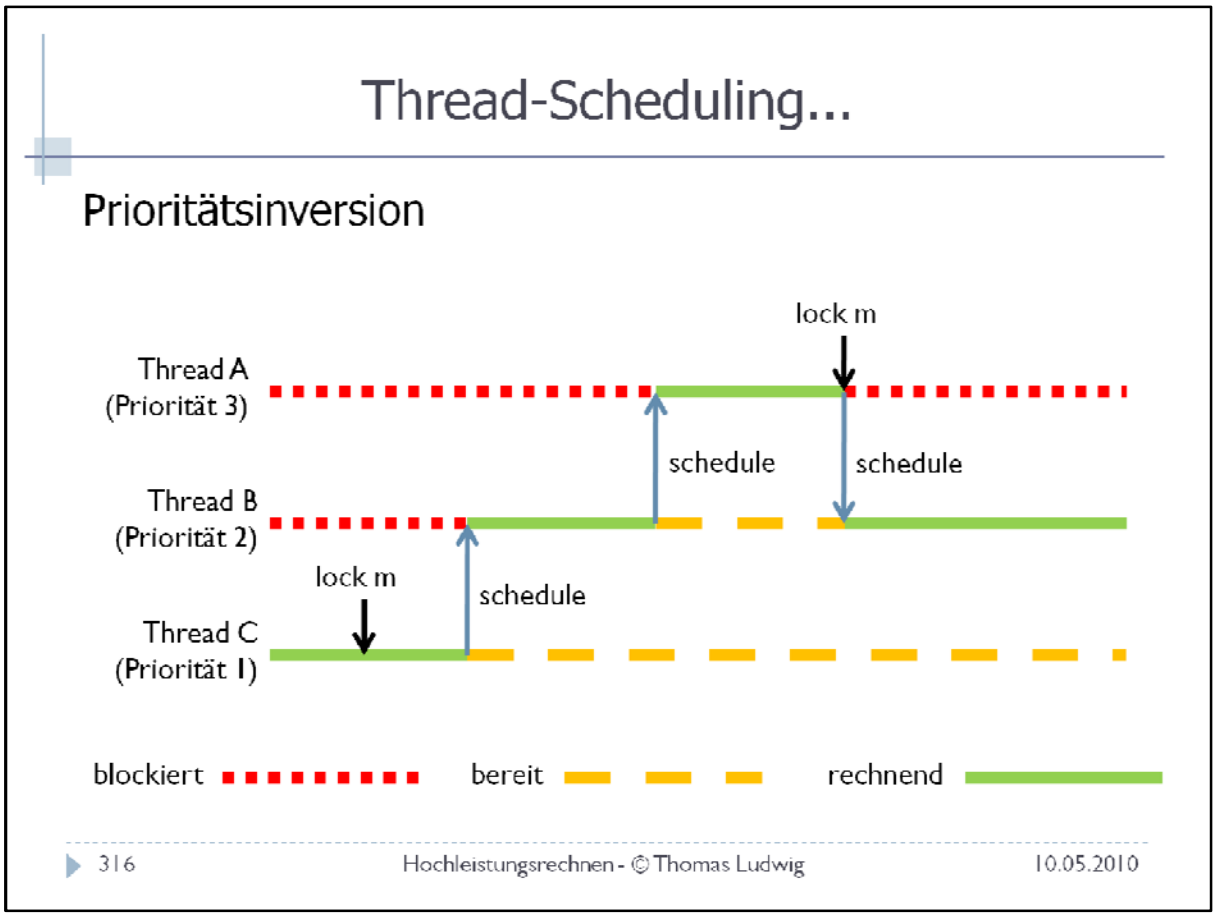
- Üblicherweise: Scheduling mit Prioritäten
- Ziel: Threads mit höchster Priorität rechnen
- Strategie bei gleichen Prioritäten
  - FIFO (First-In-First-Out)
    Unterbrechnung nur durch höherpriore Threads
  - RR (Round Robin)
    Unterbrechung und Weiterschalten nach Ablauf einer Zeitscheibe

Thread-Scheduling...

