2 Threads

Threads sind die Grundbausteine paralleler Software und bilden – wie in Abschnitt 1.4 bereits erwähnt – aus Sicht des Betriebssystems die kleinste Einheit der Parallelität [55]. Im Prinzip werden Threads auf Multicore-Prozessoren genauso benutzt wie auf Systemen mit nur einem Prozessorkern. Dennoch lohnt sich auch für Entwickler, denen Threads wohlbekannt sind, ein Blick auf einige Aspekte, die erst auf Multicore-Prozessoren zum Tragen kommen. In diesem Kapitel gehen wir auf die wichtigsten Aspekte ein und zeigen typische Probleme beim Umstieg auf parallele Systeme auf. Dabei interessiert uns, wie das Betriebssystem die Threads auf die Kerne verteilt, was sich an ihrem Verhalten gegenüber der Ausführung auf Single-Core-Systemen ändert und welche Rolle Speicherzugriffe bei der Programmierung mit Threads spielen.

2.1 Arbeiten mit Threads

Das Konzept von Threads reicht in die Anfangszeit der Betriebssystementwicklung zurück. Heutzutage unterstützen nahezu alle Betriebssysteme Threads, was insbesondere für grafische Benutzeroberflächen unerlässlich ist. Während Threads in der Vergangenheit vor allem dafür eingesetzt wurden, voneinander unabhängige Abläufe zu entkoppeln, ist auf Multicore-Systemen der Aspekt der Geschwindigkeitssteigerung durch Parallelisierung hinzugekommen. Bevor wir näher auf die Parallelisierung mit Threads eingehen, widmen wir uns jedoch den Grundkonzepten der threadbasierten Programmierung.

2.1.1 Erzeugung und Beendigung

Es gibt zwei Grundoperationen, die für das Arbeiten mit Threads von Bedeutung sind: die Erzeugung eines neuen Threads und das Warten auf die Beendigung eines Threads. Listing 2.1 zeigt ein einfaches Beispiel. Aus dem Hauptprogramm wird durch Aufruf von `createThread` ein neuer Thread gestartet. Die zu erledigende Arbeit besteht in diesem Fall aus der Funktion `doWork`. Am Ende von `main` wird mittels `join` gewartet, bis der Thread seine Arbeit vollendet hat. Falls der Thread schon vor dem Aufruf von `join` fertig ist, kann in `main` sofort mit eventuellen Aufräumarbeiten fortgefahren werden.

```
void main() {
  // ... Initialisierung
  // starte neuen Thread
  Thread t = createThread(lambda () {doWork();});
  // ... Hauptprogramm
  // warte auf Thread
  t.join();
  // ... Aufräumarbeiten
}

doWork() {
  // ... Berechnungen des Threads
}
```

Listing 2.1 *Arbeiten mit Threads*

Je nach Programmiersprache bzw. Bibliothek gibt es verschiedene Möglichkeiten, bei der Erzeugung eines Threads den auszuführenden Programmteil anzugeben. In Sprachen wie C wird dazu ein Funktionszeiger übergeben, in objektorientierten Sprachen typischerweise eine Methode oder eine Instanz einer ausführbaren Klasse. Mit »ausführbar« ist hier gemeint, dass die Klasse über eine bestimmte Methode verfügt (z. B. `run`), die von dem neu erzeugten Thread aufgerufen wird. Sofern die Sprache Lambda-Ausdrücke und Closures unterstützt (siehe Kasten auf S. 25), können auch diese einem Thread übergeben werden, was insbesondere die Parameterübergabe erleichtert. In unserem Pseudocode verwenden wir eine Lambda-Notation.

2.1.2 Datenaustausch

Da alle Threads eines Prozesses Zugriff auf denselben Speicherbereich haben, können sie über gemeinsame Variablen bzw. Objekte Daten austauschen (engl. *shared memory programming*). Dabei ist jedoch in mehrerlei Hinsicht Vorsicht geboten. Zum einen muss sichergestellt werden, dass die Threads immer die aktuellen Daten »sehen«. Das ist selbst auf Single-Core-Systemen nicht zwangsläufig der Fall, da Variablen nach Möglichkeit in Prozessorregistern gehalten werden und nicht jede Änderung sofort in den Hauptspeicher zurückgeschrieben wird. Zum anderen können beim Zugriff auf gemeinsame Daten Konflikte auftreten, die Inkonsistenzen zur Folge haben. Auf die Lösung dieser Probleme werden wir in Abschnitt 2.3 und in Kapitel 3 detailliert eingehen.

