Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2019	**Aufgabenblatt 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	[1] – 19.05.2019 23:55 Uhr [2] – 19.05.2019 23:55 Uhr

Aufgabe 2.1: Parallelisierung I (Tafelübung)

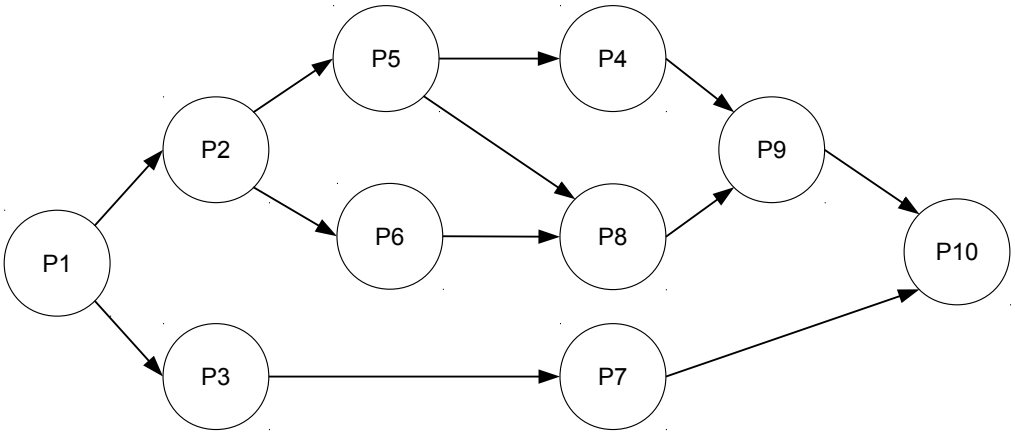

Abbildung 1: Abhängigkeitsgraph

Wie unterscheidet sich die Herangehensweise von parbegin/parend und fork/join? Gegeben sei obenstehender Abhängigkeitsgraph. Setzen Sie diesen mit Hilfe der aus der Vorlesung bekannten Befehle fork/join und parbegin/parend in Pseudocode um.

Aufgabe 2.2: Prozessmanagement (Tafelübung)

Benennen Sie die möglichen Zustände eines Prozesses und Skizzieren Sie die Übergange.

Aufgabe 2.3: Prozesse und Threads (1 Punkt) (Theorie[1])

a) Erklären Sie, was Prozesse sind und wofür sie benötigt werden. Gehen Sie dabei auf die Aussage „Prozesse sind instanziierte Programme“ ein. **(0,3 Punkte)**

b) Erklären Sie die Begriffe *Parallelität* und *Nebenläufigkeit*? **(0,2 Punkte)**

c) Nennen Sie zwei Ereignisse, die eine Prozessumschaltung zur Folge haben und erklären Sie, wie dieses Umschalten realisiert wird. Gehen Sie dabei auch auf den Begriff „Process Control Block“ (PCB) ein. **(0,2 Punkte)**

d) Was unterscheidet einen User Level Thread von einem Prozess? **(0,3 Punkte)**

Aufgabe 2.4: Parallelisierung II (1 Punkt) (Theorie[1])

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

```
int main(void)
{
    int a, b, c, d, e, f, g, h, i, j, k;
    a = jobA();
    b = jobB(a);
    c = jobC(a);
    d = jobD(c);
    e = jobE();
    f = jobF(c, e);
    g = jobG(d, f);
    h = jobH();
    i = jobI(g);
    j = jobJ(b, g);
    k = jobK(i, j);

    return k;
}
```

a) Zeichnen Sie einen Prozessabhängigkeitsgraphen, der die Abhängigkeiten der einzelnen Jobs darstellt. Jede aufgerufene Funktion soll dabei einem Task bzw. Prozess/Thread entsprechen. **(0,5 Punkte)**

b) Schreiben Sie basierend auf dem Graphen ein Programm in Pseudocode mit fork/join, das die Jobs möglichst effizient abarbeitet. **(0,5 Punkte)**

Aufgabe 2.5: Raytracing (3 Punkte) (Praxis[2])

Ziel dieser Aufgabe ist es, die Berechnungen für ein mittels Raytracing generiertes Bild zu parallelisieren. Die Funktionen zur Erzeugung des Bildes (Pixel-Daten) sind bereits vorgegeben. Ihre Aufgabe besteht darin, ein Programm zu schreiben, das die zu berechnenden Pixel sinnvoll auf mehrere Prozesse aufteilt und die Bildbereiche anschließend wieder zu einem Gesamtbild zusammenzufügt. Zu bearbeiten ist die Datei `main.c`. Beachten Sie die *Abgabeformalitäten*. Die Aufgabe gliedert sich in folgende Bestandteile:

a) Parsen der Eingabe **(0,5 Punkte)**:
Ihr Programm soll wie folgt aufgerufen werden können:

```
./raytracer <Anzahl der Prozesse> <Auflösung> <Anzahl der Samples>
```

Die Eingabeparameter sind im folgenden erklärt:

- **Anzahl der Prozesse:** Ist die Anzahl an Prozessen, auf die die Berechnungen aufgeteilt werden sollen. Muss einen Wert im Bereich von 1-128 (einschließlich) haben.
- **Auflösung:** Gibt die Breite des Bildes in Pixeln an. Die Höhe ist identisch zur Breite, demnach hat ein Bild mit dem Eingabeparameter `1024` die Maße `1024x1024`. Muss einen Wert im Bereich von 1-10000 (einschließlich) haben.