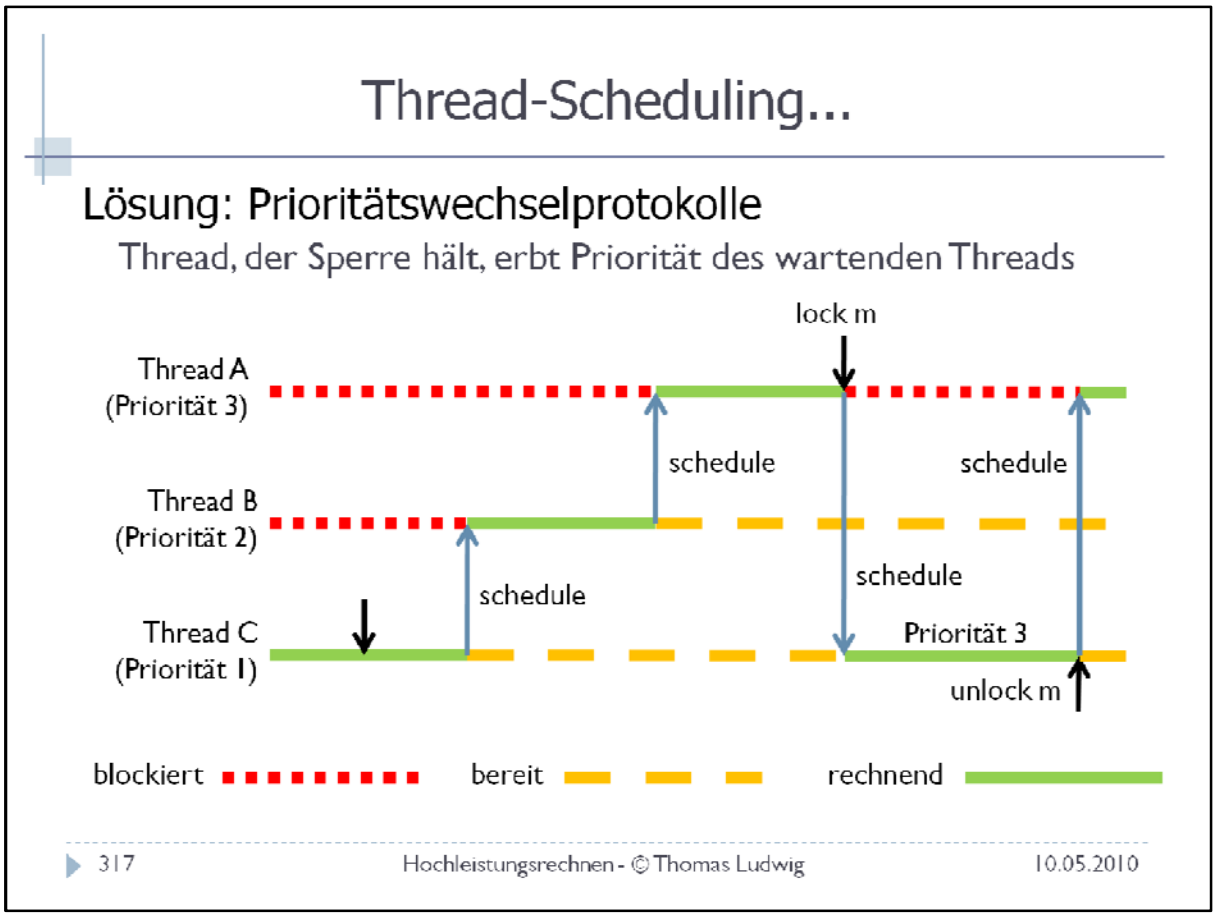
Probleme

- Fairness bei Mehrbenutzersystemen mit FIFO

  Bösartiger Benutzer kann System blockieren
- Abhilfe: Zeitscheiben und dynamische Änderung der Prioritäten durch das Betriebssystem
  - Durch Rechnen: Priorität ↓, Zeitscheibe ↑
  - Durch Warten: Priorität ↑, Zeitscheibe ↓
- Niederpriore Threads können höherpriore blockieren
  - Prioritätsinversion



Thread C kommt erst einmal nicht an die Reihe und blockiert damit das Weiterlaufen von Thread A, der ja die höhere Priorität hätte.



Thread C kann weiterarbeiten und gibt in der Folge die Sperre wieder frei.

Thread-Scheduling...

