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 Zusatzaufgaben sind ein Stück schwerer als die „normalen“ Aufgaben und geben zusätzliche Punkte.

Aufgabe 3: Deadlock (10 Punkte)

Ziel der Aufgabe ist es, das Entstehen, Erkennen, Auflösen und Vermeiden von Deadlocks anhand eines konkreten Szenarios kennenzulernen.

Zu den untenstehenden Aufgaben existiert ein vorimplementiertes Programm, welches aus mehreren Dateien besteht. Die Dateien `main.c und main.h` bilden dabei die „äußere“ Struktur des Programmes, die den allgemeinen Ablauf vorgeben (dürfen nicht modifiziert werden). Die Datei `3_b.c` ist eine Implementierung des unten beschriebenen Arbeitsablauf im Szenario, die ihr für Aufgabenteil b) ausführen sollt (ohne sie zu ändern). `3_c.c` ist inhaltlich identisch mit `3_b.c`, soll von euch jedoch für Aufgabenteil c) angepasst werden. Diese Vorgabe ist von der Veranstaltungswebseite herunterzuladen, zu entpacken und zu vervollständigen! Der Ordner `vorgabe-A3.tar.gz` lässt sich mittels `tar -xzvf vorgabe-A3.tar.gz` entpacken.

Beachtet bitte, dass die Theoriefragen sich nicht wie sonst üblich nur zu Beginn des Aufgabenblattes befinden, sondern mit der Programmieraufgabe vermischt sind. Auch wenn ihr Schwierigkeiten beim Lösen der Programmieraufgabe haben solltet, lassen sich Fragen beantworten.

Theoriefragen: Verklemmungen (3 + 3 Punkte)

Betrachtet folgendes Szenario:

Zwei Mitarbeiter teilen sich die Arbeit eine Klausur zu korrigieren. Es gibt einen Karton mit den Klausuren und eine Liste für die Noten, welche an unterschiedlichen Orten aufbewahrt werden. Ein Mitarbeiter kann nur dann die Klausuren korrigieren, wenn er den Karton mit allen Klausuren und die Liste hat, weil er die Note für jede korrigierte Klausur sofort in die Liste eintragen muss. Da jeder Mitarbeiter auch noch andere Arbeit zu erledigen hat, arbeitet er immer nur eine gewisse Zeit an der Klausurkorrektur. Danach gibt er die Klausuren und die Liste wieder zurück, damit der nächste Mitarbeiter an den Klausuren arbeiten kann. Da die Mitarbeiter die Korrektur möglichst schnell fertigstellen wollen, warten sie auf die Klausuren oder die Liste, falls diese gerade nicht da sind.

Die Mitarbeiter in diesem Szenario sollen nun keine Ahnung von Deadlocks haben. Deswegen haben sie nicht geregelt, in welcher Reihenfolge die notwendigen Dinge geholt werden müssen. Der eine Mitarbeiter holt sich immer zuerst die Klausuren und danach die Liste, der andere macht es genau andersherum.

a) Vorbedingungen für Verklemmungen (1 + 1,5 + 0,5 Punkte)

1. Lest euch die obige Beschreibung der Ausgangssituation durch. Was ist sind die Betriebsmittel? Sind sie konsumierbar oder wiederverwendbar?

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

2. Damit Verklemmungen entstehen können, müssen bestimmte Bedingungen erfüllt sein: *mutual exclusion*, *hold and wait* und *no preemption*. Beschreibt, wodurch diese Bedingungen beim Korrigieren erfüllt werden.
   Damit wirklich eine Verklemmung entsteht, muss zur Laufzeit noch eine weitere Bedingung eintreten. Nennt diese und beschreibt kurz allgemein (nicht Szenario-bezogen), wie diese eintreten kann.
3. Wieso kommt es in dem Szenario nicht zu einem *Livelock*?

⇒ `antworten.txt`

b) Das Szenario (3 Punkte) In dieser Aufgabe sollt ihr die Implementierung des Szenarios kennenlernen. Zur Durchführung der Korrektur sind zwei Dinge nötig: Der Karton mit den Klausuren und die Liste für die Noten. Diese sind mit den Semaphoren *„klausuren“* und *„liste“* implementiert. Um das Programm zu vereinfachen werden die Klausuren und die Liste nicht wirklich geholt und zurückgebracht, sondern es werden nur die beiden Semaphoren *„klausuren“* und *„liste“* belegt und freigegeben.

