Pthreads

David Klaftenegger

Seminar: Multicore Programmierung
Sommersemester 2009

16.07.2009

Inhaltsverzeichnis

1 Einführung **3**
1.1 Pthreads 3
1.2 Speichermodell 4

2 Pthreads - API **5**
2.1 Thread-Management 5
2.2 Mutex 7
2.3 Conditions 10
2.4 Thread-Abbruch 13

3 Probleme **14**
3.1 Implementierungsvielfalt 14
3.2 Prioritätsinversion 15

4 Alternativen zu Pthreads **15**

Literatur **17**

1 Einführung

Der Begriff *Threads* (deutsch: Fäden) bezeichnet im Allgemeinen unabhängige Ausführungsstränge eines Programms. Sie werden auch als *leichtgewichtige Prozesse* beschrieben, da sie Prozessen ähneln, aber mit geringerem Aufwand erzeugt werden können.
Jeder Thread verfügt über einen eigenen Stack für die unabhängige Ausführung, alle anderen Statusinformationen werden mit dem übergeordneten Prozess geteilt.
Um die Abarbeitung von Threads zu verteilen existieren drei Modelle:

1. `1:1` Threads werden direkt auf verteilbare Entitäten (Prozesse) des Betriebssystems abgebildet, vom Betriebssystem also wie vollwertige Prozesse verteilt

2. `N:M` Threads werden auf `M` *virtuellen Prozessoren* ausgeführt. Sowohl das Betriebssystem, als auch die Thread-Software-Bibliothek übernehmen einen Teil der Steuerung, welcher Thread auf welchem Prozessor ausgeführt wird.

3. `N:1` Threads laufen komplett innerhalb eines Betriebssystem-Prozesses ab, weswegen auch nur ein Prozessor gleichzeitig genutzt werden kann. Es handelt sich hierbei also nicht um „echte“ Parallelität.

1.1 Pthreads

Eine häufig anzutreffende Form von Threads ist die Pthreads-API. Der Name steht für POSIX Threads, die API ist seit 1995 in IEEE Std 1003.1c standardisiert, die letzten Änderungen fanden im Jahr 2008 statt. Der Standard definiert eine C-Schnittstelle zur Verwendung von Threads, und trifft keine Aussagen über die Implementierung.
Als generelles Leitbild orientiert sich Pthreads an der fork/join-Parallelität wie von [Qui04] beschrieben.

Obwohl im Rahmen von POSIX (Portable Operating System Interface for Unix) entwickelt existieren auch reine Software-Implementierungen [Eng06] und Implementierungen für Betriebssysteme, die nicht mit Unix kompatibel sind. Am bekanntesten ist hierbei pthreads-w32 für Windows [Joh09].

Als Thread-Verteilungsmodell wird je nach Implementierung eines der drei Modelle genutzt, wobei die `N:1`-Verteilung nur in Software-Implementierungen ohne Multicore-Nutzung verwendet wird. In letzter Zeit zeichnet sich eine

Tendenz zur Nutzung des `1:1`-Modells ab, doch existieren zum Beispiel unter FreeBSD noch aktuelle Implementierungen mit dem `N:M`-Modell.

1.2 Speichermodell

Das Pthreads zu Grunde liegende Speichermodell ist shared memory (gemeinsam genutzter Speicher), das heißt alle Threads können über den gesamten Adressraum verfügen.