Fehler beim Datenaustausch sind oft nur schwer zu erkennen, da das tatsächliche Verhalten von verschiedenen Faktoren wie dem Scheduling (siehe Abschnitt 2.2) und den Optimierungen durch den Compiler abhängt. So werden beispielsweise bei deaktivierter Optimierung die Registerinhalte meist sofort in den Speicher zurückgeschrieben, weshalb das Problem der Sichtbarkeit oft nur

Exkurs

Bei der parallelen Programmierung müssen wir häufig Funktionen als Argumente an andere Funktionen übergeben. Beispiele dafür sind die Threaderzeugung (wir übergeben die Einsprungfunktion) und die Übergabe von Rückruffunktionen (engl. *call back*). In C werden dazu Funktionszeiger übergeben, objektorientierte Sprachen arbeiten üblicherweise mit Funktionsobjekten (Funktoren). Ein Funktionsobjekt repräsentiert eine Funktion, kann aber wie jedes andere Objekt in einer Variable gespeichert oder einer Funktion übergeben werden. Im Pseudocode verwenden wir für Funktionsobjekte den generischen Typ:

```
Function<Parameter1, Parameter2, ..., Return>
```

Die Typparameter definieren die Typen der Argumente und den Typ des Rückgabewerts der Funktion. Funktionsobjekte werden je nach Programmiersprache auf verschiedene Weise definiert. Zunehmend setzen sich sogenannte *Lambda-Funktionen* durch. Wir verwenden folgende Syntax:

```
lambda (Argumente) Anweisungsblock
```

Das Schlüsselwort `lambda` kennzeichnet eine Lambda-Funktion, es folgt eine Liste von Argumenten und ein Anweisungsblock. Damit lässt sich z. B. folgende Additionsfunktion definieren:

```
Function<int, int, int> add =
  lambda (int a, int b) {return a + b;};
```

Lambda-Funktionen, die einer anderen Funktion als Argument übergeben werden, lassen sich direkt innerhalb der Argumentliste definieren:

```
// Aufruf mit Lambda-Funktion
int result = calculate(47, 11,
  lambda (int a, int b) {return a + b;});

int calculate(int x, int y, Function<int, int, int> f) {
  return f(x, y); // Aufruf der durch das Funktionsobjekt
}                 // repräsentierten Funktion
```

Darüber hinaus unterstützen Lambda-Funktionen das Konzept von *Closures*. Damit ist gemeint, dass freie Variablen der Funktionsdefinition über den lexikalischen Kontext gebunden werden. Betrachten wir dazu folgendes Beispiel:

```
int offset = 42;
int result = calculate(47, 11,
  lambda (int a, int b) {return a + b + offset;});
```

Die Variablen `a` und `b` sind durch die Funktionsdefinition gebunden, `offset` jedoch nicht: Diese Variable existiert nur im Kontext des Aufrufers. Der Compiler bindet nun diese Variable an die Lambda-Funktion. In unserem Beispiel wird der Wert von `offset` in das Funktionsobjekt kopiert, damit er auch nach dem Verlassen des Aufrufkontextes zur Verfügung steht. Dadurch kann die Lambda-Funktion an einen anderen Thread übergeben und dort ausgeführt werden.

bei aktivierter Optimierung zutage tritt. Auf Single-Core-Systemen kommen zudem viele Konflikte aufgrund der verschränkten Ausführung nur sehr selten zum Vorschein.

Hinweis

Aktivieren Sie Compileroptimierungen für den Test und führen Sie die Tests auf Multicore-Systemen aus, um Fehler beim Datenaustausch erkennen zu können.

Wenn es nur darum geht, einem Thread zu Beginn Daten zur Verfügung zu stellen, gibt es meist einen einfacheren Weg: Bei den gängigen Programmiersprachen bzw. Bibliotheken kann man einem Thread bei dessen Erzeugung ein oder mehrere Argumente übergeben. Damit erübrigt sich der Datenaustausch über gemeinsame Variablen. Darüber hinaus kann man zum Teil auch Attribute angeben, beispielsweise um die Threadpriorität zu setzen. Die Art und Weise der Übergabe von Argumenten bzw. Attributen hängt jedoch stark von der eingesetzten Sprache bzw. Bibliothek ab. Details dazu finden sich im zweiten Teil dieses Buches.

