Programmierung mit Threads

1. Prozesse und Threads
2. Thread-Programmierung
3. Die Pthreads-Schnittstelle
4. Thread-sichere Programmierung
5. Threads und Linux
6. Bewertung

1. 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 (eine Art Unterprozess)
  (auch bezeichnet als „leichtgewichtiger Prozess")
  - Aufspaltung dieser Eigenschaften:
    Prozess ist Einheit der Ressourcenbelegung
    Thread ist Einheit der Prozessorzuteilung

Prozesse und Threads (2)

Eigenschaften von Threads

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

Prozesse und Threads (3)

Bedeutung von Threads

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

Threads und Hochleistungsrechnen?

Fakten

- Alle modernen Betriebssysteme können mit Threads in Prozessen umgehen
- Sie bilden diese auf die Prozessorkerne des Systems ab
- Moderne Bibliotheken arbeiten mit Threads oder müssen zumindest thread-sicheren Code implementieren
  - Details später
- Der Compiler für OpenMP-Programme erzeugt Threads
  - Leider nicht auf eine einfach nachvollziehbaren Weise

2. Thread-Programmierung

Nochmal im Überblick

- Prozesse
  - Jeder Prozess hat eigenen Adressraum
  - Kommunikation zwischen Prozessen nur mit Betriebssystem-Unterstützung
    - signals, pipes, sockets, streams
    - shared memory segments
- Threads
  - Nutzen gemeinsam den Adressraum ihres Prozesses
  - Nutzen auch alle anderen Ressourcen ihres Prozesses gemeinsam: offene Dateien, sockets zur Kommunikation mit anderen Prozessen usw.
  - Müssen sorgfältig koordiniert werden, um falsche Programmergebnisse zu verhindern

Thread-Programmierung (2)

Drei Varianten der Programmierung von Mehrkernsystemen

- 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
    z.b. die Implementierung von Copy&Paste-Mechanismen
  - Kommunikation aufwendig und unkomfortabel

Thread-Programmierung (3)

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

Thread-Programmierung (4)

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

3. Die Pthreads-Schnittstelle

Eingeteilt in drei (informelle) Klassen

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

Die Pthreads-Schnittstelle (2)

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)
```

Die Pthreads-Schnittstelle (3)

```
void* print_hello (void* threadid)
{ printf("Hello world from thread #%d!\n",
         (int)threadid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{ pthread_t threads[NUM_THREADS];
  for (int t = 0; t < NUM_THREADS; t++) {
    printf("In main: creating thread %ld\n", t);
    pthread_create(&threads[t], NULL,
                   print_hello, (void*)t);
  }
  for (int t = 0; t < NUM_THREADS; t++) {
   pthread_join(threads[t], NULL);
  }
}
```

Die Pthreads-Schnittstelle (4)

Mögliche Ausgabe des Programms bei `NUM_THREADS=5`