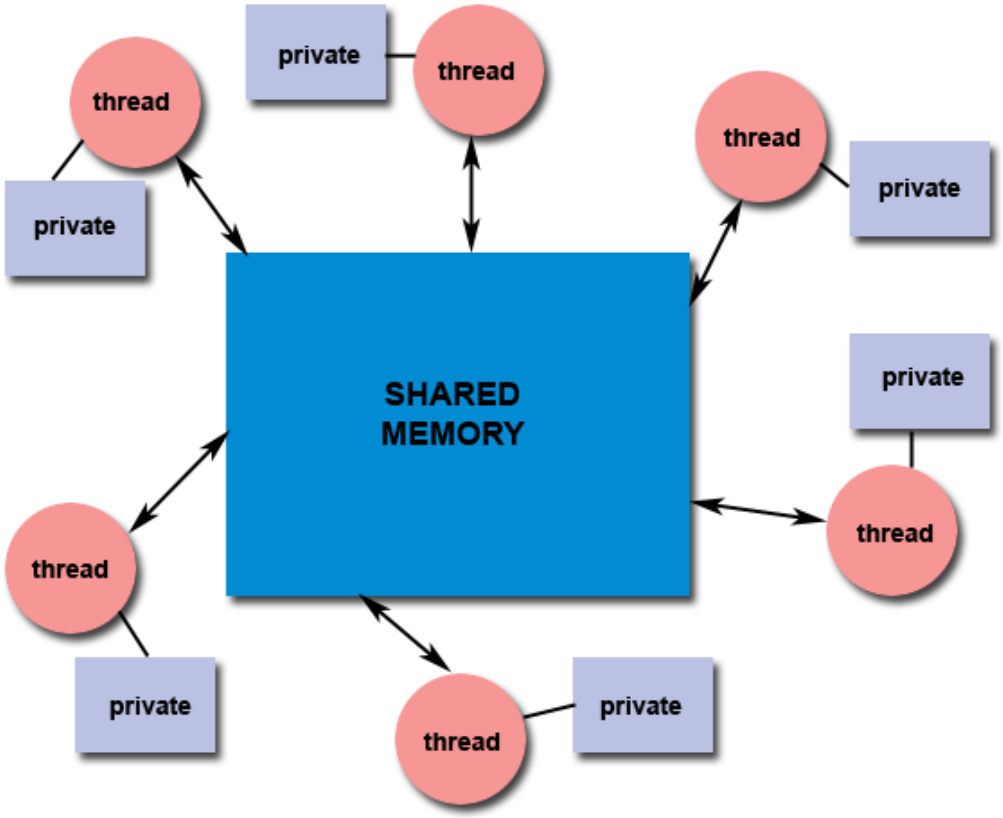

Abbildung 1: Speichermodell (nach [Bar09])

Als privater Speicher steht bei Pthreads der eigene Stack zur Verfügung, im gemeinsam genutzten Speicher kann jeder Thread alles lesen und schreiben, sofern er die Speicheradresse der Daten kennt. Zusätzlich stellt pthreads mit *keys* ein Konzept bereit, um Thread-spezifische Daten (beziehungsweise Zeiger darauf) außerhalb des Thread-Stacks abzulegen. Es liegt in der Verantwortung des Programmierers dafür zu sorgen, dass keine inkonsistenten

Zustände durch Schreiboperationen auf gemeinsam genutzte Daten entstehen.

2 Pthreads - API

Im Folgenden soll die Funktionsweise der Funktionen der API von Pthreads erläutert werden.[1]

2.1 Thread-Management

Zunächst werden die Funktionen zum Erstellen, Beenden und Verwalten von Threads vorgestellt.[2]

2.1.1 Thread-Erstellung

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

Diese Funktion erzeugt einen neuen Thread, welcher start_routine ausführt. Der erste Parameter thread ist dabei ein Zeiger auf ein Objekt vom Typ pthread_t, welches den neuen Thread repräsentiert. Für jeden neu erstellten Thread ist dabei genau ein solcher pthread_t-Repräsentant notwendig.
Mit dem zweiten Parameter attr können spezielle Attribute für den neuen Thread, wie z.B. die Stackgröße gesetzt werden. Wird NULL übergeben, so werden die implementationsabhängigen Standardattribute gesetzt.
Als dritten Parameter erwartet pthread_create einen Zeiger start_routine auf eine Funktion mit Rückgabetyp **void*** und einem Parameter mit Typ **void***. Der letzte Parameter ist ein Zeiger vom Typ **void***, welcher an die aufgerufene Funktion übergeben wird.

Beispiel:

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

void* do_work(void* param) {
```

[1] Pthreads-Datentypen werden in blau dargestellt
[2] Funktionen für das Thread-Management werden in grün dargestellt

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

int main (int argc, char* argv[]) {
  pthread_t thread;
  int rc = pthread_create(&thread, NULL, do_work, NULL);
  if (rc){
    printf("ERROR; return code from pthread_create() is
       %d\n", rc);
    exit(-1);
  }
  pthread_exit(NULL);
}
```

2.1.2 Thread-Ende

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

Die Funktion pthread_exit wird benötigt, um einen Rückgabewert von Threads an den Rest des Programms zurückzugeben, der zurückgegebene Wert wird als Zeiger vom Typ **void*** mit dem pthread_t-Repräsentanten des Threads assoziiert.
Der Aufruf von pthread_exit ist optional, falls kein Wert zurückgegeben werden soll. Der Thread endet in diesem Fall mit der vollständigen abarbeitung der aufgerufenen Funktion. Lediglich im Falle von main() muss pthread_exit aufgerufen werden, falls auf eventuell noch nicht fertig abgearbeitete Threads gewartet werden soll, wie im obigen Beispiel von pthread_create.

