Pthreads

David Klaftenegger

Seminar: Multicore Programmierung
Sommersemester 2009

16.07.2009

Inhaltsverzeichnis

Threads

Was sind Threads

- innerhalb eines Prozesses
- unabhängige Ausführungsstränge
- „leichtgewichtige Prozesse"

Scheduling

- `1:1` - Thread = Prozess
- `N:M` - Threads auf virtuellen Prozessoren
- `N:1` - Threads in Software (keine Parallelität)

Pthreads

Was ist Pthreads

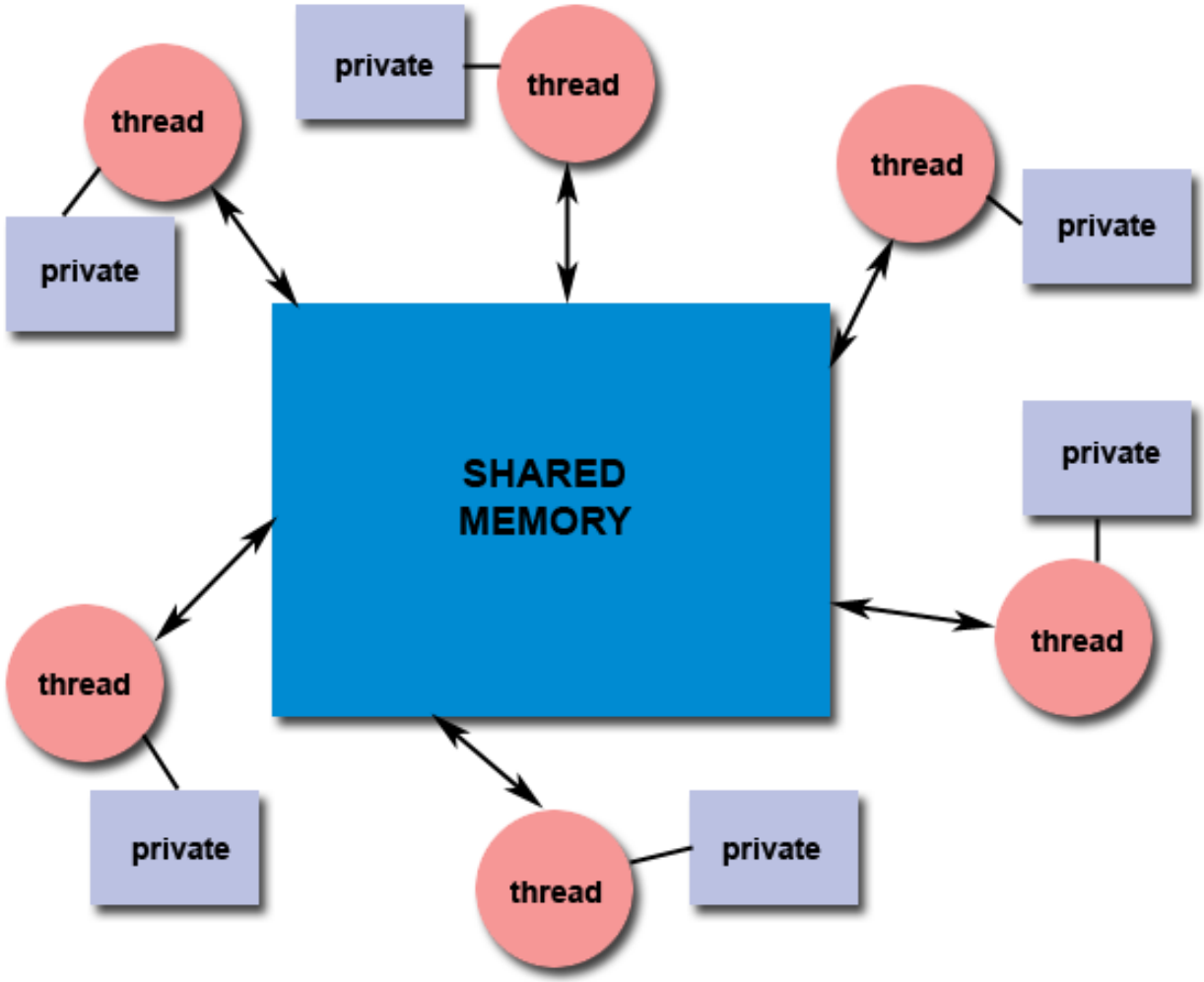
- POSIX Threads
- IEEE Standard 1003.1c (1995)
- C-API

