Weitere Hinweise zu den Übungen Betriebssysteme

- Die abgegebenen Antworten/Programme werden automatisch auf Ähnlichkeit mit anderen Abgaben überprüft. Wer beim Abschreiben[1] erwischt wird, verliert ohne weitere Vorwarnung die Möglichkeit zum Erwerb der Studienleistung in diesem Semester!
- Die optionalen Aufgaben (davon wird es jeweils eine auf den Aufgabenblättern 0–3 geben) sind ein Stück schwerer als die „normalen“ und geben *keine* zusätzlichen Punkte für das jeweilige Aufgabenblatt – aber jeweils ein „Bonus-Sternchen“ ★. Wenn ihr drei Sternchen sammelt, müsst ihr das letzte Aufgabenblatt (A4) nicht bearbeiten!

Aufgabe 2: Thread-Synchronisation (10 Punkte)

1. Scheduling (2 Punkte)

⇒ antworten.txt

Betrachtet die folgenden 3 Prozesse, die gleichzeitig in die Ready-Liste der CPU aufgenommen werden. Zur Vereinfachung sei angenommen, dass die Längen der CPU-Bursts und I/O-Bursts pro Prozess immer gleich lang sind und dass der Scheduler sie kennt:

Prozess-ID	CPU-Burst	I/O-Burst
A	3	4
B	6	5
C	2	2

Ihr könnt AnimOS verwenden um euch einen ersten Überblick über die verschiedenen Schedulingverfahren zu machen.

(`http://ess.cs.tu-dortmund.de/Software/AnimOS/`)

Achtung: In der Klausur habt ihr AnimOS nicht zur Verfügung, deswegen solltet ihr die Lösungen der Schedulingverfahren auch ohne AnimOS erarbeiten können.

1. Wendet auf die Prozesse A, B und C das Shortest-Process-Next (SPN) Scheduling-Verfahren aus der Vorlesung an. Geht dabei davon aus, dass der Scheduler die Länge der CPU-Bursts kennt und sie nicht vorhersagen muss.
   Notiert die CPU- und I/O-Verteilung für die ersten 30 ms. Prozesswechsel dauern 0 ms und sind damit zu vernachlässigen. Es gilt die Annahme, dass mehrere I/O-Vorgänge parallel ausgeführt werden können. Zu Beginn sind die Prozesse in alphabetischer Reihenfolge in der Bereit-Liste.
   Stellt eure Ergebnisse wie in der folgenden ASCII-Zeichnung dar, eine Spalte entspricht dabei einer Millisekunde (nutzt hierbei keine Tabulatoren!).
   ```
   +---+--------------+
   | A |     CCCCCCII | C: Prozess nutzt die CPU
   | B |   CCIIIII    | I: Prozess führt I/O-Operationen durch
   | C |CCCII      CCC|
   +---+--------------+
   ```
2. Welches Problem birgt Shortest-Process-Next als Scheduling-Verfahren?

[1] Da wir im Regelfall nicht unterscheiden können, wer von wem abgeschrieben hat, gilt das für Original **und** Plagiat.

2. Synchronisation (2 Punkte)

⇒ antworten.txt

1. Was sind Race-Conditions und warum sind sie gefährlich?
2. Was ist aktives Warten und welche Probleme bringt es mit sich?

3. Programmierung: Parallelisierung von Datenverarbeitungen (6 Punkte)

Aufgrund von modernen Prozessoren mit vielen Recheneinheiten war eine parallele Ausführung von Datenverarbeitungs-Operationen nie relevanter als heute. Eine häufig angewendete Methodik ist das Aufteilen einer einzigen Operation auf viele kleine Operationen, die jeweils einen Teilbereich der Daten bearbeiten. Diese kleinen Operationen können parallel von verschiedenen Prozessoren ausgeführt werden. Wir nennen diese Operationen im weiteren Verlauf **Working-Set**.