Kommen wir noch einmal auf das Programmiermodell des gemeinsamen Speichers zurück. Eine Variable ist von mehreren Threads aus zugreifbar, wenn sie global deklariert ist oder auf dem Heap liegt. Da lokale Variablen immer auf dem Stack liegen, sind sie nur für einen Thread sichtbar, da jeder Thread über einen eigenen Stack verfügt. Voraussetzung ist jedoch, dass keine Referenzen oder Zeiger auf Objekte, die auf dem Stack liegen, zwischen den Threads ausgetauscht werden. Die in Listing 2.2 gezeigte Funktion, die die Anzahl der Vorkommen eines Zeichens in einem String bestimmt, kann man bedenkenlos aus mehreren Threads gleichzeitig aufrufen, da die Variablen `n` und `i` lokal sind. Allerdings muss sichergestellt sein, dass der String nicht durch einen anderen Thread verändert wird. In C++ ist das für Argumente, die als Wert übergeben werden (engl. *call by value*), automatisch sichergestellt. In Sprachen wie Java und C# werden Strings jedoch als Referenz übergeben. Falls der übergebene String parallel durch einen anderen Thread verändert wird, muss deshalb eine Kopie angelegt werden.

```
int count(string s, char c) {
  int n = 0;
  for(int i = 0; i < s.length(); i++) {
    if(s.get(i) == c) {
      n++;
    }
  }
  return n;
}
```

***Listing 2.2** Funktion zum Zählen der Vorkommen eines Zeichens in einem String*

2.1.3 Threadpools

Bei der Parallelisierung hat man es oftmals mit zahlreichen, relativ kleinen Aufgaben (engl. *tasks*) zu tun. Beispielsweise lassen sich viele Algorithmen aus der Bildverarbeitung parallelisieren, indem man das Ausgangsbild in Teile zerlegt und diese parallel verarbeitet. Für jede Teilaufgabe eines Problems einen neuen Thread zu erzeugen, kostet jedoch unnötig viel Zeit, da die zur Threaderzeugung notwendigen Betriebssystemaufrufe teuer sind. Deshalb liegt es nahe, Threads wiederzuverwenden. Zu diesem Zweck gibt es Threadpools. Ein Threadpool besteht aus einer Menge von Threads, die einmal zu Beginn erzeugt werden und so lange die anfallenden Aufgaben abarbeiten, bis der Threadpool beendet wird.

Die Aufgaben werden üblicherweise als Taskobjekte in eine Warteschlange (engl. *queue*) eingereiht. Ein Taskobjekt besteht mindestens aus einem Zeiger auf die auszuführende Funktion, ergänzt um eventuelle Argumente, die dieser Funktion übergeben werden. Sofern die Programmiersprache es unterstützt, kann dem Threadpool auch eine Lambda-Funktion zur Ausführung übergeben werden. Die Threads des Pools entnehmen die Taskobjekte der Warteschlange und führen die dazugehörigen Funktionen aus. Abbildung 2.1 illustriert die grundlegende Funktionsweise eines Threadpools.