```
In main: creating thread 0
In main: creating thread 1
Hello world from thread #0!
In main: creating thread 2
Hello world from thread #2!
Hello world from thread #1!
In main: creating thread 3
Hello world from thread #3!
In main: creating thread 4
Hello world from thread #4!
```

Die Pthreads-Schnittstelle (5)

Thread-Attribute (1)

- Keller und Kellergröße
- Scheduling-Schema und Priorität
- Affinität
- **`detachstate`**: Verhalten bei Beendigung des Threads

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

Die Pthreads-Schnittstelle (6)

Thread-Attribute (2)

- **`pthread_attr_init`**
  Attributstruktur initalisieren
- **`pthread_attr_destroy`**
  Attributstruktur löschen
- **`pthread_attr_get`***XYZ* **/ `pthread_attr_set`***XYZ*
  Lesen und setzen von Attributwerten
  XYZ = **`stacksize`**, **`schedpolicy`**, ...

Die Pthreads-Schnittstelle (7)

Thread-Verwaltung (1)

- **`pthread_self`**
  Liefert eigene Thread-ID
- **`pthread_exit`**
  Eigene Terminierung
- **`pthread_cancel`**
  Terminiert anderen Thread
  - Kann maskiert werden
  - Terminierung asynchron oder an bestimmten Punkten

Die Pthreads-Schnittstelle (8)

Thread-Verwaltung (2)

- **`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 Pthreads-Schnittstelle (9)

Thread-Synchronisation durch eine Barriere

- **`pthread_barrier_init(…,`*`count`*`)`**

  Erzeuge und initiiere eine Barriere und spezifiziere die Anzahl der Threads, die über sie synchronisiert werden sollen

- **`pthread_barrier_wait`**

  Jeder zu synchronisierende Thread ruft die Funktion auf und wird schlafend gelegt bis die vorher spezifizierte Anzahl von Threads in die Barriere eingelaufen sind. Danach werden alle fortgesetzt. Einer(!) der fortgesetzten Threads erhält den reservierten Rückgabewert `PTHREAD_BARRIER_SERIAL_THREAD`. Man nutzt diesen, um Aktionen zu steuern, die nur einer durchführen soll, wie z.B. das Melden des Verlassens der Barriere. Alle anderen erhalten den Rückgabewert 0.

- **`pthread_barrier_destroy`**

  Lösche die Barriere mit ihren Ressourcen

Die Pthreads-Schnittstelle (10)

Mutex-Operation

- wechselseitiger Ausschluss, mutual exclusion
- Z.B. Schutz von (globalen) Variablen bei mehreren Schreibern

- **`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`**

Die Pthreads-Schnittstelle (11)

Anmerkungen zu Mutex-Operationen

- Operationenpaar (a), (b) muss unteilbar sein
- Bei Terminierung eines Threads werden Sperren nicht automatisch freigegeben

Die Pthreads-Schnittstelle (12)

Bedingungsvariable

- Zur Signalisierung von Bedingungen zwischen Threads
- Erlauben Realisierung von Monitoren

  (strukturierte Form des wechselseitigen Ausschluss nach Hoare)
  - Extern sichtbare Funktionen eines Moduls stehen unter wechselseitigem Ausschluss
  - Aufrufer braucht sich damit nicht um Synchronisation zu kümmern

Die Pthreads-Schnittstelle (13)

Verwendungszweck

- Ein Thread soll eine Aktivität ausführen, wenn eine Bedingung erfüllt ist, z.B. Variable>Grenzwert
- Mit Mutex: Thread prüft regelmäßig Variable
  - Sehr kostspielig, ähnlich spinlock im Betriebssystem
- Mit Bedingungsvariable: Thread wird blockiert, bis ihm ein **anderer** Thread signalisiert, dass die Bedingung erfüllt ist
  - Etwas ähnlich zu Semaphor-Objekt

Die Pthreads-Schnittstelle (14)

Operationen auf „Bedingungsvariablen"

das ist nicht die Variable der Bedingung, die uns interessiert, **sondern** das Objekt der Pthreads-Bibliothek zur Steuerung !

- **`pthread_cond_init`**
  Initialisiert Bedingungsvariable
- **`pthread_cond_destroy`**
- **`pthread_cond_wait (`***`cond_var,mutex`***`)`**
  Gibt eine Sperre frei (a), blockiert dann bis Bedingung signalisiert wird (b) und belegt Sperre wieder
- **`pthread_cond_signal (`***`cond_var`***`)`**
  Signalisiert Bedingung; setzt (mindestens) einen wartenden Thread fort

Die Pthreads-Schnittstelle (15)

Operationen auf Bedingungsvariablen...

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

Die Pthreads-Schnittstelle (16)

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
  - Threads können fälschlicherweise aufgeweckt werden ???
- Bedingungsvariable sind opake Datentypen
- Signalisierung sollte bei belegtem Mutex erfolgen

Die Pthreads-Schnittstelle (17)

```
void* producer (void)
{
  mutex_lock(&m);
  while (true)
  {
    while (produced > n)
      cond_wait(&c, &m);

    produce();
    produced++;
    cond_signal(&c);
  }
  mutex_unlock(&m);
}
```

```
void* consumer (void)
{
  mutex_lock(&m);
  while (true)
  {
    while (produced == 0)
      cond_wait(&c, &m);

    consume();
    produced--;
    cond_signal(&c);
  }
  mutex_unlock(&m);
}
```

4. Thread-sichere Programmierung

Definition

- Eine Funktion kann gleichzeitig von mehreren Threads ohne gegenseitige Behinderung ausgeführt werden

Implementierungsansätze ohne gemeinsame Variable

- Wiedereintrittsfähigkeit (Re-entrancy)
- Thread-lokaler Speicher
- Unveränderbare Objekte

Implementierungsansätze mit gemeinsamen Variablen

- Gegenseitiger Ausschluss
- Atomare Operationen

Thread-sichere Programmierung (2)

```
// thread-safe but not re-entrant
int increment_counter (void)
{
  static int counter = 0;
  static mutex_t mutex = MUTEX_INIT;
  int result;

  mutex_lock(&mutex);

  result = counter++;

  mutex_unlock(&mutex);

  return result;
}
```

Thread-sichere Programmierung (3)

```
// thread-safe and re-entrant
int increment_counter (void)
{
  static int counter = 0;

  result = atomic_inc(&counter);

