Abgabefrist Übungsaufgabe 2

Die Lösung muss abhängig von der Tafelübung abgegeben werden, an der ihr teilnehmt:

- **W1** – Übung U2 in KW20 (13.05. - 16.05.): Abgabe bis **Mittwoch, den 22.05.2024 23:59**
- **W2** – Übung U2 in KW21 (21.05. - 23.05.): Abgabe bis **Montag, den 27.05.2023 12:00**

In darauffolgenden Tafelübungen werden teilweise einzelne abgegebene Lösungen besprochen, teilweise auch ein Lösungsvorschlag aus dem Tutorenteam.

Allgemeine Hinweise zu den BS-Übungen

- Es ist **nicht** mehr möglich, Einzelabgaben im AsSESS zu tätigen. Falls ihr (statt in einer Dreiergruppe) zu zweit oder zu viert abgeben möchtet, klärt dies bitte **vorher** mit eurem Übungsleiter! Der Lösungsweg und die Programmierung sind gemeinsam zu erarbeiten.
- 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 Zusatzaufgaben sind ein Stück schwerer als die „normalen“ Aufgaben und geben zusätzliche Punkte.
- Die Aufgaben sind über AsSESS (`https://sys-sideshow.cs.tu-dortmund.de/ASSESS/`) abzugeben. Dort gibt **ein** Gruppenmitglied die erforderlichen Dateien ab und nennt dabei die anderen beteiligten Gruppenmitglieder (Matrikelnummer, Vor- und Nachname erforderlich!). Namen und Anzahl der abzugebenden C-Quellcodedateien[2] variieren und stehen in der jeweiligen Aufgabenstellung; Theoriefragen sind grundsätzlich in der Datei `antworten.txt`[3] zu beantworten. Bis zum Abgabetermin kann eine Aufgabe beliebig oft abgegeben werden – es gilt die letzte, vor dem Abgabetermin vorgenommene Abgabe.
- Sobald eine Abgabe korrigiert wurde, kann das Ergebnis ebenfalls im AsSESS eingesehen werden.
- Ihre Programme müssen von gcc (also kein C++) mit der Option `-Wall` kompilierbar sein (als Referenzumgebung dienen die Poolrechner, wie empfehlen zusätzlich `-Werror`). Sollten Warnungen entstehen können ihnen dafür Punkte abgezogen werden. Programm die nicht übersetzt werden können werden nicht als Lösung akzeptiert und mit maximal einem Punkt bewertet.
- Nutzen sie in ihrem Quelltext eine sinnvolle Einrückung für Blöcke und vermeiden sie mehrere Anweisungen in einer Zeile. In Fällen von unleserlicher Formatierung können ihnen Punkte abgezogen werden.

Hinweis zu Übungen an Feiertagen (20.05.):

Bitte besuchen sie ausnahmsweise einen der anderen Termine. Sie können auch Termine in der jeweils anderen Woche besuchen.

Hinweis zu Übungen während der FVV (21.05. ab 14 Uhr):

Alle Übungen ab 14 Uhr fallen wegen der Fachschaftsvollversammlung aus. Bitte besuchen sie ebenfalls eine andere Übung.

[1] Da wir im Regelfall nicht unterscheiden können, wer von wem abgeschrieben hat, gilt das für Original **und** Plagiat.
[2] codiert in UTF-8
[3] reine Textdatei, codiert in UTF-8

Aufgabe 2: Thread-Synchronisation (10 Punkte)

2.1 Theoriefragen: Scheduling / Synchronisation (4 Punkte)

⇒ antworten.txt

Betrachtet die folgenden vier Prozesse, die direkt nacheinander in die *Ready*-Liste aufgenommen werden (Ankunftszeit 0). Zur Vereinfachung sei angenommen, dass die CPU- und E/A-Stöße pro Prozess immer gleich lang und dem Scheduler bekannt sind. Die Prozesse führen periodisch erst einen CPU-Stoß und dann einen E/A-Stoß durch (Zeiteinheit: 1 ms).

Prozess	CPU-Stoßlänge	E/A-Stoßlänge
A	3	6
B	7	2
C	2	6
D	5	4

1. Wendet auf die Prozesse A, B, C und D, die direkt nacheinander in dieser Reihenfolge laufbereit werden, das Scheduling-Verfahren *Virtual Round Robin (VRR)* aus der Vorlesung an. Der Scheduler gibt bei *VRR* jedem Prozess eine Zeitscheibe von 3 ms. Notiert die CPU- und E/A-Verteilung für die ersten 30 ms.
   Es gelten folgende Annahmen:

   - Prozesswechsel dauern 0 ms und können somit vernachlässigt werden.

   - Es können mehrere E/A-Vorgänge parallel ausgeführt werden.

   Stellt eure Ergebnisse wie im folgenden Beispiel dar, eine Spalte entspricht dabei 1 ms. Nutzt dazu in eurem Editor eine Monospace-Schriftart und keine Tabulatoren. „`C`“ = Prozess nutzt die CPU, „`E`“ = Prozess führt E/A-Operationen durch, „`-`“ = Prozess ist laufbereit:

   ```
   A: CCCEE---CCC
   B: ---CCEEE--- ...
   C: -----CCC---
   ```

2. Was ist der wesentliche Vorteil von *Virtual Round Robin* gegenüber *Round Robin*? Nennt zudem die Implementierungsunterschiede zwischen den beiden Verfahren, die diesen Vorteil bewirken.

3. Welchem Verfahren nähert sich *Round Robin* an, wenn eine zu lange Zeitscheibe gewählt wird?

Hinweis: Ihr könnt AnimOS[4] verwenden, um euch einen ersten Überblick über die verschiedenen Scheduling-Verfahren zu verschaffen. Während der Klausur habt ihr AnimOS aber nicht zur Verfügung! Ihr müsst die Lösungen daher auch ohne AnimOS erarbeiten können.