- **Anzahl der Samples:** Beschreibt, wie viele Samples pro Pixel berechnet werden sollen. Ein hoher Wert sorgt für glattere Kanten, ist jedoch auch rechenintensiver. Darf einen beliebig hohen Wert ≥ 1 haben.

Wandeln Sie die Eingabeparameter in einen angemessenen Datentyp um und überprüfen Sie sie auf ihre Richtigkeit (Anzahl der Parameter und erlaubtes Intervall). Die Reihenfolge der Eingabeparameter muss dabei eingehalten werden. Sind alle Parameter korrekt, so geben Sie anschließend die eingegebenen Parameter auf der Konsole aus. Andernfalls brechen Sie das Programm mit einer Fehlermeldung ab. Eine Beispielausgabe könnte wie folgt aussehen:

```
./raytracer 4 1024 16
Anzahl der Prozesse: 4
Auflösung: 1024x1024
Samples: 16
```

b) Arbeit Aufteilen **(1,5 Punkte)**:
Die Funktion `ray_renderScene()` zur Berechnung des Bildes, erhält unter anderem einen Parameter, der in Pixel-Koordinaten angibt, welcher rechteckige Teilbereich des Bildes berechnet werden soll. Dieser wird über die struct `bmp_Rect` beschrieben:

```
typedef struct{
    int x; /* x-position der oberen linken Ecke */
    int y; /* y-position der oberen linken Ecke */
    int w; /* Breite des Rechtecks */
    int h; /* Höhe des Rechtecks */
}bmp_Rect;
```

Der Ursprung `(x=0, y=0)` eines Bildes befindet sich in der oberen linken Ecke. Die X-Koordinate wächst nach rechts hin und die Y-Koordinate wächst nach unten hin. Beispielsweise beträgt bei einem Bild mit den Maßen `256x256` die Koordinate des unteren rechten Pixels `(x=255, y=255)`. Überlegen Sie sich nun, wie Sie das Gesamtbild in möglichst gleich große Teilbereiche (Rechtecke) unterteilen, so dass jeder Prozess einen Bereich erhält und das gesamte Bild abgedeckt ist. Der Elternprozess selbst soll dabei keinen Bildbereich erhalten, er ist nur für das Erzeugen der Kindprozesse verantwortlich. Rufen Sie in einer Schleife die Funktion `fork()` auf, um die einzelnen Kindprozesse zu erstellen. Jeder Kindprozess ruft die Funktion `ray_renderScene()` mit dem entsprechenden Teilbereich auf und speichert das Ergebnis (Rückgabewert) in einer separaten Datei ab. Hierfür kann die Funktion `bmp_saveToFile()` aus der Vorgabe verwendet werden.

c) Zusammenfügen **(1 Punkt)**:
Schließlich soll der Elternprozess mittels `waitpid()` auf die Terminierung aller Kindprozesse warten. Danach soll ein leeres Bild mittels `bmp_loadFromData()` erzeugt werden und dieses anschließend mit den Daten aus den Teilbildern gefüllt werden, sodass das Gesamtbild entsteht. Nutzen Sie hierzu die Funktion `bmp_stamp()` aus der Vorgabe. Die zuvor von den Kindprozessen abgespeicherten Teilbilder können mithilfe der Funktion `bmp_loadFromFile()` geladen werden. Speichern Sie das finale Bild unter dem Namen `final.bmp` ab.
Achten Sie auf korrekte Speicherverwaltung!

Wichtige Funktionen:

Folgend werden die wichtigsten Funktionen aufgelistet, welche zur Bearbeitung der Aufgabe notwendig sind.

- `pid_t fork()`

  Erzeugt eine Kopie des aktuellen Prozesses. Die Funktion fork gibt als Rückgabewert den pid des Kindprozesses zurück. Der Rückgabewert im Kindprozess ist stets 0.

  Weiterer Informationen: `man 2 fork`.

- `pid_t waitpid(pid_t pid, int *wstatus, int options)`

  Parameter:

  - `pid`: Der pid des Kindprozesses.
  - `wstatus`: Status des Kindprozesses.
  - `options`: Definiert das Verhalten der Funktion.
  - `return`: Die Kindprozess-ID (`pid`) oder -1 bei einem Fehler.

  In der Aufgabe sollte folgender Aufruf verwendet werden:
  `waitpid(<pid>, NULL, 0)`
  Mit `0` als Option wird die Ausführung des Prozesses welcher die Funktion aufgerufen hat so lange angehalten bis der Prozess mit dem angebenen pid beendet wurde. Zurückgegeben wird der pid selber oder `-1` falls ein Fehler aufgetreten ist.

