Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, J. Pods

Abgabe: 11. Dezember 2012, 12 Uhr, per E-mail an `jurgis.pods@iwr.uni-heidelberg.de`

Übung 12 Peterson Lock mit Threads (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)

Seit 2011 gibt es einen neuen C++-Standard, C++11. Dieser enthält nun auch Threads in der Standardbibliothek und hat zu diesem Zweck ein komplett auf *concurrent programming* basierendes Speichermodell in die Sprache integriert. Die neuen Thread-Klassen basieren auf den `PThreads` aus C-Zeiten, haben jedoch eine echte (gewöhnungsbedürftige) C++11-Syntax bekommen. Beachten Sie zur Verwendung unbedingt die Hinweise am Ende des Blattes!

Auf der Homepage stehen Ihnen in einer `zip`-komprimierten Datei `c++11threadtools.zip` einige Hilfsdateien zur Verwendung der C++11-Threads zur Verfügung. In der Datei `checkpeterson.cc` befindet sich eine Implementierung des oben angegebenen naiven Peterson-Locks. Das Programm können Sie mit dem Befehl `make checkpeterson` übersetzen. Testen Sie mit mindestens 5 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 -std=c++11 -pthread
5 #CFLAGS    = -O3 -c -std=c++11 -pthread
```

Kommentieren Sie Zeile 4 aus und Zeile 5 ein, und kompilieren Sie neu mit `make clean` und danach `make checkpeterson`. 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 PHLR::memoryBarrier();
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?

Teilaufgabe (e)

Eine alternative zu den auf Assemblerbefehlen basierenden Memory Barriers ist die Verwendung von *atomaren* Anweisungen. In C++11 sind diese jetzt Teil der Sprache. Verwenden Sie für die Zählvariable statt `int count` nun `std::atomic<int> count`. Was ergibt sich jetzt?

Übung 13 Threads: Semaphoren (5 Punkte)

In der Vorlesung habe Sie das Konzept von Semaphoren sowie von inaktivem Warten über Bedin-

gungsvariablen kennengelernt. Interessanterweise gibt es in der C++-Bibliothek keine Implementierung einer Semaphore, wir müssen also selbst eine erstellen. In dem auf der Homepage zur Verfügung gestellten Archiv `c++11threadtools.zip` hat die Klasse `Semaphore` folgendes Layout:

```
1  class Semaphore
2  {
3          public:
4
5                  //! \brief make a semaphore with initial value
6                  Semaphore (int init);
7
8                  //! \brief make a semaphore with initial value 0
9                  Semaphore ();
10
11                 //! \brief decrement value
12                 void P ();
13
14                 //! \brief release semaphore
15                 void V ();
```

Ihre Aufgabe ist es, die P- und V-Methoden der Semaphore implementieren. Neben dem Wert, den die Semaphore verwalten soll, brauchen Sie noch einen `std::mutex` und eine `std::condition_variable` Die Datei `semaphore.hh` enthält obige Header-Information, in der Datei `semaphore.cc` befindet sich das Grundgerüst, deren Methoden Sie ausimplementieren sollen.

Beachten Sie die Schnittstellen der C++-Objekte. Ein guter Startpunkt ist die Thread-Referenz auf `http://en.cppreference.com/w/cpp/thread`. Machen Sie sich außerdem mit dem RAII- Prinzip vertraut (`http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization`), das in C++ extensiv genutzt wird.

Übung 14 Threads: 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 C++ Threads 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 Funktionen, die Erzeuger und Vebraucher implementieren. Verwenden Sie für die Implementierung das bereitgestellte leere Gerüst `producerconsumer.cc`. Dieses kann durch Aufruf von `make` ohne Anpassung des `Makefiles` übersetzt werden. Für diese 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-Zählvariable 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 Threads in C++

Das Thread-Basisobjekt `std::thread` existiert seit der Veröffentlichung von C++11 in der Standardbibliothek. Im Gegensatz zu anderen Programmiersprachen wie z.B. Java leitet der Programmierer jedoch nicht von dieser Basisklasse ab, um einen Thread zu implementieren. Stattdessen wird dem Thread-Objekt im Konstruktor eine Funktion übergeben, die der Thread ausführen soll. Ein Beispiel:

```
// This function will be called from a thread
void call_from_thread() {
  std::cout << "Hello World!" << std::endl;
}
```

```
int main() {
  // Launch a simple thread
  std::thread t1(call_from_thread);
  // Join the thread with the main thread
  t1.join();
```

In der main-Methode wird ein `std::thread`-Objekt erzeugt, dem ein Pointer auf die Function `call_from_thread` übergeben wird. Der Thread führt diese Methode aus und wird mittels `t1.join()` wieder mit dem Hauptthread (der, der die main-Methode ausführt) verbunden, also beendet.

Für einfache Funktionen reicht diese Syntax aus. Will man jedoch ein Thread-Objekt mit einem Zustand versehen, kommt man so schnell an sein Grenzen. Der C++11-Standard setzt für diese Fälle auf die sogenannten *lambda functions* [2], also ad hoc definierte anonyme Funktionen, mit denen beliebig komplexe Funktionen zusammengestellt werden können. Ein Beispiel:

```
// Class implementing a stateful thread
class MyThread
{
  public:
    MyThread(int i)
    {
      value = i;
    }

    void run()
    {
      std::cout << "Hello World, value = " << value << std::endl;
    }

    void run(int myvalue1, int myvalue2)
    {
      if(myvalue1 > myvalue2)
        std::cout << "Hello World, myvalue = " << myvalue1 << std::endl;
      else
        std::cout << "Hello World, myvalue = " << myvalue2 << std::endl;
    }

  private:
    int value;
};
```

Der Thread ist nun in eine Klasse gekapselt, in der main-Methode wird ein Objekt dieser Klasse erzeugt und mittels einer Referenz an eine lambda function übergeben, die wiederum im Konstruktor des Threads `t2` übergeben wird:

```
  MyThread mythread(12);

  // Launch a slightly more complex thread using a lambda function
  std::thread t2( [&mythread]() { mythread.run(); } );
  t2.join();

  // Use a lambda function with parameters
  std::thread t3( [&mythread](int v1, int v2) { mythread.run(v1,v2); }, 5, 3 );
  t3.join();
  std::thread t4( [&mythread](int v1, int v2) { mythread.run(v1,v2); }, 2, 13 );
  t4.join();
```

Am Beispiel der Threads `t3, t4` wird gezeigt, wie man zusätzliche Parameter an die lambda function übergeben kann. Die Parameter werden einfach als komma-separierte Liste hinter der lambda function an den Thread übergeben.

Das Ganze kann man beliebig komplex gestaltet. Besonders unschön wird es, wenn man nun noch templatisierte Klassen verwendet, da man dann innerhalb einer Zeile den gesamten Klammer-Zoo `(), {}, [], <>` verwenden darf.

Auf der Vorlesungs-Homepage finden Sie ein zip-Archiv `c++11threadtools.zip`, das einige Werkzeuge zum Thread-Handling und Beispiele bereitstellt.

[2] `http://en.wikipedia.org/wiki/Anonymous_function#C.2B.2B`

- Das oben angeführte Beispiel befindet sich in `threads_test.cc`
- Es ist ein Makefile beigelegt, übersetzen können Sie unter Linux mit `make`.
- Für die Lösung der Semaphoren-Aufgabe müssen Sie das `Makefile` nicht anpassen.
- Für die Lösung des Erzeuger-Vebraucher-Problems verwenden Sie bitte das bereitgestellte Gerüst `producerconsumer.cc`. Dieses ist im `Makefile` als target angegeben und wird bei Eingabe von `make` übersetzt.
- Die Gerüste `resources.cc,tsp.cc` benötigen wir auf dem nächsten Aufgabenblatt.
- Beachten Sie die ausführlichen Hinweise auf der Homepage.