Komplikationen

- Manche Threads müssen auf einem bestimmten Prozessor bleiben (z.B. Interrupt-Threads)
- Interaktion mit Caches
  - Effizienzverlust, wenn Thread wandert

    Konzept: Prozessoraffinität
- Bei starker Synchronisation zwischen Threads
  - Effiziente Abarbeitung nur, wenn alle Threads eines Prozesses gleichzeitig laufen

    Konzept: Gangscheduling

Thread-Programmierung

Nochmal im Überblick

- Prozesse
  - Jeder Prozeß hat eigenen Adreßraum
  - Kommunikation nur mit Betriebssystem-Unterstützung

    Signale, pipes, sockets, streams

    Gemeinsame Speicherbereiche
- Threads
  - Gemeinsamer Programmcode für alle Threads
  - Gemeinsamer Speicher, E/A, usw.
  - Koordination (Synchronisation) wesentlich!

Thread-Programmierung...

Varianten der Programmierung von SMPs

- Unabhängige Prozesse
  - Z.B. bei WWW- und Datei-Servern (`fork()`)
  - Keine spezielle Programmierung nötig
  - Wechselseitiger Schutz der Server-Prozesse
  - Hohe Antwortzeiten
- Kommunizierende Prozesse
  - Wenn Kooperation **und** Schutz erforderlich sind
    Z.B. X Window Client und Server
  - Kommunikation aufwendig und unkomfortabel

Thread-Programmierung...

- Threads
  - Bei allen Arten von Servern
  - Für parallele Programme
    - Hohe Effizienz
    - Einfache Kooperation
    - Kein wechselseitiger Schutz durch Betriebssystem
    - Korrekte Synchronisation schwieriger

Thread-Programmierung...

Im folgenden

- Einführung in POSIX-Threads (Pthreads)
  - Standard IEEE P1003.1c
  - In vielen Systemen realisiert
  - Konzepte in anderen Thread-Realisierungen meist ähnlich
  - POSIX-konforme Realisierungen unter Linux

Die Pthread-Schnittstelle

Eingeteilt in drei (informelle) Klassen

- Thread-Verwaltung
- Mutex-Verwaltung
- Bedingungsvariablen-Verwaltung

Siehe: https://www.llnl.gov/computing/tutorials/pthreads/ – sehr gutes Tutorial zum Thema

Tippe unter Linux: `man pthreads`

Die Pthread-Schnittstelle...

Thread-Erzeugung

- Programmiermodell
  - Bei Start eines Prozesses existiert genau ein Thread
  - Dieser erzeugt ggf. weitere Threads und wartet auf deren Ende
  - Terminierung des Prozesses bei Terminierung des Master-Thread
- Funktion zur Thread-Erzeugung

```
int pthread_create(pthread_t *new_thrd_ID,
  const pthread_attr_t *attr,
  void *(*start_func)(void *),
  void *arg)
```


Das Kreieren eines Thread liefert eine ID zurück. Man übergibt als Parameter die Attribute für diesen Thread, einen Zeiger auf eine Funktion, die den Thread-Code darstellt und die Parameter für diesen Thread.

Die Pthread-Schnittstelle...

- Thread-Attribute
  - Keller und Kellergröße
  - Scheduling-Schema und Priorität
  - `detachstate`:Verhalten bei Beendigung des Threads

    Bei `detached thread` erfolgt die Freigabe der Ressourcen sofort bei Beendigung, ansonsten erst nach Abschlußsynchronisation

Die Pthread-Schnittstelle...

Thread-Attribute

- **`pthread_attr_init`**
  Attributstruktur initalisieren
- **`pthread_attr_destroy`**
  Attributstruktur löschen
- **`pthread_attr_get / pthread_attr_set`**
  Lesen und setzen von Attributwerten

Die Pthread-Schnittstelle...

Thread-Verwaltung

- **`pthread_self`**
  Liefert eigene Thread-ID
- **`pthread_exit`**
  Eigene Terminierung
- **`pthread_cancel`**
  Terminiert anderen Thread
  - Kann maskiert werden
  - Terminierung nur an bestimmten Punkten
  - Vor Terminierung **`cleanup handler`** aufrufen

Die Pthread-Schnittstelle...

Thread-Verwaltung...

- **`pthread_join`**

  Wartet auf Terminierung des spezifizierten Threads (Abschlußsynchronisation)

- **`pthread_sigmask`**

  Setzt Signalmaske (jeder Thread hat eigene Maske)

- **`pthread_kill`**

  Sendet Signal an anderen Thread innerhalb des Prozesses

  - Von außen nur Signal an *irgendeinen* Thread möglich

Die Pthread-Schnittstelle...