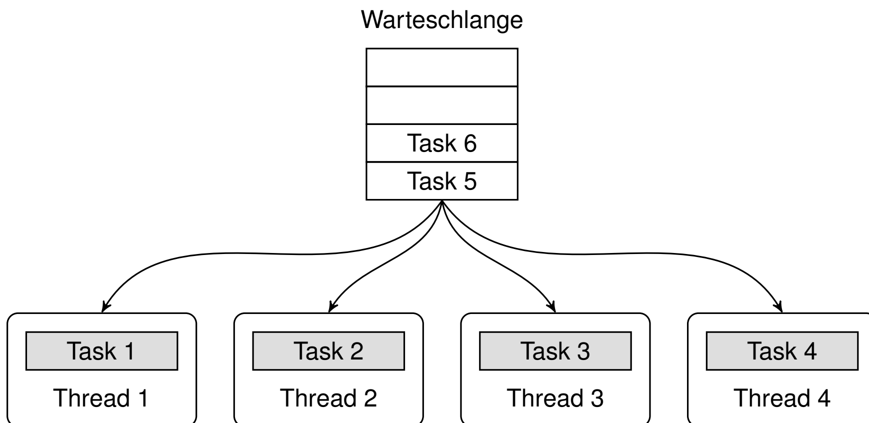

Abbildung 2.1 *Threadpool*

Der Einsatz von Threadpools ist immer dann sinnvoll, wenn über die Laufzeit eines Programms viele unabhängige Aufgaben zu bearbeiten sind. Dies ist zum Beispiel bei Webservern, die fortwährend Anfragen bearbeiten müssen, der Fall. Listing 2.3 skizziert die Implementierung eines Webservers mithilfe eines Threadpools. Um mehrere Clients gleichzeitig bedienen zu können, wird für jede Anfrage dem Threadpool ein Task übergeben, der die Funktion `handleClient` aufruft. Glücklicherweise muss man sich in der Regel nicht um die Implementierung des

Threadpools kümmern. Java und C# bringen Threadpools von Haus aus mit, und auch für C/C++ sind fertige Implementierungen verfügbar.

```
ServerSocket serverSocket;
ThreadPool threadPool;
// ... Initialisierung
while(true) {
  ClientSocket clientSocket = serverSocket.accept();
  threadPool.execute(lambda () {handleClient(clientSocket);});
}
```

Listing 2.3 Implementierung eines Webservers mithilfe eines Threadpools

Ein Vorteil von Threadpools gegenüber der manuellen Verwendung von Threads ist, dass man die Anzahl der zu startenden Threads an zentraler Stelle konfigurieren kann. Dies kann automatisch erfolgen, indem man die Anzahl der verfügbaren Prozessorkerne beim Programmstart abfragt und die Anzahl der Threads abhängig von der Systemauslastung anpasst. Auf diese Weise erreicht man ein hohes Maß an Portabilität bei optimaler Auslastung der vorhandenen Hardware. Allerdings wird die zentrale Warteschlange bei vielen Threads und kleinen Aufgaben schnell zu einem Flaschenhals. Wir kommen beim Thema Taskparallelität in Abschnitt 4.1 darauf zurück.

2.2 Scheduling

Die Zuteilung von Rechenzeit an die Threads übernimmt der *Scheduler* des Betriebssystems. Auf einem Single-Core-System werden Threads meistens verschränkt ausgeführt, also abwechselnd dem Prozessor zugeteilt (engl. *interleaving*). Es handelt sich in diesem Fall also nur um eine logische Form der Parallelität. Der Scheduler arbeitet auf Single-Core-Systemen in der Regel nach dem Zeitscheibenverfahren (engl. *time slicing*). Das heißt, dass er den laufenden Thread nach Ablauf seiner Zeitscheibe unterbricht und auf einen anderen Thread umschaltet. Auf diese Weise werden alle Threads der Reihe nach bedient (engl. *round robin*). Die Auswahl des nächsten Threads geschieht anhand der Threadpriorität, die vom Programmierer gesetzt wurde. Im Detail gibt es hier Unterschiede zwischen den verschiedenen Betriebssystemen, die wir hier jedoch nicht näher betrachten. Ebenso existieren neben prioritätsgesteuerten Zeitscheibenverfahren weitere Scheduling-Strategien, die vorwiegend in Echtzeitsystemen zum Einsatz kommen.

Multicore-Systeme hingegen erlauben eine echt parallele Ausführung mehrerer Threads. Jeder Prozessorkern bearbeitet mindestens einen Thread und kann üblicherweise auch mehrere Threads verschränkt ausführen. So ist die Anzahl der Threads nicht durch die Anzahl der Kerne beschränkt.

