timon.kelter [☺] tu-dortmund.de

Übung zur Vorlesung
Rechnerarchitektur
Sommersemester 2014

# Übungsblatt 11

**(10 Punkte)**

**Besprechung am Montag, 7. Juli 2014**

## 11.1 Locks und Synchronisation (3 Punkte)

In der Vorlesung wurden verschiedene Methoden zur hardwareunterstützten Realisierung eines Locks gezeigt.

- Test & Set
- Test & Set mit exponentiellen Backoff
- Test & Test & Set
- Load-Linked/Store-Conditional
- Fetch & Increment

Skizzieren Sie das Vorgehen der einzelnen Verfahren in Pseudocode und stellen Sie dabei die Eigenschaften der Verfahren gegenüber.

## 11.2 Barrieren (2 Punkte)

Der folgende Quelltext aus den Vorlesungsfolien implementiert eine Variante der Barrierensynchronisation.

```
class Barrier {
  int counter;
  Mutex mutex;
  boolean flag = false;

  void barrier( int p );
}
```

```
void barrier( int p ) {
  mutex.lock();
  if ( counter == 0 ) flag = false;
    int mycount = ++counter;
  mutex.unlock();
  if ( mycount == p ) {
    counter = 0;
    flag = true;
  } else
    while ( flag == false ) {};
}
```

Verändern Sie den Quelltext, gegebenenfalls durch Hinzufügen weiteren Pseudocodes, so dass die folgenden Ziele erreicht werden. Es ist eine Lösung gesucht, die folgende Forderungen erfüllt.

a) Ermöglichung der $n$-fachen / mehrfachen Nutzung der Barriere durch die teilnehmenden Threads.

b) Verhindern von Deadlocks, falls ein teilnehmender Thread vor Erreichen der Barriere terminiert (z. B. durch unbehandelte Ausnahme).

## 11.3 Parallelisierung mit OpenMP (5 Punkte)

Zur Nutzung von Parallelrechnern bieten sich verschiedene Möglichkeiten, je nachdem welche Art von Parallelität vorliegt. Während Task-Parallelität berücksichtigt werden kann indem für jeden Task ein eigener Thread angelegt wird, ist Datenparallelität auf diese Weise nur umständlich zu realisieren. In dieser Aufgabe sollen Sie das Parallelitäts-Framework OpenMP nutzen um einen einfachen Algorithmus zu beschleunigen.
OpenMP ist eine Erweiterung von C++, die Informationen zur Parallelität in C++ Pragmas ablegt. Diese Pragmas werden dann vom C++-Compiler, der OpenMP-kompatibel sein muß, ausgelesen und der damit annotierte Code wird automatisch parallelisiert. Dieses Vorgehen ermöglicht eine sehr kompakte Formulierung von Datenparallelität, i.W. in den Fällen in denen verschiedene Iterationen einer Schleife parallel in mehreren Threads ausgeführt werden sollen.

Informieren Sie Sich für diese Aufgabe unter `https://computing.llnl.gov/tutorials/openMP/` über die Syntax von OpenMP, speziell über die folgenden Punkte:

- Schleifenparallelisierung,
- Synchronisationsmechanismen und
- Datenzugriffs-Annotationen.

Zur Bearbeitung der Aufgabe benötigen Sie einen OpenMP-fähigen C++-Compiler, Beispiele sind:

- Der *GNU C Compiler (Linux)*. Kostenlos, Aufruf als `g++ -fopenmp source.cpp`
- *Microsoft Visual C++ Professional/Ultimate (Windows)*. Über das DreamSpark-Premium-Programm bei der IRB für den Privatgebrauch kostenlos erhältlich. Aufruf als `CL /openmp source.cpp`

Falls Ihnen kein eigener Rechner zur Verfügung stehen sollte, können Sie diese Aufgabe auch im Terminalpool des Lehrstuhl 12 bearbeiten.

a) Schreiben Sie ein C++-Programm, daß 10MB zufälligen Text generiert. Nutzen Sie für die Generierung des Textes folgende Funktion:

```
void initText( string &s ) {
  static const char alpha[] =
    "abcdefghijklmnopqrstuvwxyz";

  for ( int i = 0; i < s.length(); i++ ) {
    s[i] = alpha[rand() % (sizeof(alpha) - 1)];
  }
}
```

b) Implementieren Sie eine einfache Textsuche, die einen gegebenen, festen Text $S$ in dem zufällig generiertem Text $T$ sucht indem sie $S$ an jeder möglichen Stelle an $T$ "anlegt" und die Zeichen von $S$ der Reihe nach mit denen von $T$ an der entsprechenden Position vergleicht. Implementieren Sie eine Zeitmessung für die Suchfunktion mithilfe der C-Funktion clock. Achten Sie hierbei darauf, daß bei der Suche *keine* I/O-Operationen durchgeführt werden (wie z.B. das Ausgeben von Nachrichten auf `cout` oder Aufrufe von `printf`), da diese die Zeitmessung verfälschen.

c) Testen Sie verschiedene Möglichkeiten Ihre Implementierung aus Teil b) mithilfe der OpenMP-Pragmas zu parallelisieren. Messen Sie auch hier jeweils die Zeit und geben Sie eine Parallelisierung an, die auf dem von Ihnen benutzten Rechner gut funktioniert.

d) Ergänzen Sie ihre Lösung aus c) so, daß die Anzahl Treffer bei der Suche ermittelt wird. Die Ermittlung soll hier online während der Suche geschehen und nicht erst nachdem der komplette Text $T$ durchsucht wurde und damit alle Treffer gefunden wurden. Nutzen Sie die Synchronisierungmechanismen von OpenMP um die Korrektheit des Ergebnisses sicher zu stellen.

e) Können Sie die Korrektheit in Teil d) auch mithilfe einer *Barriere* sicher stellen? Wenn ja, wie?