Mutex-Operation (wechselseitiger Ausschluß)

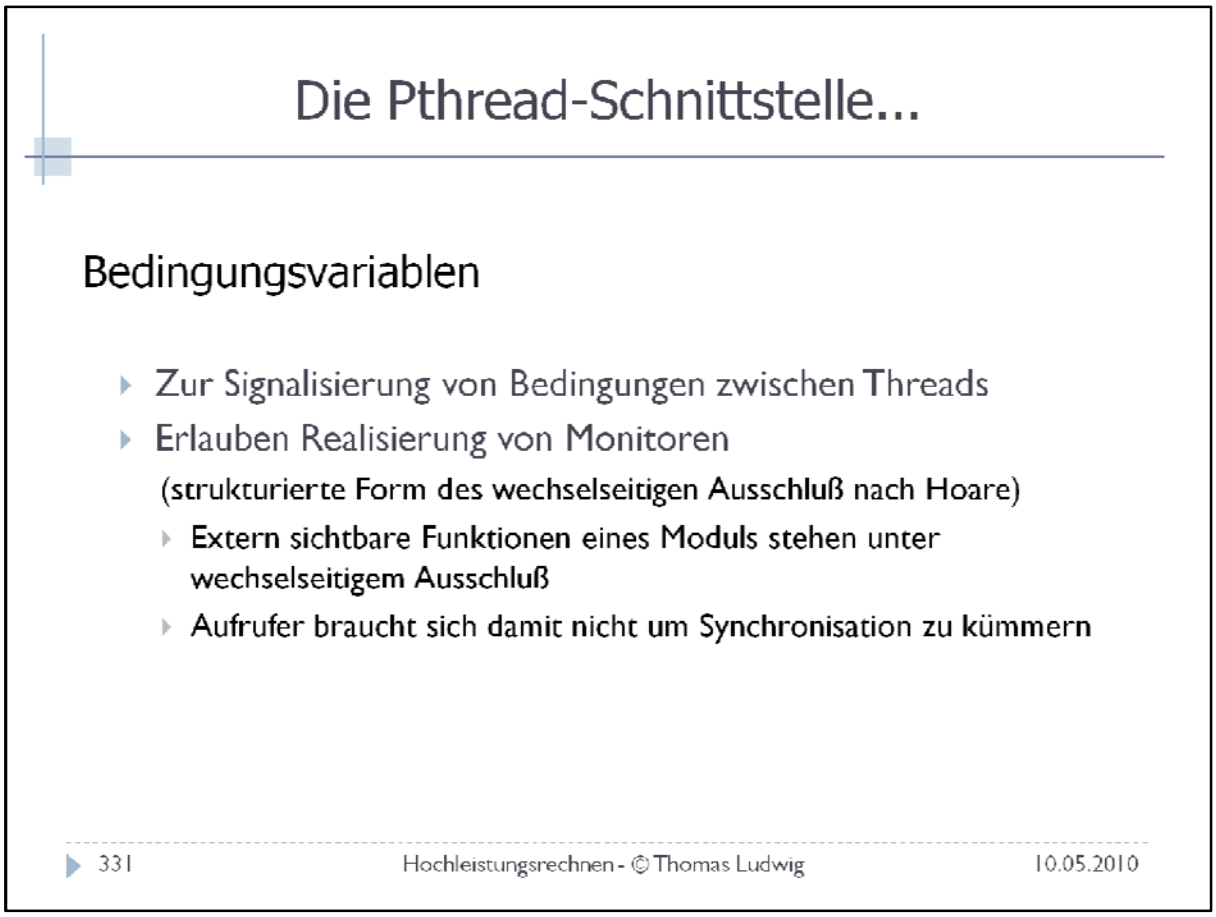
- **`pthread_mutex_init`**
  Initialisiert Sperrvariable (mutex)
- **`pthread_mutex_destroy`**
- **`pthread_mutex_lock`**
  Blockiert Thread, bis Sperre frei ist (a) und belegt dann die Sperre (b)
- **`pthread_mutex_trylock`**
  Belegt Sperre, falls möglich / kein Blockieren
- **`pthread_mutex_unlock`**

Mutex wird verwendet, wenn z.B. mehrere Schreiber auf eine Variable zugreifen. Hierdurch wird die Variable geschützt, so daß sie immer nur ein Schreiber zu einem Zeitpunkt manipulieren kann.