Ein solches Working-Set besteht aus:

- `int *input` - Pointer auf den Beginn der Eingabedaten (kann wie ein Array verwendet werden. Grenzen beachten!)
- `int *output` - Pointer auf das Ende der Eingabedaten (kann wie ein Array verwendet werden. Grenzen beachten!)
- int length - Länge des Ein-/Ausgabe-Arrays
- `int (*operation)(int, struct WorkingSet*)` - Funktionspointer auf die Operation, die auf den Daten ausgeführt werden soll. Der erste Parameter gibt den Index des Eingabefeldes an, auf dem die Operation durchgeführt werden soll. Der zweite Parameter bestimmt das WorkingSet, auf dem gearbeitet wird. Der Rückgabewert ist das Ergebnis der Operation.
- `char done` - Flag, das angibt, ob das Working-Set schon bearbeitet wurde

Ziel der Aufgabe ist es, eine simple parallel arbeitende Bildverarbeitung nachzustellen, wozu die Working-Sets verwendet werden sollen. Im ersten Aufgabenschritt soll mithilfe mehrerer Threads eine Verarbeitung von vorgegebenen Working-Sets realisiert werden, die eine Erhöhung der Bildhelligkeit pro Pixel simulieren.

Hinweis: Achtet bei System- und Bibliotheksaufrufen auf eine korrekte Fehlerbehandlung! Für diese Aufgabe wird ein Vorgaberahmen zur Verfügung gestellt:

http://ess.cs.tu-dortmund.de/Teaching/SS2017/BS/Downloads/vorgabe-A2.tar.gz

Die Dateien `aufgabe2.c`, `aufgabe2.h` und `aufgabe2_bonus.c` sollen (außer zu Testzwecken!) nicht modifiziert werden, da wir für die Korrektur die Dateien der Vorgabe verwenden. In `workingset.h` sind je nach Lösungsansatz keine oder nur geringfügige Änderungen notwendig.

a) Threads erzeugen, starten und beenden (3 Punkte)

⇒ `aufgabe2_a.c`

In der Datei `aufgabe2_a.c` findet ihr den vorgegebenen Rahmen, mit dem die Aufgabe bearbeitet werden soll. Durch diese Quelldatei wird eine Header-Datei eingebunden, welche die Größe des Datenbestandes und den Grad der Parallelisierung beeinflusst. Diese Datei solltet ihr nicht (oder nur testweise) modifizieren, da wir die vorgebene Datei zur Korrektur verwenden.

- Startet in der `main`-Funktion die Threads mithilfe der Funktion **`pthread_create(3)`**. Beachtet hier, die Anzahl, die durch `THREAD_NUM` definiert ist, zu verwenden. Die Threads sollen die Funktion run() ausführen.
- Die Funktion do_WorkingSet(WorkingSet *workingSet) soll für das angegebene Working-Set für alle Werte des Eingabefeldes (input-Wert aus Datenstruktur WorkingSet) die Operation (operation-Wert) aufrufen und den Rückgabewert in das Ausgabefeld (output-Wert) speichern. Abschließend sollte die Variable `done` auf 1 gesetzt werden, damit das Working-Set als bearbeitet markiert ist.
  Hinweis: Die Operation kann mit „`task->operation(parameter1, parameter2);`“ wie eine normale Funktion aufgerufen werden.
- Die Funktion `run()` soll die folgenden Schritte durchführen:
  - Überprüfen, ob das vorgegebene Working-Set schon bearbeitet wurde. Falls dies der Fall ist, kann mit dem nächsten Working-Set weitergemacht werden.
  - Falls das Working-Set noch nicht bearbeitet wurde, muss für dieses Working-Set die Funktion do_WorkingSet aufgerufen werden.
  - Anschließend müsst ihr die Variable `done` des Working-Sets auf 1 setzen, damit auch für andere Prozessoren sichtbar ist, dass dieses Working-Set bereits bearbeitet wurde.
  - Des Weiteren müssen der globale (`global_done`) und der thread-spezifische (`thread_done[threadNr]`) Zähler der abgearbeiten Working-Sets inkrementiert werden.
- Stellt mit **`pthread_join(3)`** sicher, dass das Programm erst beendet wird, wenn alle Threads ihre Aufgabe erledigt haben.