2.2.1 Lastverteilung

Auf Multicore- und Multiprozessorsystemen versucht das Betriebssystem, die Last optimal auf die verfügbaren Kerne zu verteilen. Dazu verschiebt der Scheduler Threads zwischen den Kernen bzw. Prozessoren. Das Verschieben eines Threads von einem Prozessorkern auf einen anderen kostet jedoch Zeit. Erschwerend kommt hinzu, dass das Betriebssystem nichts über das Speicherzugriffsverhalten der Anwendung weiß. Genau das kann sich negativ auf die Laufzeit auswirken, wenn zwei Threads intensiv an denselben Daten arbeiten, aber auf verschiedenen Kernen oder gar Prozessoren ausgeführt werden. So müssen die Daten bei jeder Änderung vom Cache des einen Kerns in den des anderen transportiert werden, worunter die Geschwindigkeit leidet. Da die kleinste Verwaltungseinheit des Cache-Kohärenzprotokolls eine Cache-Zeile ist, passiert das auch dann, wenn der zweite Thread nur Daten liest, die zufällig in derselben Cache-Zeile liegen. Wir vertiefen das Thema in Abschnitt 2.3.2.

Es gibt verschiedene Strategien, um den Aufwand, der mit dem Verschieben eines Threads einhergeht, zu reduzieren. Zunächst versuchen die Scheduler in der Regel, einen Thread auf dem Kern auszuführen, auf dem er zuvor bereits ausgeführt wurde (engl. *soft affinity*). Sinn und Zweck dieser Vorgehensweise ist, dass sich mit einer relativ hohen Wahrscheinlichkeit noch Daten des Threads in dem Cache des Kerns befinden. Falls der Thread auf einen anderen Kern verschoben werden muss, sollte dies unter Berücksichtigung der Hardwarearchitektur erfolgen. So kennt beispielsweise der Linux-Scheduler die Cache-Hierarchie und versucht, einen »benachbarten« Kern zu finden, der sich mit dem ursprünglich verwendeten Kern einen Cache teilt. Nur wenn es gar nicht anders geht, wird ein Thread auf einen anderen Prozessor verschoben. In diesem Fall müssen die Daten natürlich neu in den entsprechenden Cache geladen werden. Die Strategie der Lastverteilung variiert jedoch zwischen den Betriebssystemen und sogar zwischen verschiedenen Versionen desselben Betriebssystems.

Ob ein Thread verschoben werden muss, hängt unter anderem davon ab, wie viele Threads vorhanden sind. Bei der Parallelisierung stellt sich deshalb schnell die Frage, wie viele Threads gestartet werden sollen. Es liegt nahe, mindestens so viele Threads zu starten, wie Prozessorkerne zur Verfügung stehen. Nur so können alle Kerne auch wirklich genutzt werden. Werden mehr Threads gestartet, als Kerne zur Verfügung stehen, muss der Scheduler wie auf Single-Core-Systemen periodisch zwischen den einzelnen Threads umschalten. Die mit einer solchen Überbelegung (engl. *oversubscription*) verbundenen Kontextwechsel kosten jedoch Zeit und machen einen Teil des Geschwindigkeitsgewinns zunichte. Deshalb sollte man bei der Erzeugung von Threads nicht nach dem Motto »viel hilft viel« verfahren. Im Idealfall führt jeder Kern einen Thread aus. Entscheidend ist allerdings, ob alle Threads die CPU wirklich nutzen. Enthalten sie blockierende Operationen wie das Warten auf Peripheriegeräte, können und sollen wesentlich

mehr Threads gestartet werden, als Kerne vorhanden sind. Ansonsten kommt es zu einer Unterbelegung (engl. *undersubscription*) des Systems.

2.2.2 Affinitäten und Prioritäten

Die meisten Betriebssysteme bieten Mechanismen an, mit denen sich verhindern lässt, dass ein Thread auf einen anderen Prozessorkern verschoben wird. Dazu wird der Thread fest an einen Kern gebunden (engl. *pinning*). In der Regel geschieht dies mithilfe sogenannter Affinitäten, die dem Scheduler vorschreiben, auf welchem Kern ein Thread ausgeführt werden soll. Deshalb spricht man in diesem Zusammenhang auch von harten Affinitäten (engl. *hard affinities*). Affinitäten sind nicht auf einen einzelnen Kern beschränkt. Meist kann man eine Teilmenge von Kernen angeben, die vom Scheduler verwendet werden dürfen, beispielsweise in Form einer Bitmaske. Die Zuordnung eines Threads zu einem oder mehreren Kernen durch den Programmierer schränkt jedoch die Portabilität der Anwendung ein. Sofern man keine Echtzeitanwendungen für genau eine festgeschriebene Hardwarekonfiguration entwickelt, sollte man deshalb darauf verzichten. Schließlich möchte man sich als Anwendungsentwickler normalerweise ja gerade nicht um die zugrunde liegende Hardware kümmern.