Pthreads

Implementierungen

- LinuxThreads (`1:1`, veraltet)
- NPTL (Linux, `1:1`)
- KSE (FreeBSD, `N:M`)
- libthr (FreeBSD, `1:1`)
- pthreads-w32 (Windows)
- GNU Portable Threads (plattformunabhängig, `N:1`)

Speichermodell



Speichermodell

privater Speicher

- Stack
- keys (Thread-spezifische Zeiger)

gemeinsamer Speicher

- alles Andere

Thread-Erstellung

Code

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

- erzeugt neuen Thread
- benötigt Repräsentanten
- startet Funktion in neuem Thread
- übergibt genau einen Parameter an die Funktion

Thread-Erstellung

Beispiel

```
void* do_work(void* param) {
  int i, a = 0;
  for(i = 0; i <= 100; i++) {
    a += i;
  }
  printf("computed %d\n", a);
}
```

Beispiel

```
  int rc = pthread_create(&thread, NULL, do_work,
      NULL);
```

Thread-Ende

Code

```
void pthread_exit(void* value_ptr);
```

- beendet einen Thread
- speichert einen Zeiger als Thread-Ergebnis

Thread-Ende

Beispiel

```
void* do_work(void* param) {
  int i;
  int a = 0;
  for(i = 0; i <= 100; i++) {
    a += i;
  }
  pthread_exit((void*)a);
}
```

Beispiel

```
  pthread_exit(NULL);
```

Thread-Synchronisation: Join

Code

```
int pthread_join(pthread_t thread, void** value_ptr);
```

- wartet auf eine Thread
- speichert Rückgabe-Zeiger in übergebener Adresse

Beispiel

```
void* a;
pthread_join(thread, &a);
int i = a;
printf("computed %d\n", i);
```

Mutex

Fehler

```
int global;

void* add50000(void* param) {
  int i;
  for(i = 0; i < 50000; i++) {
    global += 1;
  }
}
```

wechselseitiger Ausschluss

- Mutex (von mutual exclusion)
- Serialisierung kritischer Abschnitte

Mutex-Initialisierung

Code

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

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

- Objekt für wechselseitigen Ausschluss
- Initialisierung mit Makro oder Funktionsaufruf

Mutex-Verwendung

Code

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

- kritischer Abschnitt: Mutex sperren
- gesperrter Mutex: exklusiver Zugriff auf die geschützte Resource
- keine Überprüfung, dass Zugriff überall durch Mutex geschützt ist
- am Ende des kritischen Abschnitts: Mutex freigeben

Mutex-Verwendung

Beispiel

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int global;

void* add50000(void* param) {
  int i;
  pthread_mutex_lock(&mutex);
  for(i = 0; i < 50000; i++) {
    global += 1;
  }
  pthread_mutex_unlock(&mutex);
}
```

Condition

Warten auf Bedingung

- oft kann ein kritischer Abschnitt erst begonnen werden, wenn eine Bedingung erfüllt ist
- periodisches Vorgehen bremst das Programm aus
  - Mutex sperren
  - Bedingung überprüfen
  - Mutex freigeben oder kritischen Abschnitt bearbeiten

Lösung

Mit Conditions (Bedingungen) stellt Pthreads einen Mechanismus bereit um Threads aufzuwecken, die auf eine bestimmte Bedingung warten

Condition-Initialisierung

Code

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_init(pthread_cond_t* restrict cond,
                      const pthread_condattr_t*
                          restrict attr);
```

- Objekt für Condition
- Initialisierung mit Makro oder Funktionsaufruf

Condition-Verwendung

Code

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

- aktueller Thread wartet auf Bedingung
- Mutex wird freigegeben
- beim Aufwachen wird Mutex sofort wieder gesperrt