[4] https://ess.cs.uos.de/software/AnimOS/CPU-Scheduling/index.html → CPU-Scheduling

2.2 Programmierung: Campustüten (6 Punkte)

Wir möchten folgende Situation mit einem C-Programm simulieren:

An der Mensa werden insgesamt 40 Campustüten verteilt. 5 Helfer verteilen diese Tüten gleichzeitig an die Studierenden.

Die 5 Helfer sollen durch leichtgewichtige Prozesse (`POSIX-Threads`) nachgebildet werden. Eine Tüte auszugeben dauert in der Simulation jeweils 1 Sekunde, danach wird die Tütenanzahl um eins verringert.

a) Threads erzeugen, starten und beenden (2.5 Punkte)

⇒ `aufgabe2a.c`

- Legt eine Variable an, die die Anzahl der verbleibenden Campustüten enthält.
- Erstellt eine Funktion `void *verteilen(void* arg)`, in der die Arbeit eines Helfers durchgeführt wird.
  - **Vor** dem Verteilen einer Tüte soll die verbleibende Anzahl mit **printf(3)** ausgegeben werden.
  - Das Austeilen einer Tüte soll durch die Funktion **sleep(3)** und einer Dauer von 1 Sekunde simuliert werden.
  - **Nach** dem Austeilen einer Tüte soll der Zähler der verbleibenden Campustüten dekrementiert werden.
  - Die Funktion soll so lange Tüten austeilen, bis keine mehr übrig sind.
- Startet mit **pthread_create(3)** einen Thread für jeden Helfer, der Campustüten verteilen kann, sodass 5 Helfer-Threads gleichzeitig laufen.
- Die Threads aller Helfer sollen Tüten verteilen, bis keine mehr übrig sind und danach mit **pthread_exit(3)** terminieren.
- Am Ende des Programms, also wenn alle Helfer-Threads fertig sind, gebt die Anzahl der verbliebenen Tüten aus.
- Stellt dabei mit **pthread_join(3)** sicher, dass das Programm erst beendet wird, wenn alle Threads ihre Aufgaben erledigt haben. Beachtet dabei, dass ihr erst alle Threads startet und danach auf das Ende der Threads wartet.

Die Threads sollen in dieser Teilaufgabe noch nicht synchronisiert werden! Ihr müsst (u.a.) die Header-Datei `pthread.h` includieren und euer Programm mit der gcc-Option `-pthread` übersetzen, damit die richtigen Bibliotheken eingebunden werden.

Denkt bei allen Teilaufgaben daran, dass Systemaufrufe fehlschlagen können. Fangt diese Fehlerfälle ab – die Aufrufe melden dies über bestimmte Rückgabewerte, siehe die jeweiligen man-Pages, – gebt geeignete Fehlermeldungen aus (z.B. unter Zuhilfenahme von **perror(3)**), und beendet euer Programm danach ordnungsgemäß.

b) Analyse (2 Punkte)

⇒ `antworten.txt`

1. Welches Problem beobachtet ihr bei der Ausführung des entwickelten Programms?
2. Wie nennt man eine solche Situation?
3. Beschreibt schrittweise anhand von **zwei** parallel ausgeführten Threads, wie das beobachtete Problem entstehen kann.

c) Synchronisation (1.5 Punkte)

⇒ `aufgabe2c.c`

Erweitert euer Programm aus Aufgabenteil a), indem ihr mit einem Mutex den Zugriff auf den Campustüten-Zähler synchronisiert. Nutzt dazu **pthread_mutex_lock(3)** und **pthread_mutex_unlock(3)**. Initialisiert die Mutexvariable zuvor mit **pthread_mutex_init(3)** und entfernt sie mit **pthread_mutex_destroy(3)**, wenn sie nicht mehr gebraucht wird. Das Dekrementieren der des Zählers soll nun **vor** dem Herausgeben einer Campustüte geschehen.

Zusatzaufgabe 2: Semaphor-Synchronisation (2 Punkte)

⇒ `aufgabe2_extra.c`

Löst das Synchronisationsproblem mit einem POSIX-Semaphor statt mit einem Mutex. Verwendet hierzu eine geeignete Teilmenge der Funktionen **sem_init(3)** / **sem_destroy(3)** und **sem_post(3)** / **sem_wait(3)** / **sem_trywait(3)** (Übersicht: **sem_overview(7)**).

Tipps zu den Programmieraufgaben:

- Kommentiert euren Quellcode ausführlich, so dass wir auch bei Programmierfehlern im Zweifelsfall noch Punkte vergeben können!
- Denkt daran, dass viele Systemaufrufe fehlschlagen können! Fangt diese Fehlerfälle ab (die Aufrufe melden dies über bestimmte Rückgabewerte, siehe die jeweiligen man-Pages), gebt geeignete Fehlermeldungen aus (z.B. unter Zuhilfenahme von **perror(3)**), und beendet euer Programm danach ordnungsgemäß.
- Die Programme sollen sich mit dem gcc auf den Linux-Rechnern im IRB-Pool übersetzen lassen. Der Compiler ist mit den folgenden Parametern aufzurufen:
  `gcc -Wall -D_GNU_SOURCE -pthread`
  Weitere (nicht zwingend zu verwendende) nützliche Compilerflags sind: `-ansi -Wpedantic -Werror -D_POSIX_SOURCE`
- Achtet darauf, dass sich der Programmcode ohne Warnungen übersetzen lässt, z.B. durch Nutzung von `-Werror`.