  Weitere Informationen: `man 2 wait`

- ```
  bmp_Image* ray_renderScene(const bmp_Rect * canvas,
                             int resolution_w, int resolution_h,
                             int samples, const ray_Scene* scene,
                             const char * process_name)
  ```

  - `canvas`: Der Bereich des Bildes der berechnet werden soll.
  - `resolution_w`: Breite des Gesamtbildes.
  - `resolution_h`: Höhe des Gesamtbildes.
  - `samples`: Die Anzahl der Samples.
  - `scene`: Die Szene, die dargestellt werden soll.
  - `process_name`: Name des aktuellen Prozesses. Wird für die Statusmeldung verwendet. Wird `NULL` angegeben so wird die Statusmeldung (Prozentanzeige) unterdrückt.
  - `return`: Berechnetes Bild. Das zurückgegebene Bild besitzt die Breite und Höhe des angegebenen Teilbereichs (`canvas`). Das Bild muss vom Aufrufer wieder frei gegeben werden (`bmp_free()`).

  Die Funktion generiert den angegebenen Teilbereich des Gesamtbildes und gibt ihn als Bild zurück.

- `size_t bmp_saveToFile(const bmp_Image * img, const char * filename)`
  ```

  - `img`: Das Bild, das in eine Datei geschrieben werden soll.
  - `filename`: Dateiname unter dem die Bilddatei abgespeichert werden soll. Als Endung sollte `*.bmp` verwendet werden.
  - `return`: Anzahl der Bytes die geschrieben wurden oder 0 falls ein Fehler aufgetreten ist.

  Speichert ein Bild in einer Bitmap-Datei.

- `bmp_Image* bmp_loadFromFile(const char * filename)`

  - `filename`: Dateiname der Bitmap-Datei.
  - `return`: Das geladene Bild oder `NULL` falls die Datei nicht geladen werden konnte. Das Bild muss vom Aufrufer wieder frei gegeben werden (`bmp_free()`).

  Lädt ein Bild aus einer Bitmap-Datei.

- `bmp_Image* bmp_loadFromData(int w, int h, const Color * data)`

  - `w`: Breite des Bildes.
  - `h`: Höhe des Bildes.
  - `data`: Farbdaten, mit denen das Bild initialisiert werden soll. Kann auf `NULL` gesetzt werden, wenn keine Initialisierung benötigt.
  - `return`: Das neu erzeugte Bild. Das Bild muss vom Aufrufer wieder frei gegeben werden (`bmp_free()`).

  Erzeugt ein neues Bild mit der angegebenen Höhe und Breite.

- ```
  void bmp_stamp(const bmp_Image * src_img, bmp_Image * dst_img,
                 int dst_x, int dst_y)
  ```

  - `src_img`: Quelle. Das Bild von dem die Pixel-Daten kopiert werden sollen.
  - `dst_img`: Ziel. Das Bild in das die Pixel-Daten eingefügt werden sollen.
  - `dst_x`: Die X-Koordinate an der das Quell-Bild (obere linke Ecke) im Ziel-Bild eingefügt wird.
  - `dst_y`: Die Y-Koordinate an der das Quell-Bild (obere linke Ecke) im Ziel-Bild eingefügt wird.

  Kopiert die gesamten Pixel-Daten aus dem Quell-Bild an die angegebene Stelle im Ziel-Bild. Die Koordinaten (`dst_x, dst_y`) beschreiben die Position der oberen linken Ecke des Quell-Bildes relativ zur oberen linken Ecke des Ziel-Bildes.

- `void bmp_free(const bmp_Image * img)`

  - `img`: Das freizugebende Bild.

  Die Funktion gibt sämtlichen vom gegebenen Bild verwendeten Speicher frei.
  ```

Testen:
Probieren Sie ihr Programm mit einer unterschiedlichen Anzahl an Prozessen aus und messen Sie die Laufzeiten (z.B. mit `time`). Sie können sich Gedanken zu den folgenden Fragen machen (**optional**):

- Kann die Berechnungszeit (bei gleicher Auflösung und Sample-Zahl) durch Parallelität reduziert werden?
- Warum steigert sich die Geschwindigkeit, bei Erhöhung der Prozessanzahl über ein gewisses Maß hinaus, nicht?

Hinweise:

- **Abgabe:** Laden Sie den gesamten Code (Vorgabe mit bearbeiteter Datei `main.c`) als .zip Datei im Abgabeverzeichnis hochladen.
- **Vorgaben:** Sie können die Datei `main.c` beliebig durch eigene Funktionen/Variablen erweitern. Bitte halten Sie sich jedoch bei der Programmierung immer an die Vorgabe. Eine Missachtung kann zu Punktabzug führen.
- **Makefile:** Bitte verwenden Sie für diese Aufgabe das Makefile aus der Vorgabe