Beispiel:

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

2.1.3 Thread-Synchronisation: Join

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

Mit dem Aufruf von pthread_join kann zwischen Threads synchronisiert werden: Der Aufruf blockiert, bis der Thread beendet ist, welcher durch thread repräsentiert wird.
In der übergebenen Adresse value_ptr wird der Zeiger gespeichert, welche pthread_exit im entsprechenden Thread übergeben wurde.

Beispiel:

```
  pthread_t thread;
  int rc = pthread_create(&thread, NULL, do_work,NULL);
  if (rc){
    printf("ERROR; return code from pthread_create() is
       %d\n", rc);
    exit(-1);
  }
  void* a;
  pthread_join(thread, &a);
  int i = a;
  printf("computed %d\n", i);
}
```

2.2 Mutex

Ein wechselseitiger Ausschluss, häufig auch *Mutex* (Verkürzung von *mutual exclusion*) ist ein Verfahren um kritische Abschnitte so zu serialisieren, dass nebenläufige Zugriffe auf gemeinsam genutzte Speicherbereiche keinen inkonsistenten Zustand erzeugen können.
Als klassisches Beispiel dient hier das lost-update-Problem, allgemeiner spricht man von *Race Conditions* (Wettlaufsituationen), da das Ergebnis des Programmablaufs von einem ,,Wettlauf" zwischen zwei oder mehr Threads abhängt.

Zur Veranschaulichung betrachte man den folgenden Code:

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

#define NUM_THREADS 10
```

```
int global;

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

int main (int argc, char* argv[]) {
  global = 0;
  pthread_t thread[NUM_THREADS];
  int t, rc;
  for(t=0; t<NUM_THREADS; t++){
    rc = pthread_create(&thread[t], NULL, add50000, NULL);
    if (rc){
      printf("ERROR; return code from pthread_create() is
         %d\n", rc);
      exit(-1);
    }
  }
  for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], NULL);
    if (rc) {
      printf("ERROR; return code from pthread_join() is
         %d\n", rc);
      exit(-1);
    }
  }

  printf("global is %d\n", global);
}
```

Da die Threads unabhängig voneinander die Variable global auslesen, um eins erhöhen, und wieder zurückschreiben, kann hier der Fall auftreten, dass mehrere Threads denselben Wert auslesen, und somit Inkrementierungen verloren gehen.
Um dieses Problem zu umgehen ist ein wechselseitiger Ausschluss erforderlich.[3]

[3]Die zum wechselseitigen Ausschluss gehörenden Funktionen und Makros werden in rot dargestellt

2.2.1 Mutex-Initialisierung

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Zur Initialisierung mit den Standardeinstellungen eines Mutex stellt Pthreads das Makro PTHREAD_MUTEX_INITIALIZER bereit. Dieses ist gegenüber einem Aufruf von pthread_mutex_init zu bevorzugen, da für die Standard-Parameter eine statische Initialisierung möglich ist.

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

Falls erforderlich kann mit pthread_mutex_init ein Mutex initialisiert werden, der über mehrere Prozesse geteilt wird, oder dessen Prioritätsverwaltung nicht der Standardeinstellung entsprechen soll. Es sollte beachtet werden, dass nicht alle Pthreads-Implementierungen alle Attribute zur Verfügung stellen.

2.2.2 Mutex-Verwendung

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

Diese beiden Funktionen dienen dem Sperren, bzw. Freigeben eines Mutex. Ein Thread, der in eine kritische Sektion gelangt kann mit pthread_mutex_lock sicherstellen, dass kein anderer Thread dieselbe Sperre hält, der Thread hat somit exklusiven Zugriff auf die vom Mutex geschützten Resourcen.

Um den Zugriff auf den Mutex, und damit die Resourcen, wieder freizugeben muss pthread_mutex_unlock aufgerufen werden.

Pthreads selbst stellt keine Konstrukte zur Vefügung, die die Verwendung einer Resource an einen Mutex koppeln, es liegt somit in der Verantwortung des Programmierers sicherzustellen, dass gemeinsam genutzte Resourcen durch wechselseitigen Ausschluss geschützt werden.

Folgender Code behebt das zu Beginn gezeigte Problem beim Zugriff auf eine globale Variable:

```
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);
}
```

2.3 Conditions

Häufig ist es nicht ausreichend einen wechselseitigen Ausschluss für Resourcen zu definieren, da eine bestimmte *Bedingung* erfüllt sein muss, damit ein Thread seine Arbeit fortsetzen kann.
Für die regelmäßige Überprüfung dieser Bedingung wäre es jeweils notwendig alle anderen Threads anzuhalten, die mit der Resource arbeiten, was die Laufzeit des Programms erheblich verlängern kann.

Um den wartenden Thread zielgerichtet zu informieren, dass er seine Arbeit fortsetzen kann, stellt Pthreads *Conditions* (Bedingungen) bereit.[4]

2.3.1 Condition-Initialisierung

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

Ähnlich wie beim wechselseitigen Ausschluss kann eine Condition mit Hilfe des Makros PTHREAD_COND_INITIALIZER initialisiert werden.
Die Funktion pthread_cond_init ist normalerweise nicht notwendig, da nur eine einzige Eigenschaft zusätzlich gesetzt werden kann: Die Möglichkeit die Condition über mehrere Prozesse zu teilen. Diese Funktionalität wird nicht von allen Pthreads-Implementierungen unterstützt, und sollte nur verwendet werden, wenn dies unbedingt notwendig ist.

2.3.2 Condition-Verwendung

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

[4]Die zu Conditions gehörenden Funktionen und Makros werden in orange dargestellt

Mit dem Aufruf von pthread_cond_wait wird der aufrufende Thread schlafen gelegt bis ein Signal für diese Condition ausgelöst wird.
Dabei wird der Mutex mutex freigegeben, sodass andere Threads auf die von ihm geschützten Resourcen zugreifen können. Sobald die Condition ein Signal erhält, versucht der Thread den Mutex erneut zu sperren, sobald dies gelungen ist wird der Thread fortgesetzt. Dies bedeutet insbesondere, dass vor und nach dem Aufruf von pthread_cond_wait der übergebene Mutex gesperrt ist, ein Aufruf mit ungesperrtem Mutex ist ein Programmierfehler mit nicht-definiertem Verhalten.

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

Die beiden Funktionen pthread_cond_signal und pthread_cond_broadcast lösen Signale auf der übergebenen Condition cond aus.
Dabei wird pthread_cond_signal verwendet um einen wartenden Thread zu wecken, pthread_cond_broadcast hingegen, wenn mehrere Threads auf die Condition warten.

Falls mehrere Threads auf die Condition warten und pthread_cond_signal verwendet wird, so ist lediglich sichergestellt, dass mindestens einer der Threads aufgeweckt wird, der Pthreads-Standard erlaubt hier aber ausdrücklich sogenannete *spurious wakeups*, d.h. es können mehrere Threads aufgeweckt werden, weshalb diese nach dem Aufwachen und Sperren des Mutex dennoch das mit der Condition signalisierte Prädikat erneut prüfen müssen.

Der folgende Programmcode demonstriert die Verwendung von Conditions:

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

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

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);
  }
}

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);
  }
}


int main (int argc, char* argv[]) {
  global = 0;
  pthread_t thread[2];
  int t, rc;
  rc = pthread_create(&thread[0], NULL, add50000, NULL);
  if (rc){
    printf("ERROR; return code from pthread_create() is
       %d\n", rc);
    exit(-1);
  }
  rc= pthread_create(&thread[1], NULL, print_global, NULL);
  if (rc){
    printf("ERROR; return code from pthread_create() is
       %d\n", rc);
    exit(-1);
  }
  for(t=0; t<2; t++) {
    rc = pthread_join(thread[t], NULL);
    if (rc) {
      printf("ERROR; return code from pthread_join() is
         %d\n", rc);
      exit(-1);
    }
  }
}
```

2.4 Thread-Abbruch

Zum Beispiel bei spekulativer Ausführung eines Threads kann es sinnvoll sein diesen vorzeitig abzubrechen. Jedoch muss dabei darauf geachtet werden, dass kein inkonsistenter Programmzustand entsteht.[5]

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

Um einen Thread abzubrechen muss sein Repräsentant an pthread_cancel übergeben werden.

