Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2016	**Aufgabenblatt 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao, Dr. Peter Janacik, Tutoren
Abgabetermin:	[1] – 22.05.2016 23:55 Uhr [2] – 29.05.2016 23:55 Uhr

Aufgabe 2.1: Prozesse und Threads (1 Punkt) (Theorie[1])

a) Welche Arten der Prozessumschaltung gibt es? Beschreiben Sie diese kurz in eigenen Worten. (0,2 Punkte)

b) Benennen und erklären Sie die Prozesszustände und skizzieren Sie deren Zusammenhänge bzw. Übergänge. Swapping kann hierbei ignoriert werden. (0,2 Punkte)

c) Beschreiben Sie die Eigenschaften und Unterschiede von User-Level und Kernel-Level Threads. (0,2 Punkte)

d) Nehmen Sie an, Sie erhalten die Aufgabe, einen Netzwerk-Datenbank-Server zu implementieren. Zu verwenden sind hierbei Prozesse und Threads. Grenzen Sie zu allererst Prozesse und Threads voneinander ab. Beschreiben Sie nun grob, wann und wofür die Threads bzw. Prozesse benutzt werden. Begründen Sie Ihre Antworten. (0,4 Punkte)

Aufgabe 2.2: Parallelität (1 Punkt) (Theorie[1])

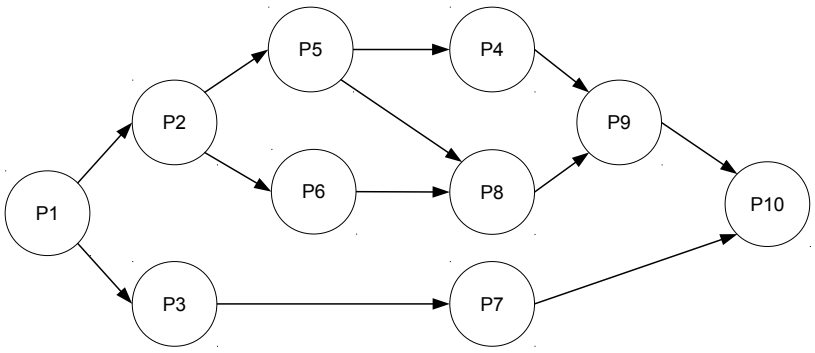

Abbildung 1: Abhängigkeitsgraph

Gegeben sei der Abhängigkeitsgraph aus Abbildung 1. Setzen Sie diesen mit Hilfe der aus der Vorlesung bekannten Befehle `fork/join` und `parbegin/parend` (jeweils 0,5 Punkte) in Pseudocode um.

Aufgabe 2.3: Parallelisierung I (Tafelübung)

Wie unterscheidet sich die Herangehensweise von `parbegin/parend` und `fork/join`?

Skizzieren Sie mit Hilfe der Befehle `fork`/`join` sowie `parbegin`/`parend` ein Programm in Pseudocode zu dem Prozessvorgängergraphen in der Abbildung 2.

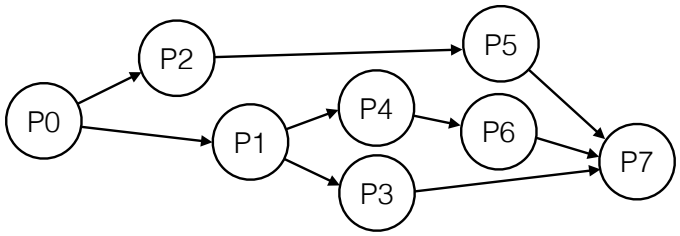

Abbildung 2: Prozessvorgängergraph der Prozesse P0 bis P7

Aufgabe 2.4: Parallelisierung II (Tafelübung)

Gegeben ist das folgende nicht-parallele C-Programm. Die Funktionen `jobA` ... `jobF` wurden zuvor im Programm implementiert und enthalten längerlaufende Berechnungen.

```
01  int main(void) {
02      int a,b,c,d,e,f,g,h,i;
03
04      a = jobA();
05      b = jobB(a);
06      c = jobC(a);
07      d = jobD(a,b,c);
08      e = jobE(a,b,c);
09      f = jobF(e,c,b);
10      g = jobG(a,c,f);
11      h = jobH(a,d,c);
12      i = jobI(d,f,h);
13      return jobK(a,f,g,h,i);
14  }
```

a) Welche Zeilen sind unabhängig voneinander und können in ihrer sequenziellen Reihenfolge verändert werden?

b) Zeichnen Sie einen Prozessvorgängergraphen. Jede aufgerufene Funktion soll dabei einem Task bzw. Prozess/Thread entsprechen.

c) Schreiben Sie basierend auf dem Prozessvorgängergraphen ein Programm in Pseudocode mit `fork`/`join` und `parbegin`/`parend`, das möglichst viele Funktionen parallel ausführt.

Aufgabe 2.5: Prozesswechsel (Tafelübung)

Was unterscheidet ein kooperatives von einem präemptiven Multitaskingsystem? Simulieren Sie den Prozesswechsel bei einem präemptiven System.