Die Threads sollen in dieser Teilaufgabe noch nicht synchronisiert werden - dass eine Datenkorruption auftritt ist hier normal! Außerdem kann es vorkommen, dass das Programm auch nach mehreren Sekunden nicht terminiert, in dem Fall könnt ihr das Programm mit Strg-C terminieren und neu starten. Die Implementierung soll in einer separaten Datei `aufgabe2_a.c` abgegeben werden. Ihr könnt euer Programm mit:

„`gcc -Wall -o aufgabe2_a aufgabe2_a.c aufgabe2.c -pthread`“ übersetzen.

b) Threads synchronisieren (3 Punkte)

⇒ `aufgabe2_b.c`

Übernehmt eure Lösung aus Aufgabenteil **a)** und speichert diese unter dem Dateinamen `aufgabe2_b.c` ab. Die parallele Datenverarbeitung soll nun gegen Race-Conditions abgesichert werden. Überlegt euch, welche Daten durch mehrere Threads verwendet werden und wie ihr sie durch die Verwendung von PThread-Mutexen absichern könnt. Verwendet dazu die Funktionen **`pthread_mutex_init(3)`**, **`pthread_mutex_lock(3)`**, **`pthread_mutex_unlock(3)`** und **`pthread_mutex_destroy(3)`**. Ändert gegebenenfalls die Datei `workingset.h` geeignet ab.

Stellt sicher, dass nach dem Absichern der Daten angezeigt wird „Hauptaufgabe ohne Korruption bearbeitet, prima!“.

Ihr könnt euer Programm mit:

„`gcc -Wall -o aufgabe2_b aufgabe2_b.c aufgabe2.c -pthread`“ übersetzen.

c) Parallele Nachbearbeitung mit Mutex-Conditionals (optional) ★

⇒ `aufgabe2_c.c`

Übernehmt die Änderungen eurer bisherigen Lösungen und fügt diese in die vorgegebene Datei `aufgabe2_c.c` ein!

Die vorherige Lösung soll nun so erweitert werden, dass eine parallele Nachbearbeitungsstufe für die Daten eingebaut wird, welche die zuvor berechneten Daten noch einmal glättet. Dazu besteht eine Abhängigkeit, dass ein Working-Set mit dem Index i des Arrays `blur_tasks` nicht gestartet werden darf, bevor das Working-Set mit dem Index i aus dem Array `tasks` abgeschlossen ist.

Dazu sollen weitere Threads (`THREAD_NUM` viele!) erzeugt werden, welche die Funktion post_process ausführen. Diese Funktion soll, ähnlich wie die Funktion run, Working-Sets abarbeiten. Überprüft in der Funktion post_process, ob der inputTask, von dem der outputTask abhängig ist, bereits berechnet wurde. Falls nicht, soll sich eurer Thread mithilfe der Funktion **`pthread_cond_wait(3)`** schlafen legen bis der inputTask bearbeitet wurde.

Falls der inputTask schon bearbeitet wurde (oder **`pthread_cond_wait(3)`** `zurückkehrt),` kann euer `outputTask` äquivalent zu der Verarbeitung in der Methode `run` abgearbeitet werden.

Erweitert dazu die Datenstruktur `WorkingSet` um eine **`pthread_cond_t`** Variable, mit der der wartende Thread der Nachberarbeitung das Fertigstellen eines `WorkingSets der Eingabedaten` signalisiert bekommt. Des Weiteren müsst ihr die Funktion `run` so erweitern, dass diese mit der Funktion **`pthread_cond_signal(3)`** das Fertigstellen eines `WorkingSets` signalisiert.

Ihr könnt euer Programm mit:

„`gcc -Wall -o aufgabe2_c aufgabe2_c.c aufgabe2_bonus.c -pthread`“ übersetzen.

Tipps zu den Programmieraufgaben:

- Bei der Kompilierung des Codes die Option **`-pthread`** für die Verwendung der posix-Threads nicht vergessen!
- Kommentiert euren Quellcode **ausführlich**, so dass wir bei Programmierfehlern noch Punkte vergeben können!
- Die Programme sollen dem C11- und POSIX-Standard entsprechen und sich mit dem gcc auf den Linux-Rechnern im IRB-Pool übersetzen lassen.
- Alternativ könnt ihr die Programme auch in C++ schreiben.

Abgabe bis Donnerstag 1. Juni 10:00 Uhr (Übungsgruppen in ungeraden Kalenderwochen) bzw. Dienstag 06. Juni 10:00 Uhr (Übungsgruppen in geraden Kalenderwochen)