Condition-Verwendung

Code

```
int pthread_cond_signal(pthread_cond_t* cond);
int pthread_cond_broadcast(pthread_cond_t* cond);
```

- teilt anderen Threads mit, dass die Bedingung erfüllt sein kann
- signal: weckt mindestens einen Thread
- broadcast: weckt alle Threads

Condition-Verwendung

Beispiel

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int global;
```

```
void* print_global(void* param) {
  while(global < 50000) {
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cond, &mutex);
    printf("global=%d\n", global);
    pthread_mutex_unlock(&mutex);
  }
}
```

Condition-Verwendung

Beispiel

```
void* add50000(void* param) {
  int i,h;
  for(i = 0; i < 10; i++) {
    pthread_mutex_lock(&mutex);
    for(h = 0; h < 5000; h++) {
      global += 1;
    }
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    sleep(1);
  }
}
```

Thread-Abbruch

Notwendigkeit?

- Spekulation auf Fadenebene
- Alternative:
  Thread prüft selbst, ob er noch weiter ausgeführt werden soll

Code

```
int pthread_cancel(pthread_t thread);
```

- bricht den durch thread repräsentierten Thread ab

Abbruch-Arten

Code

```
int pthread_setcancelstate(int state, int* oldstate);
int pthread_setcanceltype(int type, int* oldtype);
```

- Threads können als nicht-abbrechbar markiert werden
  - pthread_cancel wird vom Thread ignoriert
- Threads können verzögert oder sofort abgebrochen werden
  - verzögert: Ausführung bis zum nächsten cancellation point
  - sofort: inkonsistenter Datenzustand leicht möglich

Aufräumen

Code

```
void pthread_cleanup_push(void (*routine)(void*),
                          void* arg);
void pthread_cleanup_pop(int execute);
```

- Stack für Aufräum-Routinen
- push: neue Routine auf den Stack legen
- pop: neueste Routine wieder vom Stack entfernen
- im Falle eines Thread-Abbruchs werden die Routinen von oben nach unten abgearbeitet

Implementierungsvielfalt

nicht überall unterstützte Features

- Thread-Prioritäten
- Mutex-Attribute
- Condition-Zugriff aus verschiedenen Prozessen

Unterschiede

- Standard-Stackgröße pro Thread
- Maximalanzahl Threads
- Verhalten bei fehlerhaften Eingaben

Prioritätsinversion

Prioritätsinversion

Wenn Threads mit unterschiedlicher Priorität laufen, so kann es dazu kommen, dass ein Thread mit niedriger Priorität verhindert, dass ein Thread mit höherer Priorität ausgeführt wird.

Beispiel: Mutex

- Ein Thread mit niedriger Priorität sperrt einen Mutex, aufgrund seiner niedrigen Priorität wird er nicht weiter ausgeführt.
- Ein Thread mit hoher Priorität wartet auf die Freigabe des Mutex.

Prioritätsinversion

Lösung für das Beispiel

- Prioritätsschranke für Mutex
- sperrender Thread erhält für die Dauer der Sperre höhere Priorität
- funktioniert nur, wenn kein Thread mit höherer Priorität als der Schranke wartet
- wird nicht von allen Implementierungen angeboten

Alternativen zu Pthreads

Solaris Threads

- ähnlicher Funktionsumfang
- andere Funktionsnamen
- kann mit Pthreads kooperativ benutzt werden

Boost.Thread

- Thread-Bibliothek für C++
- verwendet Thread-Unterbau des Betriebssystems
- nicht alle Funktionen aus Pthreads vorhanden
- zusätzliche Strukturen um das Programmieren zu erleichtern
- wird wahrscheinlich durch C++0x überflüssig

Alternativen zu Pthreads

OpenMP

- einfach zu bedienen
- weniger flexibel
- Anzahl der Threads muss nicht vom Programmierer festgelegt werden

MPI

- verteilter Speicher
- kollektive Kommunikation
- schwer zu bedienen

Literatur


Blaise Barney.
POSIX Threads Programming.
Website, 2009.
Available online at
https://computing.llnl.gov/tutorials/pthreads/.