Weitere Hinweise zu den BS-Übungen

- Ab jetzt ist es **nicht** mehr möglich, Einzelabgaben in AsSESS zu tätigen. Falls ihr (statt in einer Dreiergruppe) zu zweit oder zu viert abgeben möchtet, klärt das bitte **vorher** mit eurem Übungsleiter!
- 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.

Aufgabe 2: Thread-Synchronisation (10 Punkte)

1 Scheduling (2 Punkte)

⇒ antworten.txt

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

Prozess-ID	CPU-Stoßlänge	E/A-Stoßlänge
A	3	2
B	2	3
C	6	6

Ihr könnt AnimOS (`https://ess.cs.tu-dortmund.de/Software/AnimOS/` → CPU-Scheduling) verwenden, um euch einen ersten Überblick über die verschiedenen Schedulingverfahren zu machen.

Achtung: In der Klausur habt ihr AnimOS nicht zur Verfügung. Ihr solltet die Lösungen daher auch ohne AnimOS erarbeiten können.

1. Wendet auf die Prozesse A, B und C das *Round-Robin*-Scheduling-Verfahren aus der Vorlesung an. Der Scheduler gibt dabei jedem Prozess eine Zeitscheibe von 3 ms. Notiert die CPU- und E/A-Verteilung für die ersten 30 ms. (Annahmen: Prozesswechsel sind zu vernachlässigen; es können mehrere E/A-Vorgänge parallel ausgeführt werden.)
   Stellt eure Ergebnisse wie in der folgenden ASCII-Zeichnung dar (die ersten 11 ms sind vorausgefüllt), eine Spalte entspricht dabei 1 ms. Nutzt dazu in eurem Editor eine Monospace-Schriftart und keine Tabulatoren.
   ```
   +---+-----------     +
   | A |CCCEE---CCC     | C: Prozess nutzt die CPU
   | B |---CCEEE--- ... | E: Prozess führt E/A-Operationen durch
   | C |-----CCC---     | -: Prozess ist laufbereit
   +---+-----------     +
   ```
2. Welches Problem birgt *Round Robin* als Scheduling-Verfahren?

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

2 Primzahlserver (8 Punkte)

In der Vorgabe[2] befinden sich zwei Dateien, die einen Primzahl-Server (der kontinuierlich Primzahlen *produziert*) und einen Client (der diese *konsumiert*) in Form eines mehrfädigen Programms implementieren:

prime.c Die Primzahl-Anwendung, die zwei POSIX-Threads erzeugt. Einer davon (**primserv**) generiert Primzahlen und schreibt diese in einen geteilten Speicher. Ein weiterer POSIX-Thread (**primeat**) konsumiert die Primzahlen.

Makefile Eine Steuerdatei für GNU Make, mit dem ihr bequem das Programm **prime** erstellen könnt.

2.1 Beobachten und Beschreiben (1 Punkt)

Übersetzt das Programm **prime**, indem ihr GNU Make (Kommando `make`) ohne weitere Parameter aufruft. Startet nun das vorher compilierten Programm **prime** (`./prime`). Was beobachtet ihr? Wieso „fehlen“ in den Ausgaben des Threads **primeat** sehr viele der Primzahlen, die **primserv** erzeugt hat? (⇒ antworten.txt)

2.2 Synchronisation mit POSIX-Semaphoren (4 Punkte)

Das Produzieren und Konsumieren der Primzahlen soll nun synchronisiert werden, so dass keine Zahlen mehr verloren gehen. Das Szenario ist ein klassisches Erzeuger-/Verbraucher-Problem: Da der gemeinsam genutzte Puffer (die int-Variable, über die die produzierte Primzahl zu **primeat** gelangt) nur *ein* Element aufnehmen kann, reichen zur Synchronisation zwei Semaphore aus (wie in der Übung besprochen: jeweils eine zum Zählen der **noch freien** bzw. der **schon belegten** Elemente des Puffers).

Zunächst müsst ihr in prime.c eine geeigneten Anzahl an Semaphoren in Form von globalen Variablen erstellen und mit Hilfe der Funktion **sem_init(3)** passend initialisieren. Nutzt anschließend die Funktionen **sem_wait(3)** und **sem_post(3)**, um die beiden Threads zu synchronisieren. Vergesst nicht, am Ende die Semaphoren wieder mit **sem_destroy(3)** aufzuräumen.

Testet nun die so synchronisierte Fassung der Programme, wie bereits in Teilaufgabe 2.1 beschrieben.

2.3 Synchronisationsmuster (1 Punkt)

Der Primzahlerzeuger **primserv** möchte gerne, dass jede seiner erzeugte Primzahlen auch konsumiert wird. Es sollen keine Primzahlen verloren gehen. Da der Puffer nur ein Element aufnehmen kann, müsste es eigentlich genügen eine Semaphore zu verwenden, die einen gegenseitigen Ausschluss gewährleistet. Warum ist es dennoch erforderlich zwei Semaphoren zur Synchronisation der beiden Threads **primeat** und **primserv** zu verwenden? (⇒ antworten.txt)

2.4 Größerer Puffer (2 Punkte)

Damit Erzeuger und Verbraucher stärker entkoppelt werden, wäre es wünschenswert, den verwendeten Puffer auf *mehrere Elemente* zu vergrößern. Beschreibt, welche Modifikationen an der gemeinsamen Datenstruktur und der Synchronisation notwendig sind, um einen solche Puffer verwenden zu können. (⇒ antworten.txt)

[2] `https://ess.cs.tu-dortmund.de/Teaching/SS2019/BS/Downloads/vorgabe-A2.tar.gz`

2.5 Zusatzaufgabe: Tatsächlich größerer Puffer (2 Punkte)

Implementiert die in Aufgabenteil 2.4 skizzierten Änderungen und verwendet dafür einen Ringpuffer. Macht die Modifikationen in prime.c mit den Präprozessor-Direktiven **`#ifdef`** `RINGBUFFER`, (ggf. **`#else`**) und **`#endif`** so konfigurierbar, dass sie beim Aufruf von `make prime` nicht mitübersetzt werden, sondern nur mit `make prime-ringbuffer`.

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. Es ist das mitgelieferte Makefile (Kommando `make`) zu verwenden, oder der Compiler mit den folgenden Parametern aufzurufen:
  `gcc -Wall -D_GNU_SOURCE -o prime prime.c -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`.
- Alternativ kann auch der GNU C++-Compiler (*g++*) verwendet werden.

Abgabe bis Donnerstag, 16. Mai 10:00 Uhr (Übungsgruppen in ungeraden Kalenderwochen) bzw. Dienstag, 21. Mai 10:00 Uhr (Übungsgruppen in geraden Kalenderwochen)