2.4.1 Abbruch-Arten

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

Neu erstellte Threads können standardmäßig abgebrochen werden.
Um zu verhindern, dass ein Thread abgebrochen werden kann setzt man mit pthread_setcancelstate den Zustand auf PTHREAD_CANCEL_DISABLE, um den Ursprungszustand wiederherzustellen auf PTHREAD_CANCEL_ENABLE.
Pthreads kennt zwei unterschiedliche Arten mit Thread-Abbruch umzugehen, was mit pthread_setcanceltype eingestellt werden kann. Initial werden Threads verzögert abgebrochen. Konkret heißt dies, dass ein Thread bis zum nächsten *cancellation point* (Abbruchspunkt) weiterläuft, und erst dort abgebrochen wird. Eine Liste der Abbruchspunkte findet sich in der man-page zu pthreads [man08].
Alternativ können Threads sofort abgebrochen werden, dies bedeutet jedoch einen erhöhten Aufwand bei der Sicherstellung eines konsistenten Zustandes. Um einen Thread ohne Verzögerung abbrechen zu können setzt man mit pthread_setcanceltype seinen Typ auf PTHREAD_CANCEL_ASYNCHRONOUS, für verzögertes Abbrechen stellt man ihn auf PTHREAD_CANCEL_DEFERRED.

2.4.2 Aufräumen

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

Um im Falle eines Thread-Abbruchs notwendige Aufräumarbeiten durchzuführen müssen entsprechende Aufräum-Routinen mit pthread_cleanup_push auf den Aufräum-Stack geschrieben werden.

[5] Funktionen und Makros des Thread-Abbruchs werden wie Thread-Management-Funktionen in grün dargestellt

Beim Abbruch wird dann arg als Parameter an routine übergeben.
Wird der Thread nicht abgebrochen, so kann mit pthread_cleanup_pop die neueste Aufräum-Routine entfernt werden. Falls execute dabei ungleich 0 ist, so wird die Routine ausgeführt.

3 Probleme

3.1 Implementierungsvielfalt

Durch die enorme Menge von Implementierungen ist es schwierig portable Programme zu schreiben, die auf allen Implementierungen gut laufen. Um sicher zu gehen muss auf viele Features verzichtet werden, und die Stack-Größe sollte manuell auf ausreichende Werte für jeden Thread gesetzt werden [Bar09].

Pthreads bietet zwar Möglichkeiten Threads mit unterschiedlichen Prioritäten und Scheduling-Verfahren ablaufen zu lassen, doch die Implementierungen müssen dies nicht zwingend unterstützen.
Zum Beispiel unterstützt NPTL Thread-Prioritäten nicht in den Standardeinstellungen. Setzt man jedoch einen anderen Scheduler ein, so entstehen Echtzeit-Threads, welche mit Prioritäten 0-99 versehen werden können. Es sind root-Rechte für den Wechsel zu Echtzeit-Threads erforderlich. Für weitere Details zu Prioritäten und damit verbundenen Implementierungs-Unterschieden siehe [Mai06]
Generell sind zudem die durch die init-Funktionen bei Mutex und Condition setzbaren Attribute nicht in jeder Implementierung vorhanden, was ihre Verwendbarkeit für portable Programme stark einschränkt.

Zu diesen allgemeinen Interoperabilitätsproblemen kommen subtile Unterschiede durch die eingesetzten Scheduling-Modelle. So können offensichtlich bei GNU portable Threads keine echten Wettlaufsituationen auftreten, da diese Implementierung keine Hardware-Parallelität nutzt.
Aber auch die anderen beiden Modelle haben kleinere Unterschiede, weswegen FreeBSD in Version 7 sowohl das `N:M`-Modell zur Abwärtskompatibilität anbietet, als auch eine neue Thread-Implementierung mit `1:1`-Modell unterstützt.

3.2 Prioritätsinversion

Prioritätsinversion bezeichnet das Problem, dass Threads mit niedriger Priorität abgearbeitet werden, obwohl ein Thread mit höherer Priorität wartet. Dieser Effekt kann bei Pthreads auftreten, wenn ein Thread mit niedriger Priorität einen Mutex sperrt, auf den anschließend ein Thread hoher Priorität wartet.
Werden nun gleichzeitig viele Threads mittlerer Priorität ausgeführt, so bekommt der niedrig priorisierte Thread keine Rechenzeit mehr zugeteilt, und kann somit den Mutex auch nicht mehr freigeben. Der Thread mit hoher Priorität muss also auf das Ende der Threads mittlerer Priorität warten.

Um dieses Problem zu umgehen bietet Pthreads die Möglichkeit Threads vorübergehend eine höhere Priorität zuzuweisen, während sie einen Mutex gesperrt halten. Dies verhindert das Auftreten von Prioritätsinversion wie oben beschrieben, solange kein Thread mit höherer Priorität als der vorübergehend gewährten auf den Mutex wartet. Leider gilt auch hier, dass diese Funktionalität nicht von allen Implementierungen zur Verfügung gestellt wird.

