Abgabefrist Übungsaufgabe 2

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

- **W1** – eigene Übung U2 am 18./19.05.: Abgabe bis **Donnerstag, den 27.05.2021 08:00**
- **W2** – eigene Übung U2 am 25./26.05.: Abgabe bis **Dienstag, den 01.06.2021 08:00**

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://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	7	5
D	2	5

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?
4. Fasst als kleinen Vorgriff auf die Zusatzaufgabe 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.tu-dortmund.de/Software/AnimOS/` → CPU-Scheduling

Programmierung: Impfzentrum (6 Punkte)

In einem COVID-19-Impfzentrum können insgesamt 50 Impftermine vergeben werden. Es gibt 6 Hotlines, über die gleichzeitig Impftermine reserviert werden können. Zur Vereinfachung werden nicht wirklich konkrete Termine vergeben, sondern nur die verbleibenden Termine gezählt. Auch gibt es keine echten Anrufer, sondern die Hotlines vergeben einfach einen Termin nach dem anderen.

Die 6 Hotlines sollen durch leichtgewichtige Prozesse (POSIX-Threads) nachgebildet werden. Eine Terminreservierung dauert in der Simulation jeweils 1 Sekunde, danach wird der Termin-Zähler um eins verringert.

a) Threads erzeugen, starten und beenden (2 Punkte)

⇒ `aufgabe2a.c`

- Legt eine Variable an, die die Anzahl der verbleibenden Impftermine enthält.
- Startet mit **pthread_create(3)** einen Thread für jede Hotline, die Termine vergeben kann.
- Die Threads aller Hotlines sollen terminieren, wenn keine Impftermine mehr übrig sind.
- Dabei sollen die Threads zunächst unsynchronisiert auf die Impftermin-Variable zugreifen und **nach** jeder Terminvergabe den Impftermin-Zähler dekrementieren.
- Gebt **vor** jeder Terminvergabe und am Ende des Programms die Anzahl der verbleibenden Impftermine aus.
- Stellt dabei 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! 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 (2 Punkte)

⇒ `aufgabe2c.c`

Erweitert euer Programm aus Aufgabenteil a), indem ihr mit einem Mutex den Zugriff auf den Impftermin-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 Impftermin-Anzahl soll nun **vor** der Terminvergabe stattfinden.

d) Zusatzaufgabe 2: Semaphor-Synchronisation (2 Sonderpunkte)

⇒ `aufgabe2d.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`.