Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang

Abgabe: 05. Dezember 2013 in der Übung

Übung 12 Peterson Lock mit ThreadTools (5 Punkte)

Betrachten Sie das folgende Programmsegment, in dem zwei Threads eine gemeinsame Variable inkrementieren. Natürlich werden die Prozesse im Allgemeinen nicht sequentiell hintereinander ausgeführt, weshalb man nicht das bei sequentieller Ausführung zu erwartende Ergebnis 20 000 000 erreichen wird:

```
1  parallel increment
2  {
3    const int sections = 10000000;
4    int count = 0;
5
6    Process Π1                          Process Π2
7    {                                   {
8      for (int i=0; i<sections; i++)      for (int i=0; i<sections; i++)
9      {                                   {
10       count += 1;                         count += 1;
11     }                                   }
12   }                                   }
13 }
```

Den kritischen Abschnitt (critical section, CS) kann man beispielsweise mit dem in der Vorlesung vorgestellten Peterson-Algorithmus absichern. Das zeigt in unserer abstrakten Notation folgendes Listing:

```
1  parallel increment-peterson
2  {
3    const int sections = 10000000;
4    int in1 = 0, in2 = 0, last = 1;
5    int count = 0;
6
7    Process Π1                          Process Π2
8    {                                   {
9      for (int i=0; i<sections; i++)      for (int i=0; i<sections; i++)
10     {                                   {
11       in1 = 1;                            in2 = 1;
12       // *                                // *
13       last = 1;                           last = 2;
14       // *                                // *
15       while (in2 ∧ last == 1);            while (in1 ∧ last == 2);
16         count += 1;                         count += 1;  // CS
17       in1 = 0;                            in2 = 0;
18     }                                   }
19   }                                   }
20 }
```

Teilaufgabe (a)

Auf der Vorlesung-Homepage stehen ihnen in einer `zip`-komprimierten Datei `threadtools.zip` die ThreadTools zur Verfügung. Diese sind einige Wrapper-Klassen, die die Funktionalität der `PThreads` durch Objektorientierung vereinfachen. In der Vorlesung wurden sie *ActiveObjects* genannt. Beachten Sie zu den ThreadTools unbedingt die Hinweise am Ende des Blattes und auf der Homepage. Sie finden bei den ThreadTools eine Implementierung des oben angegebenen naiven Peterson-Locks in der Datei `checkpeterson.cc`. Das Programm können Sie mit dem Befehl `make` übersetzen. Testen Sie mit mindestens 10 Durchläufen, ob die Variable `count` das „richtige“ Ergebnis liefert. Schauen Sie als nächstes im Makefile die Zeilen 4 und 5 an:

```
4 CFLAGS     = -g -O0 -c -Wall
5 #CFLAGS    = -O3 -c
```

Kommentieren Sie Zeile 4 aus und Zeile 5 ein, und kompilieren Sie neu mit `make clean` und danach `make`. Dadurch erzeugen Sie optimierten Code. Wiederholen Sie die Messungen. Erhalten Sie im nicht-optimierten bzw. optimierten Fall die korrekten Ergebnisse? Haben Sie eine Erklärung, warum der Peterson-Lock auch im nicht-optimierten Fall nicht funktioniert?

Teilaufgabe (b)

In der Datei `membarrier.hh` finden Sie eine sogenannte *Memory Barrier*, realisiert durch einen Assembler-Befehl. Mittels dieser Memory Barrier lässt sich die *out-of-order-execution* manuell beeinflussen bzw. verhindern. Zur Erinnerung: Mittels out-of-order-execution können Maschinenbefehle vorgezogen werden, falls sich die zur Ausführung benötigten Daten bereits fertig berechnet im Speicher befinden. Die Memory Barrier kann nun vorgeben, in welcher Reihenfolge Befehle oder Speicheroperationen ausgeführt werden:

```
1 OP_1;
2 ...
3 OP_n;
4 memBarrier();
5 OP_{n+1};
```

Die Memory Barrier erzwingt hier, dass die Operationen `OP_1` bis `OP_n` vollständig ausgeführt sind, bevor `OP_{n+1}` bearbeitet werden darf. Fügen Sie nun an den mit einem Stern `// *` gekennzeichneten Stellen eine Memory Barrier ein, und testen Sie erneut mehrere Male mit und ohne Optimierung (vor einem erneuten `make` unbedingt *immer* ein `make clean` ausführen!). Wie verhält sich Ihr Programm? Diskutieren Sie kurz Ihre Beobachtungen.

Teilaufgabe (c)

Entfernen Sie die `memBarrier()`-Aufrufe nun wieder. In C/C++ können Speicherbereiche so markiert werden, daß die Reihenfolge von Lese- und Schreiboperationen auf diesen Speicherbereichen beim Übersetzen des Programms nicht verändert werden darf. Dies geschieht mit dem Schlüsselwort `volatile`. Lese- und Schreiboperationen auf `volatile`-Variablen gelangen in exakt im Programm stehender Reihenfolge in das übersetzte Programm und dürfen nicht wegoptimiert werden. Machen Sie die vier gemeinsamen Variablen `in[0]`, `in[1]`, `last` und `count` zu `volatile`-Variablen. Übersetzen Sie wiederum optimiert und nicht-optimiert und wiederholen Sie das Experiment wie in (a) und (b) mehrmals. Was beobachten Sie? Versuchen Sie, die von Ihnen beobachteten Effekte zu erklären.