4 Alternativen zu Pthreads

Solaris Threads
Solaris Threads ähneln Pthreads vom Aufbau her stark, die Funktionen haben allerdings andere Namen [Sun08]. Einige wenige Funktionen stehen in Pthreads nicht zur Verfügung, allerdings können Solaris Threads und Pthreads kooperativ im selben Programm verwendet werden, wodurch die Vorteile beider Schnittstellen genutzt werden können. Solaris Threads laufen, wie der Name bereits andeutet, nur auf Solaris-Betriebssystemen.

Boost.Thread
Für das Programmieren in C++ bietet es sich an eine abstraktere Thread-Bibliothek als Pthreads zu verwenden.
Boost.Thread ist eine frei verfügbare Thread-Bibliothek für C++, die je nach Betriebssystem eine passende Thread-Implementierung verwendet. Unter Linux und weiteren Betriebssystemen wird Pthreads als Grundlage verwendet, es werden aber nicht alle Pthreads-Funktionen zur Verfügung gestellt. So ist das Abbrechen von Threads in C++ prinzipiell nicht zu empfehlen, da es den Aufruf von Destruktoren unterdrücken kann, und somit Speicherlecks auslösen kann.

Für den Programmierer bieter Boost.Thread gegenüber Pthreads einfachere Strukturen, wie zum Beispiel scoped_lock, welches einen Mutex bei seiner Erstellung sperrt, und beim Verlassen des Sichtbarkeitsbereiches wieder freigibt. Außerdem können Threads zu Gruppen zusammengefasst werden, was die Verwaltung vereinfachen kann.

OpenMP
Bei vielen bestehenden Programmen kann eine Geschwindigkeitssteigerung durch Parallelisierung von Schleifen erreicht werden. Hierfür ist OpenMP besser geeignet als Pthreads, da es dem Programmierer die Arbeit des Thread-Erstellens abnimmt, und erweiterte Funktionen für parallele Schleifen bietet. Die automatische Erstellung von Threads bietet den Vorteil auch bei vielen Kernen eine Geschwindigkeitssteigerung zu erziehlen, ohne wie in Pthreads selbst mehr Ausführungsstränge anlegen zu müssen.
Allerdings ist OpenMP weniger flexibel als Pthreads. Es eignet sich nicht um Threads zu erzeugen, welche unterschiedlichen Aufgaben innerhalb des Programms übernehmen.

MPI
Da Pthreads einen gemeinsam genutzten Speicher voraussetzt ist es nicht beliebig skalierbar. Auf NUMA-Systemen muss der Programmierer beim Verwenden von Pthreads beachten, dass der Speicher desjenigen Prozessors belegt wird, auf dem der speicherbelegende Thread gerade läuft. Dies führt häufig zu Engpässen bei der Datenübertragung zwischen den Prozessoren.
Um auch verteilten Speicher nutzen zu können ist im Moment MPI das Mittel der Wahl. Es erlaubt nicht nur die effiziente Nutzung des gemeinsamen Speichers auf NUMA-Systemen, sondern kann auch für Cluster-Systeme eingesetzt werden.
Diese höhere Flexibilität und Skalierbarkeit sollte allerdings gegen die erheblich schwierigere Benutzung abgewogen werden.

Literatur

[Bar09] Blaise Barney. POSIX Threads Programming. Website, 2009. Available online at `https://computing.llnl.gov/tutorials/pthreads/`.

[Eng06] Ralf S. Engelschall. GNU Pth - The GNU Portable Threads. Website, 2006. Available online at `http://www.gnu.org/software/pth/`.

[Joh09] Ross Johnson. Open Source POSIX Threads for Win32. Website, 2009. Available online at `http://sourceware.org/pthreads-win32/`.

[Mai06] Gregor Maier. Thread Scheduling with pthreads under Linux and FreeBSD. Website, 2006. Available online at `http://www.net.t-labs.tu-berlin.de/~gregor/tools/pthread-scheduling.html`.

[man08] Linux man-pages project. *pthreads - POSIX threads*, 2008. Available online at `http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html`.

[Qui04] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*, pages 405–406. McGraw-Hill, 2004.

[Sun08] Multithreaded Programming Guide. Website, 2008. Pages 174–177, content available online at `http://docs.sun.com/app/docs/doc/816-5137/sthreads-96692?a=view`.