Außer durch Affinitäten lässt sich das Scheduling auch durch Prioritäten beeinflussen. Obwohl Prioritäten bekanntlich auch auf Systemen mit nur einem Kern Verwendung finden, gibt es auf Multicore-Systemen einen wichtigen Unterschied: Niederpriore Threads können parallel zu hochprioren laufen. Werden beispielsweise vier Threads unterschiedlicher Priorität auf einem Prozessor mit vier Kernen ausgeführt, können diese durchaus gleichzeitig laufen. Dahingegen wird auf Single-Core-Prozessoren ein Thread nur dann ausgeführt, wenn kein höherpriorer bereit ist. Dieser Unterschied ist vor allem dann von Bedeutung, wenn bei der Entwicklung einer Anwendung davon ausgegangen wurde, dass bestimmte Threads andere mit niedrigerer Priorität verdrängen. Solche Anwendungen führen auf Multicore-Rechnern zu Problemen, obwohl sie auf Single-Core-Rechnern einwandfrei funktionieren.

2.3 Speicherzugriff

In der Literatur über parallele Algorithmen spielt der Zugriff auf gemeinsamen Speicher oft nur eine untergeordnete Rolle. Das liegt nicht zuletzt daran, dass die meisten Anwendungen für Cluster oder massiv parallele Rechner auf einem nachrichtenorientierten Programmiermodell beruhen. Auf speichergekoppelten Systemen ist die richtige Nutzung des gemeinsamen Speichers dagegen ein entscheidender Teil des Programmentwurfs. Wie in Abschnitt 2.1.2 bereits angedeutet, muss beim Datenaustausch zwischen zwei Threads beispielsweise klar sein, wann welche Daten des einen Threads für den anderen sichtbar sind. Das ist ein wesent-

licher Punkt, der durch das Speichermodell definiert wird. Abschnitt 2.3.1 erklärt die Hintergründe, die auch für Single-Core-Architekturen gelten. Auf Multicore-Systemen hat die Speicherarchitektur zudem Einfluss auf die Geschwindigkeit eines Programms. In Abschnitt 2.3.2 beschreiben wir die Gründe dafür und was zu beachten ist, um architekturbedingte Flaschenhälse zu umschiffen.

2.3.1 Speichermodelle

Angenommen wir haben zwei Threads, die den Code aus Abbildung 2.2 ausführen. Initial seien `a` und `b` beide 0. Welche Werte haben `x` und `y`, nachdem beide Threads ausgeführt wurden? Offensichtlich hängt das Ergebnis von der Ausführungsreihenfolge ab. Allerdings würde man erwarten, dass mindestens eine der beiden Variablen den Wert 0 hat, da `x = a` oder `y = b` ausgeführt wird, bevor `a` und `b` verändert werden. Tatsächlich ist das nicht unbedingt der Fall.

`a == 0 && b == 0`	
Thread 1	**Thread 2**
`x = a;`	`y = b;`
`b = 2;`	`a = 1;`

Abbildung 2.2 *Beispiel für den Einfluss des Speichermodells*

Wie bereits in Kapitel 1 erwähnt, dürfen Compiler zur Optimierung Befehle umsortieren, und moderne Prozessoren arbeiten diese nicht immer in der gegebenen Reihenfolge ab, um die internen Einheiten bestmöglich auszulasten. Diese Umsortierungen ändern das Endergebnis eines *sequenziellen* Befehlsstroms zwar nicht, können aber in Kombination mit anderen Threads zu unerwarteten Ergebnissen führen. Abbildung 2.3 zeigt einen Ablauf, der zu dem Ergebnis `x == 1` und `y == 2` führt.

`a == 0 && b == 0`	
Thread 1	**Thread 2**
`b = 2;`	
	`y = b;`
	`a = 1;`
`x = a;`	
`x == 1 && y == 2`	

Abbildung 2.3 *Möglicher Ablauf im Falle einer Umsortierung*

Derartige Transformationen passieren auf mehreren Ebenen. Neben dem Prozessor und dem Compiler der Hochsprache führen auch Just-in-time-Compiler der