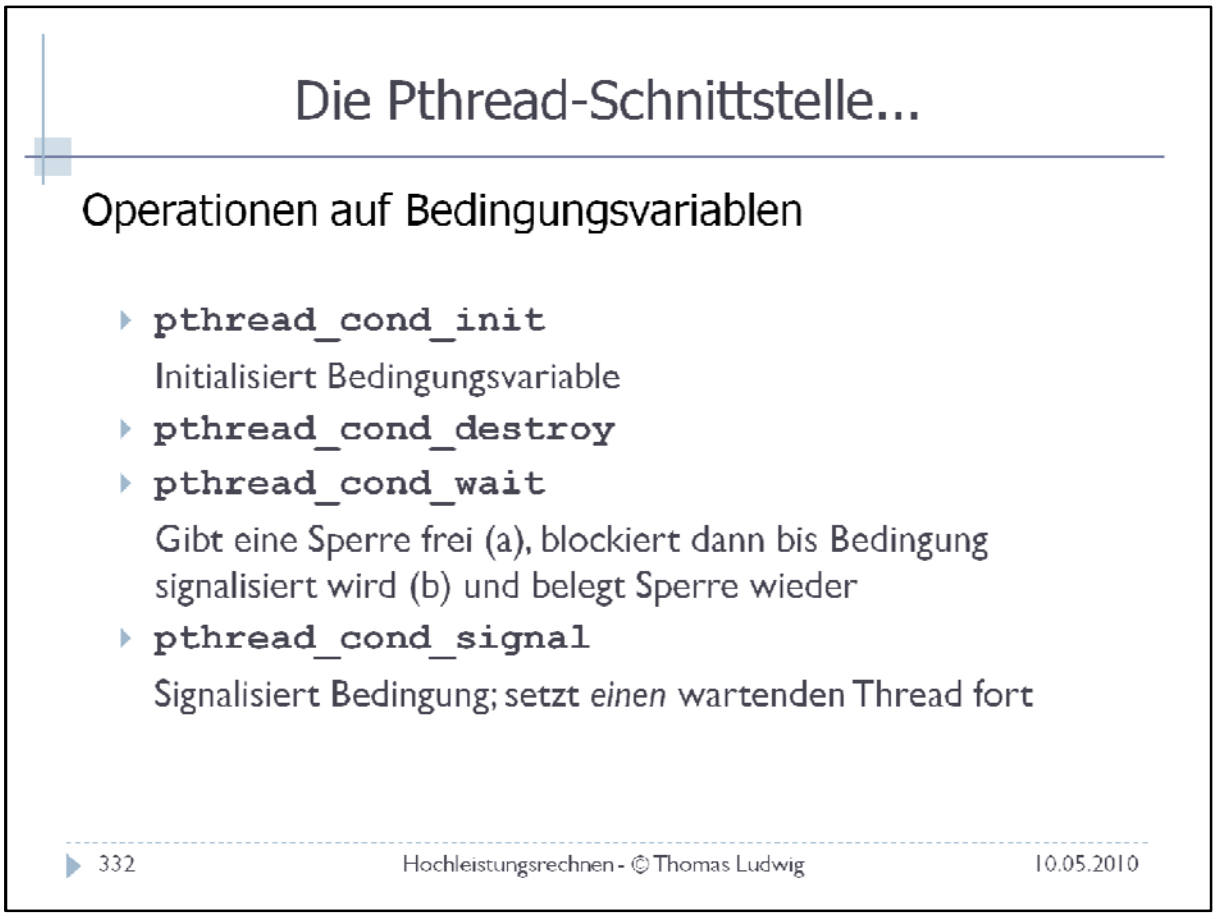
Die Pthread-Schnittstelle...

Anmerkungen zu Mutex-Operationen

- Operationenpaar (a), (b) muß unteilbar sein
- Bei Terminierung eines Threads werden Sperren nicht automatisch freigegeben
- Prioritätswechselprotokolle in einigen Implementierungen



Ein Monitor verwaltet z.B. Daten und eine Zugriffsfunktion (so etwa wie eine Klasse bei OO-Programmierung). Intern wird vom Monitor durch einen Thread sichergestellt, daß nur jeweils ein Aufrufer zu einem Zeitpunkt die internen Daten manipulieren kann.



