Abgabefrist Übungsaufgabe 2

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

- **W1** – eigene Übung U2 in KW 19 (09.-11.05.): Abgabe bis **Donnerstag, den 19.05.2022 12:00 Uhr**
- **W2** – eigene Übung U2 in KW 20 (16.-18.05.): Abgabe bis **Montag, den 23.05.2022 12:00 Uhr**

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 Gruppenmitglieder sollten nach Möglichkeit gemeinsam an der gleichen Tafelübung teilnehmen. Es sind aber auch Abgaben mit Studierenden aus verschiedenen Übungsgruppen zulässig. Es gilt dabei immer die früheste Abgabefrist. Die Lösung wird jeweils komplett bewertet und den Gruppenmitgliedern gleichermaßen angerechnet.
- 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://ess.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.

[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)

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	7	2
B	2	2
C	2	5

1. Wendet auf die Prozesse A, B und C, 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---
   ```

   Hinweis: Prozesse werden erst auf die Ready-/Vorzugs-Liste einsortiert, wenn sie ihren E/A-Stoß beendet haben. Bei den Ready-/Vorzugs-Listen handelt es sich um FiFo Datenstrukturen.

2. Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?

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

4. Was macht Shortest Process Next (SPN) anders als Round Robin bzw. Virtual Round Robin? Welche Herausforderungen bzw. Problemen bestehen bei Shortest Process Next typischerweise?

5. Fasst die zwei wichtigsten Unterschiede zwischen *Semaphoren* und *Mutexen* zusammen.

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/index.html → CPU-Scheduling

Programmierung: Eröffnungsfeier (6 Punkte)

Dieses Jahr soll ein Restaurant mit dem Namen 'Zur BS-Klausur' auf dem Campus eröffnet werden. Es sind bereits 5 Servicekräfte für das Lokal angestellt. Zur Eröffnungsfeier sind insgesamt 64 Gäste geladen, die vom gesamten Servicepersonal gleichzeitig bedient werden sollen. Leider hat die Küche noch nicht die Kapazität, die der Betrieb des Restaurants benötigen würde. Daher muss das Servicepersonal für die Eröffnungsfeier alle Bestellungen einzeln an die Küche weiterleiten, in der dann mit jeweils 5 mal Umrühren pro Gast eine Tütensuppe gekocht wird. Wenn alle Gäste bedient wurden, darf das Servicepersonal in den wohlverdienten Feierabend gehen. Zu beginn der Arbeitszeit trödeln alle Servicekräfte ein wenig und beginnen erst dann mit der eigentlichen Arbeit.

Die Eröffnungsfeier soll durch leichtgewichtige Prozesse (POSIX-Threads) nachgebildet werden. Hierfür ist ein Coderahmen vorgegeben[5].

Die Servicekräfte werden durch die Funktion **bedienen** simuliert. Darin warten sie ein bisschen (Funktion **troedeln**) und veranlassen dann das Kochen (Funktion **kochen**) für die von ihnen zu bedienenden Gäste. Wer bedient wird, wird durch ein Struct als Argument übergeben. Dieses soll durch die Funktion **bedienliste_fuer** jeweils pro Servicekraft initialisiert werden. Dabei wird eine aufsteigend für die Servicekraft vergebene Nummer zum Ausrechnen der zu bedienenden Gäste verwendet und jedem Gast eine weitere individuelle Nummer zugeordnet. Die gobalen Konstanten **ANZ_GAESTE**, **ANZ_SERVICEK**, **ANZ_UMRUEHREN** und **TROEDEL_DAUER** geben vor, wie viele Gäste und Servicekräfte es gibt, bzw. wie viel umgerührt oder getrödelt wird.

a) Threads erzeugen, starten und beenden (4 Punkte)

⇒ `aufgabe2a.c`

- Ergänzt die **main**-Funktion, sodass für alle Servicekräfte Threads mit **pthread_create(3)** gestartet werden, welche die Funktion **bedienen** mit einem passend durch die Funktion **bedienliste_fuer** initialisierten Argument abarbeiten.
- Stellt dabei mit **pthread_join(3)** sicher, dass das Programm erst beendet wird, wenn alle Threads ihre Aufgabe erledigt haben.
- Fangt mögliche Fehler bei der Ausführung von **pthread_create(3)** und **pthread_join(3)** ab und behandelt sie durch eine Ausgabe und beenden des Programms.

Die Threads sollen in dieser Teilaufgabe noch nicht synchronisiert werden! Übersetzt euer Programm mit den folgenden gcc-Flags:

```
-Wall -std=c11 -Wpedantic -O0 -D _POSIX_C_SOURCE=200809L -pthread
```

b) Analyse (2 Punkte)

⇒ `antworten.txt`

1. Welches Problem beobachtet ihr bei der Ausführung des entwickelten Programms? Wie nennt man eine solche Situation und welche Ressource wird geteilt?
2. Warum muss das Argument der Threads den Aufruf von **pthread_create(3)** überleben?
3. Warum ist die aus der Übung bekannte Funktion **pthread_self(3)** nicht geeignet, um die zu bedienenden Gäste mittels **bedienliste_fuer** innerhalb der **bedienen** Funktion zu ermitteln?

[5] https://moodle.tu-dortmund.de/pluginfile.php/2006685/mod_label/intro/vorgabe-a2.tar.bz2

c) Synchronisation (2 Punkte)

⇒ `aufgabe2c.c`

Erweitert euer Programm aus Aufgabenteil a), indem ihr das Kochen für einzelne Gäste mit einem Mutex synchronisiert und so sicherstellt, dass die Küche nicht mehr durcheinander arbeitet.

- Deklariert, initialisiert und entfernt eine Mutexvariable mit **pthread_mutex_init(3)** und **pthread_mutex_destroy(3)**.
- Nutzt **pthread_mutex_lock(3)** und **pthread_mutex_unlock(3)** an einer passenden Stelle innerhalb der vorgegebenen Funktionen.
- Fangt mögliche Fehler bei der Ausführung der Mutex-Funktionen ab und behandelt sie durch eine Ausgabe und beenden des Programms.

d) Zusatzaufgabe 2: Letzter bedienter Gast (2 Sonderpunkte)

⇒ `aufgabe2d.c`

Der letzte bediente Gast soll mit einem Gratisgetränk für sein Durchhaltevermögen belohnt werden. Ergänzt eure Lösung aus Aufgabenteil c), sodass am Ende der Simulation ausgegeben wird, welche individuelle Nummer dieser Gast hatte. Das Bestimmen des letzten Gastes erfolgt durch die Servicekräfte und soll nicht das Bedienen blockieren.

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, 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 -std=c11 -Wpedantic -O0 -D _POSIX_C_SOURCE=200809L -pthread`
- Achtet darauf, dass sich der Programmcode ohne Warnungen übersetzen lässt, z.B. durch Nutzung von `-Werror`.