Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2015	**Aufgabenblatt 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao, Dr. Peter Janacik, Tutoren
Abgabetermin:	[1] – 11.05.2015 [2] – 18.05.2015

Aufgabe 2.1: Prozesse und Threads (1,2 Punkte) (Theorie[1])

a) Was unterscheidet Prozesse und Threads voneinander? In welchen Situationen eignen sich bspw. jeweils Prozesse und Threads besser? (0,4 Punkte)

b) Nehmen Sie an, Sie möchten einen Videoserver (ein Server, der Video Content an die Besucher einer Website ausliefert) implementieren und haben dafür die Wahl zwischen Prozessen und Threads. Welches dieser Konzepte setzen Sie ein? Wie organisieren Sie dabei die Prozesse oder Threads? Begründen Sie Ihre Antwort. (0,8 Punkte)

Aufgabe 2.2: Parallelisierung I (0,8 Punkte) (Theorie[1])

Gegeben ist folgender Abhängigkeitsgraph.

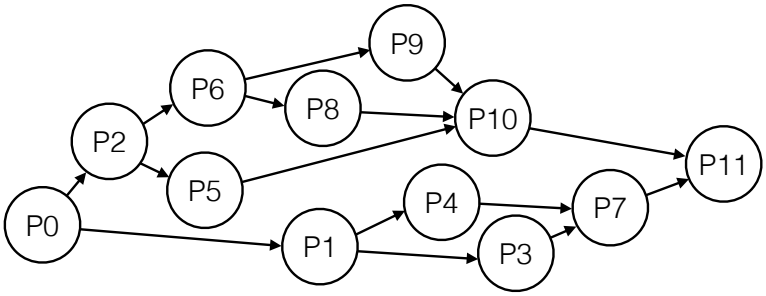

Abbildung 1: Abhängigkeitsgraph

Skizzieren Sie mit Hilfe der Befehle `fork`/`join` sowie `parbegin`/`parend` (jeweils 0,4 Punkte) ein Programm in Pseudocode zu dem Graphen in der Abbildung 1.

Aufgabe 2.3: Parallelisierung II (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.

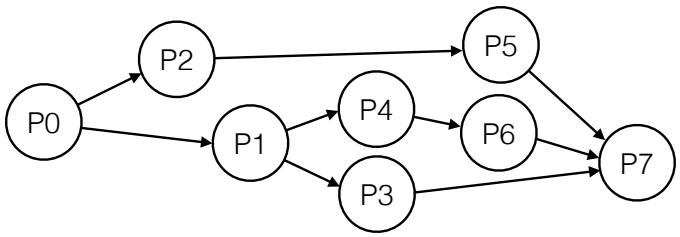

Abbildung 2: Prozessvorgängergraph der Prozesse P0 bis P7

Aufgabe 2.4: Parallelisierung III (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: Raytracer (3 Punkte) (Praxis^2)

In dieser Praxisaufgabe soll die Berechnung eines Bildes mittels Raytracing[1] parallelisiert werden. Das Ergebnis dieser Berechnung wird als Grafik im Bitmap-Format (.bmp) in eine Datei geschrieben.

Der Raytracer funktioniert in der Weise, dass für jedes Pixel ein Strahl verfolgt wird und der Schnittpunkt mit den Objekten berechnet wird. Die Farbe des Objektes an dieser Stelle ergibt dann die Farbe des Pixels. Bei Spiegelungen oder Lichtbrechungen kann der Strahl noch in weitere Richtungen weiterverfolgt werden. Da die Schnittpunktberechnung jedes Strahles unabhängig erfolgen kann, lässt sich dieser Prozess sehr gut parallelisieren.

Kern der Vorgaben ist die Funktion `raytrace`, welche die Grafik, bzw. einen kleineren Ausschnitt davon, berechnet. Um aus den errechneten Pixeldaten eine gültige Bitmap-Datei zu erzeugen, muss zuerst mit der Funktion `write_bitmap_header` ein passender BMP-Header an den Anfang der Ausgabedatei geschrieben werden, welchem dann die errechneten Pixeldaten folgen. Eine ausführliche Beschreibung der Funktionen befindet sich in den jeweiligen Header-Dateien in der Vorgabe. Die Verwendung der Funktionen kann der Funktion `raytracer_simple`, welche die Berechnung durch einen einfachen Aufruf der Funktion `raytrace` durchführt, in der Datei `main.c` entnommen werden.
Für das Ausführen und den Vergleich der Laufzeiten der unterschiedlichen Implementierungen können diese Vorgaben unverändert verwendet werden.
Ihre Aufgabe ist es, die Funktionen `raytracer_loop` und `raytracer_parallel`, angelehnt an die Vorgaben, selber zu implementieren. Um Ihnen die Bearbeitung zu vereinfachen, wurden die entsprechenden Stellen durch Kommentare im Code gekennzeichnet. Für jede der drei Funktionen wird eine eigene bmp Datei erstellt, an der Sie die richtige Funktionalität Ihrer Implementierung überprüfen können.

a) Implementieren Sie die Funktion `raytracer_loop`, sodass zunächst die Grafik durch eine per Parameter festgelegte Anzahl von Ausschnitten zerlegt wird. Anschließend sollen in einer Schleife die jeweiligen Ausschnitte durch Aufruf der Funktion `raytrace` einzelnd berechnet und beim Schreiben in der Ausgabedatei zu einem Bild zusammengefügt werden. Achten Sie darauf lediglich den benötigten Speicher für den zu berechnenden Ausschnitt zu allozieren. (1 Punkt)

b) Die Berechnung soll nun durch Verwendung von mehreren Prozessen parallel ausgeführt werden. Verwenden Sie die Funktion `fork`, um jeden Aufruf von `raytrace` von einem separaten Prozess ausführen zu lassen. Das Ergebnis jedes Prozesses soll in die gemeinsame Ausgabe-Datei geschrieben werden. Die Implementierung soll in der Funktion `raytracer_parallel` erfolgen. (2 Punkte)

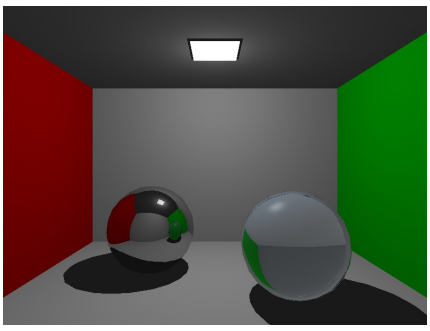

Abbildung 3: berechnete Szene

[1] Strahlenverfolgung. Details z.B. unter http://de.wikipedia.org/wiki/Raytracing

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 `raytracer` und erwartet als Parameter die Anzahl der Prozesse, die gestartet werden soll (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 unnötigen Speicher zu allozieren.
- **File Handle in Unterprozessen:** Wenn mittels `fork` ein neuer Prozess erzeugt wird, werden alle offenen file handles mit kopiert. 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.
- **Zugriff auf eine gemeinsame Datei:** Soll eine Datei zum Schreiben von mehreren Prozessen parallel geöffnet werden, so kann dazu mit `fopen` der Update-Modus (Parameter `"rb+"`) verwendet werden. Zum Springen innerhalb der Datei dient die Funktion `fseek`. Für eine genaue Beschreibung der Funktionen siehe `man fopen` und `man fseek`.