  return result;
}
```

5. Threads und Linux

Linux Thread-Bibliothek

- The Native POSIX Thread Library (NPTL)
  - Entwickelt von Red Hat, seit Kernel 2.6 unterstützt
  - Standard-Implementierung für POSIX-Threads und Teil der glibc
  - Ein POSIX-Thread wird auf einen Linux-Thread abgebildet

Threads und Linux (2)

Linux-Threads

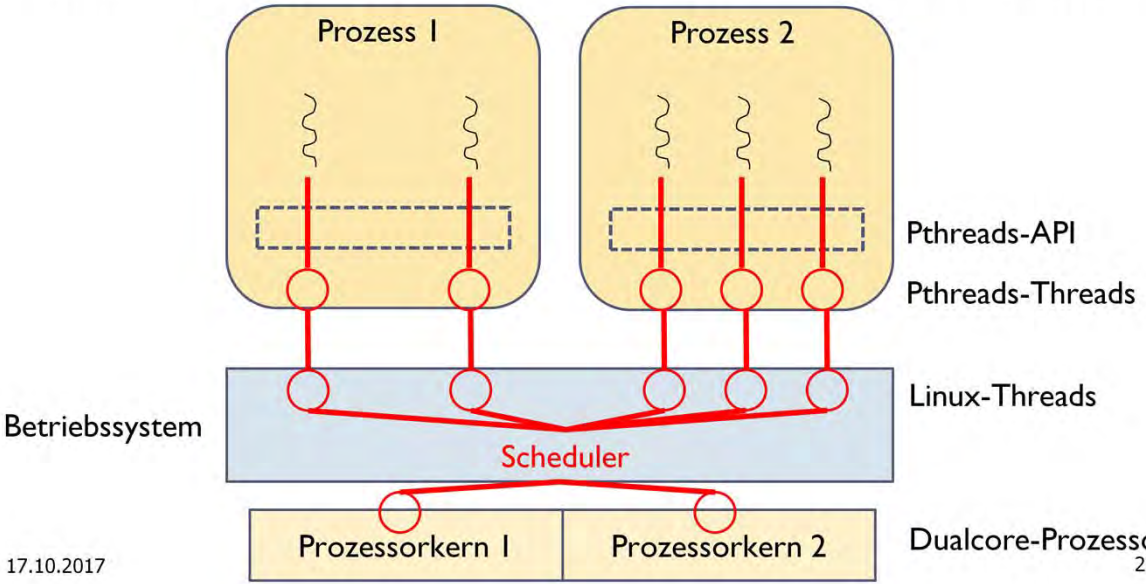
- Spezielle Variante der allgemeinen Prozesse
- Schnell erzeugbar, effiziente Nutzung
- Teilen sich alle Ressourcen mit Eltern-Prozess
- Erzeugt mit `clone()` -Aufruf (wie Kindprozesse auch)
  - Andere Parameter gestatten gemeinsame Ressourcennutzung

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

Threads und Linux (3)

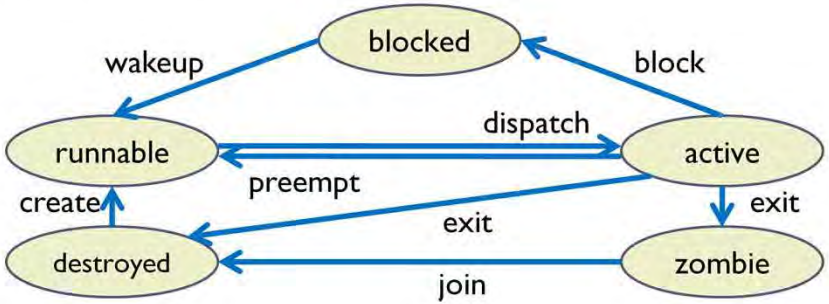
Pthreads-Threads – Linux-Threads - Prozessorkerne



Threads und Linux (4)

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

  (POSIX-Implementierungsmodell)



6. Bewertung

	Pthreads	OpenMP	MPI
Skalierbarkeit	Begrenzt	Begrenzt	Ja
Fortran / C und C++	Nein / Ja	Ja / Ja	Ja / Ja
Hohe Abstraktion	Nein	Ja	Nein
Leistungsorientierung	Nein	Ja	Ja
Portierbarkeit	Ja	Ja	Ja
Herstellerunterstützung	Unix/SMP	Verbreitet	Verbreitet
Inkrement. Parallelisierung	Nein	Ja	Nein

Programmierung mit Threads
Zusammenfassung

- Threads trennen Einheiten der Ressourcenbelegung (Prozesse) von Einheiten der Prozessorzuteilung
- Threads werden für echt parallele Programme aber auch als Strukturierungsmittel eingesetzt
- Threads teilen sich den Adressraum des sie erzeugenden Prozesses
- Die Pthreads-Schnittstelle definiert einen Standard zur Nutzung von Threads
- Pthreads werden auf die Threads des Betriebssystems umgesetzt, die dann auf die Prozessorkerne des Systems abgebildet werden
- Alle Bibliotheken müssen heutzutage thread-sicher sein

Programmierung mit Threads
Die wichtigsten Fragen

- Worin unterscheiden sich Threads und Prozesse?
- Warum sind Threads ein Thema beim Hochleistungsrechnen?
- Wann programmieren wir mit Threads?
- Welche Grundoperationen bietet die Pthreads-Schnittstelle?
- Wie werden Threads erzeugt?
- Wie funktioniert ein Mutex?
- Was versteht man unter thread-sicheren Funktionsaufrufen?
- Wie sind Threads im Linux-Betriebssystemkern genutzt?