Die zu lösende Situation besteht z.B. darin, daß ein Thread eine Aktivität ausführen soll, wenn eine bestimmte Bedingung erfüllt ist, z.B. der Wert einer Variablen überschreitet einen Grenzwert.

Die Lösung mit Mutexen sieht so aus, daß dieser Thread immer wieder Zugriff auf die Variable erwirbt und dann prüft und gegebenenfalls seine Aktionen ausführt. Das ist sehr kostspielig.

Mit Bedingungsvariablen wartet er blockierend darauf, daß ihm **ein anderer** Thread signalisiert, daß die Bedingung erfüllt ist. Die Signalisierung wird nur dort potentiell ausgelöst, wo der Wert der Variablen durch einen anderen Thread erhöht wird.

Die Pthread-Schnittstelle...

Operationen auf Bedingungsvariablen...

- **`pthread_cond_timewait`**
  Wie **`wait`** aber mit begrenzter Wartezeit
- **`pthread_cond_broadcast`**
  Wie **`signal`**, aber mit Fortsetzung aller wartenden Threads

Die Pthread-Schnittstelle...

Amerkungen zu Bedingungsvariablen

- Operationspaar (a), (b) in pthread_cond_wait (Freigabe des Mutex, Warten auf Bedingung) muß unteilbar implementiert sein
- Bedingungsvariable „merken“ sich die Signalisierung nicht
  - Wenn bei Signalisierung kein wartender Thread existiert, bleibt sie ohne Wirkung
  - Auch dann, wenn später ein Thread auf die Bedingung wartet
- Signalisierung muß bei belegtem Mutex erfolgen

Linux und Threads

Linux Thread-Bibliotheken

- Next Generation Posix Threading (NGPT)
  - nicht mehr weitergeführt (!)
- LinuxThreads
  - nicht mehr weitergeführt (!)
- The Native Posix Thread Library (NPTL)
  - dies ist jetzt im Kernel 2.6 die Standard-Implementierung für die POSIX-Threads
  - ein POSIX-Thread wird auf einen Linux-Threads abgebildet

Siehe: http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

Linux und Threads...

Linux-Threads

- Spezielle Variante der allgemeinen Prozesse
- Schnell erzeugbar, effiziente Nutzung
- Teilen sich alle Ressourcen mit Eltern-Prozeß
- Erzeugt mit `clone()`-Aufruf (wie Kindprozesse auch)
  - Andere Parameter gestatten gemeinsame Ressourcennutzung

Kernel-Threads in Linux

- Spezielles Thread-Objekt im Betriebssystemkern
- Kein eigener Adreßraum
- Arbeiten nur im Betriebssytemmodus
- Sind allerdings dem Scheduler unterworfen
- Zur Strukturierung von Aktivitäten im Kernel

Vergleich der Ansätze

	Pthreads	MPI
Skalierbarkeit	Begrenzt	Ja
Fortran / C	Ja? / Ja	Ja / Ja
Hohe Abstraktion	Nein	Nein
Leistungsorientierung	Nein	Ja
Portierbarkeit	Ja	Ja
Herstellerunterstützung	Unix/SMP	Verbreitet
Inkrem. Parallelisierung	Nein	Nein

Inkrementelle Parallelisierung bedeutet: man kann das Programm stückweise von einem sequentiellen in ein paralleles Programm umbauen. Z.B. je nach Zeit, die einem zur Verfügung steht. Dies geht hier in beiden Fällen nicht. Man kann nur entweder ein richtiges voll parallelisiertes Programm erstellen, oder man arbeitet mit dem sequentiellen weiter.

Programmierung mit Threads
Zusammenfassung

- Threads trennen Einheiten der Ressourcen-belegung von Einheiten der Prozessorzuteilung
- Threads können auf der Ebene des Benutzers und des Systems realisiert werden
- In der Praxis finden wir hybride Ansätze (1:1)
- Das Scheduling von Threads ist komplexer als das von Prozessen
- Threads werden für echt parallele Programme aber auch als Strukturierungsmittel eingesetzt
- Die Pthreads-Schnittstelle definiert einen Standard zur Nutzung von Threads