Teilaufgabe (d)

Wiederholen Sie Teilaufgabe (c), nun allerdings wieder mit dem Aufruf der Memory Barrier an den gekennzeichneten Stellen. Welche Effekte beobachten Sie nun?

Übung 13 ThreadTools: Semaphoren (5 Punkte)

In der Vorlesung habe Sie das Konzept der Semaphore sowie die inaktives Warten über Bedingungsvariablen kennengelernt. In den in der letzten Aufgabe vorgestellten ThreadTools gibt es eine Klasse `Semphore`, die den Datentyp Semaphore über Bedingungsvariablen realisiert. Dazu ist Sie von der von der Klasse `Condition` abgeleitet und hat folgendes Layout:

```
1  /** Implements a semaphore deriving from a condition variable.
2   */
3  class Semaphore : private Condition<unsigned int>
4  {
5  public:
6
7    //! \brief make a semaphore with initial value
```

```
8     Semaphore (int init);
9
10    //! \brief make a semaphore with initial value 0
11    Semaphore ();
12
13    //! \brief decrement value
14    void P ();
15
16    //! \brief release semaphore
17    void V ();
18  };
```

Ihre Aufgabe ist es, die P- und V-Methoden der Semaphore zu implementieren. Dazu benötigen Sie die Methoden `acquire()`, `wait()`, `release()` und `signal()` der Basisklasse `Condition`. Die `Condition` enthält außerdem eine Variable `value`, die von der Semaphore erhöht oder erniedrigt wird. Die bereitgestellten ThreadTools enthalten eine Datei `semaphore.hh` mit obiger Header-Information sowie zusätzlich in der Datei `semaphore.cc` das Grundgerüst, deren Methoden Sie ausimplementieren müssen.

Übung 14 ThreadTools: Erzeuger-Verbraucher-Problem mit Ringpuffer (10 Punkte)

In der Vorlesung haben Sie besprochen, wie Erzeuger-Verbraucher-Probleme mit Semaphoren gelöst werden können. Dieses Problem sollen Sie nun mit den ThreadTools (in der Vorlesung `ActiveObjects` genannt) implementieren: Ein Puffer wird vom Erzeuger gefüllt. Ist der Erzeuger am Ende des Puffers angelangt, beschreibt er den Pufferanfang neu, ältere dort gespeicherte Aufträge werden überschrieben. Ein Verbraucher liest zu bearbeitende Aufgaben vom Pufferende. Für das Programm brauchen Sie

- einen Puffer, der den gewünschten Datentyp speichern kann,
- eine Semaphore, die freie Pufferplätze absichert,
- eine Semaphore, die belegte Pufferplätze absichert.
- Initialisieren Sie die Semaphore, die freie Pufferplätze absichert, zu Beginn mit der Anzahl aller Pufferplätze, da diese initial frei sind.

Schreiben Sie nun Klassen, die Erzeuger und Vebraucher implementieren. Diese sollen von der Klasse `BasicThread` abgeleitet werden und müssen die virtuelle Methode `run()` überladen, in der Erzeuger und Verbaucher ihre Aufgaben durchführen. Verwenden Sie für die Implementierung das bereitgestellte leere Gerüst `producerconsumer.cc`. Dieses kann durch Aufruf von `make` ohne Anpassung des `Makefiles` übersetzt werden. Die Aufgabe benötigen Sie Ihre Lösung aus der vorherigen Semaphoren-Aufgabe. Wer diese nicht lösen kann, melde sich bitte per E-Mail beim Tutor.

Testen Sie Ihr Programm mit einer Pufferlänge von 5. Der Puffer darf beliebige Datentypen speichern. Der Erzeuger soll in einer Schleife 20 Stücke produzieren und als produzierte Ware die aktuelle Schleifen-Zahl in den Puffer schreiben. Lassen Sie beide Threads ausgeben, welchen Pufferplatz sie gerade bearbeiten und welche Ware sie dort abgelegt bzw. gefunden haben.

Hinweise zu den ThreadTools (ActiveObjects)

Die in der Vorlesung eingeführten `ActiveObjects` heissen in unseren Übungen `ThreadTools`. Ein Thread wird durch die Basisklasse `BasicThread` realisiert.

- Auf der Vorlesungs-Homepage finden Sie ein zip-Archiv `threadtools.zip`, das die Klassen für die `BasicThreads`, allgemeine Werkzeuge zum Thread-Handling und einige Beispiele bereitstellt. Es ist ein Makefile beigelegt, übersetzen können Sie unter Linux mit `make`.
- Verwenden können Sie alle ThreadTools durch Inkludieren der Datei `tt.hh`. Für die Lösung der Semaphoren-Aufgabe müssen Sie das `Makefile` nicht anpassen.
- Für die Lösung des Erzeuger-Verbraucher-Problems verwenden Sie bitte das bereitgestellte Gerüst `producerconsumer.cc`. Dieses ist im `Makefile` als target angegeben und wird bei Eingabe von `make` übersetzt.
- Das Gerüst `resources.cc` benötigen wir auf dem nächsten Aufgabenblatt.
- Beachten Sie die ausführlichen Hinweise auf der Homepage.