Aufgabe 2.6: Bitcoin Mining (3 Punkte) (Praxis^2)

In dieser Praxisaufgabe soll die Berechnung von Blockhashes parallelisiert werden, die im Rahmen des Bitcoin-Systems benötigt wird. Mehr Informationen zum Blockhash und zur Funktionsweise von Bitcoin finden Sie in diesem Video: `www.youtube.com/watch?v=Lx9zgZCMqXE` (Blockhash: ca. ab 10. Minute)

Das Bitcoin-System basiert auf einem öffentlichen Buchungssystem, der Block-Kette. Alle Transaktionen von Bitcoins werden in dieser Blockkette festgehalten, um die Integrität und Richtigkeit zu gewährleisten. Damit Transaktionen von Bitcoins stattfinden können, müssen diese in die Blockkette eingetragen werden. Um Transaktionen zu bestätigen und damit diese in die Blockkette aufzunehmen, müssen diese in Blöcke zusammengefasst werden. Ein solcher Block muss bestimmten Regeln entsprechen, um in die Blockkette aufgenommen zu werden.

Ein Block enthält folgende Informationen:

Version:	Die Bitcoin Versionsnummer
Previous Block hash:	Validierter Hash des vorherigen Blockes. Dadurch entsteht eine immer weiterführende Verkettung von Blöcken.
Merkle Root:	Ein spezifischer Hash, der einzelne Transaktionen zusammenfasst und so in den Block einfügt.
Timestamp:	Ein Zeitstempel
Bits:	Repräsentiert die Schwierigkeit des Blockes, d.h., wie viele Nullen in den ersten Stellen des Hashes erscheinen müssen.
Nonce:	Da der Hash einer konstanten Eingabe immer gleich ist, wird der nonce-Wert eingeführt. Dieser Wert wird angepasst, um einen validierenden Hash zu finden.

Aus diesen Informationen wird ein Hash gebildet. Ein Hash ist aber erst dann valide, wenn er einem bestimmten Muster entspricht. Dieses Muster entspricht einer bestimmten Anzahl von Nullen in den ersten Stellen des Hashes. Aufgrund dieser Eigenschaft ist die Berechnung von Blöcken sehr schwierig aber paralellisierbar.

Ihre Aufgabe ist es, die Funktionen `bitcoin_loop` und `bitcoin_parallel` angelehnt an die Funktion `bitcoin_simple` selbst zu implementieren. Die Funktionen `bitcoin_loop` und `bitcoin_parallel` sollen zum gleichen Ergebnis führen wie die Funktion `bitcoin_simple`, dieses aber auf dem nachfolgend in den Teilaufgaben a und b definierten Weg erreichen. Deshalb können Sie wesentliche Teile der Funktion `bitcoin_simple` übernehmen. Um die Bearbeitung zu vereinfachen, sind relevante Stellen im Code ausgiebig kommentiert.

a) Implementieren Sie die Funktion `bitcoin_loop`, sodass zunächst die Berechnungen der Hashes in eine in einem Parameter festgelegte Anzahl an Ausschnitten zerlegt werden. Anschließend sollen in einer Schleife die jeweiligen Ausschnitte durch Aufruf der Funktion `calculate_hash` einzeln berechnet werden. (1 Punkt)

b) Die Berechnung soll nun durch Verwendung von mehreren Prozessen parallel ausgeführt werden. Verwenden Sie die Funktion `fork`, um jeden Ausschnitt in einem separaten Prozess zu berechnen. Die Implementierung soll in der Funktion `bitcoin_parallel` erfolgen. (2 Punkte)

Hinweise:

- **Kompilieren und Ausführen:** Um die Kompilierung des aus mehreren Dateien bestehenden Projektes zu vereinfachen, haben wir ein make-File vorgegeben. Zum Kompilieren reicht es, in der Kommandozeile `make` einzugeben. Die ausführbare Datei heißt `bitcoin` und erwartet als Parameter die Anzahl der Prozesse, die gestartet werden sollen (siehe auch die README Datei). Unter Solaris sollte stattdessen `gmake` verwendet werden.
- **fork/join in C:** Unter Unix (auch Cygwin) können Sie Prozesse mit der Funktion `fork` abspalten. Mit der Funktion `wait` können Sie auf das Ende von Kind-Prozessen warten. Nähere Informationen erhalten Sie auf den entsprechenden man-Pages, wenn Sie in der Kommandozeile man `fork` bzw. `man wait` eingeben.
- **Prozessanzahl:** Bitte beachten Sie, dass die Anzahl der Prozesse, die für die Berechnung verwendet werden, variabel ist und ihr Code somit auch bei einer ungeraden Anzahl von Prozessen (bzw. Funktionsaufrufen) korrekt funktionieren muss.
- **Speicher:** Achten Sie darauf, nicht unnötig viel Speicher zu allozieren.
- **File Handle in Unterprozessen:** Wenn mittels `fork` ein neuer Prozess erzeugt wird, werden alle offenen file handles mitkopiert. Auf einigen Systemen kann dies zu Fehlern führen. Es wird daher empfohlen, diese vor dem `fork` zu schließen und anschließend (ggf. in beiden Prozessen) neu zu öffnen. Zudem ist ein `fflush(stdout)` vor dem forken hilfreich, um mehrfache `printf`-Ausgaben zu vermeiden.