Vorgegeben ist (siehe `vorgabe-A3.tar.gz`):

- Die Funktion `main()` (in `main.c`), welche das Programm startet, mit Signalhandler für den Programmabbruch (der Signalhandler soll in dieser Aufgabe von euch nicht weiter betrachtet werden).
- Zwei fertige Mitarbeiter-Threads, die gestartet werden und die Funktionen `mitarbeiter_a()` bzw. `mitarbeiter_b()` ausführen (in `3_b.c`, siehe auch Tabelle unten). Dabei nehmen sie abwechselnd je einen der folgenden Zustände ein, der in globalen Variablen vermerkt wird: *„andere_arbeit“*, *„hole_klausuren“*, *„hole_liste“* und *„korrigieren“*.
- Globale Variablen für die Semaphoren *„klausuren“* und *„liste“*.
- Die fertigen Funktionen `andere_arbeit_ausfuehren()`, `deadlock_erkennung()` (wird für Aufgabenteil b) und die Zusatzaufgabe benötigt), und die Funktion `programm_abbruch()`, wo die Semaphoren und Threads aufgeräumt werden (soll in dieser Aufgabe von euch nicht weiter betrachtet werden).

Durchzuführen ist:

- Verschafft euch einen Überblick über die Vorgabe (ohne Punkte).
- Führt das Programm testweise aus (in der Version `3_b.c`) und beschreibt detailliert, was passiert. Geht dabei auf eure Antworten aus dem ersten Teil der Theoriefragen ein. Beschreibt dabei, nach wie vielen Sekunden ein Deadlock eintritt.
- Erklärt, wie und wieso es zu einem Deadlock kommt, insbesondere im Bezug auf die Laufzeitbedingung (siehe oben). Was müssen die Mitarbeiter in welcher Reihenfolge tun, damit es zu einem Deadlock kommt?
- Um ein Deadlock zu erkennen, soll der Professor regelmäßig nach seinen Mitarbeitern schauen und feststellen, ob sich diese in einem Deadlock befinden. Beschreibt in eigenen Worten, wie der Professor feststellen könnte, dass ein Deadlock vorliegt.

⇒ `antworten.txt`

Arbeitsablauf der Mitarbeiter-Threads:

Schritt	Mitarbeiter A	Mitarbeiter B
1	Andere Arbeit ausführen.	
2	**Klausuren holen:** Status auf *„hole_klausuren“* setzen; Semaphore *„klausuren“* belegen; Dauer: 4 s.	**Liste holen:** Status auf *„hole_liste“* setzen; Semaphore *„liste“* belegen; Dauer: 3 s.
3	**Liste holen:** Status auf *„hole_liste“* setzen; Semaphore *„liste“* belegen; Dauer: 3 s.	**Klausuren holen:** Status auf *„hole_klausuren“* setzen; Semaphore *„klausuren“* belegen; Dauer: 4 s.
4	**Korrigieren:** Status auf *„korrigieren“* setzen; Dauer: 5 s.	
5	Semaphore *„liste“* freigeben.	Semaphore *„klausuren“* freigeben.
6	Semaphore *„klausuren“* freigeben.	Semaphore *„liste“* freigeben.
7	Weiter mit Schritt 1	

Ihr könnt das Programm mit dem Makefile übersetzen; eine Anleitung zur Nutzung von Makefiles findet ihr auf Übungsblatt 2 (Kommando `make b` aufrufen).

Theoriefragen und Programmierung: Deadlocks erkennen, auflösen und vermeiden (2 + 1 + 1 Punkte)

c) Deadlock erkennen In der Vorgabe ist der Funktionsaufruf `deadlock_erkennung()` sowie `check_for_deadlocks()` vorgegeben. Diese Funktionen werden bereits im Elternprozess gestartet, und `check_for_deadlocks()` wird alle 12 Sekunden aktiv. Implementiert in `check_for_deadlocks()` eine Erkennung, ob es zu einem Deadlock gekommen ist. Sobald ein Deadlock erkannt wurde, soll eine entsprechende Nachricht auf der Konsole ausgegeben werden (**printf(3)**). Orientiert euch dabei an den bereits vorgegebenen Ausgaben, zum Beispiel aus den Mitarbeiter-Threads. Beachtet dabei, dass ihr den Zugriff auf den gemeinsam genutzten Speicher (also der Status der Mitarbeiter) mit der weiteren Semaphore `status` überall da, wo auf den Status der Mitarbeiter zugegriffen wird, schützen müsst.

⇒ `3_c.c`

d) Deadlock auflösen Beschreibt eine Möglichkeit, wie sich Verklemmungen generell auflösen lassen.

Erklärt, wie sich eure Erkenntnisse auf den Professor und die blockierten Mitarbeiter in dem Szenario übertragen lassen.

⇒ `antworten.txt`

e) Deadlocks vermeiden Deadlocks lassen sich in diesem Szenario auch *vermeiden*. Beschreibt, wie man den Arbeitsablauf der beiden Mitarbeiter-Threads ändern müsste, um Deadlocks prinzipiell zu vermeiden.

⇒ `antworten.txt`

Zusatzaufgabe: Deadlock auflösen (2 Punkte)

Implementiert eure Überlegungen aus Aufgabenteil d). Es ist hier explizit nicht gefragt, die Überlegungen aus Aufgabenteil e) zu implementieren, da dann kein Deadlock mehr eintreten kann.

⇒ `3_extended.c`

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 -pthread -std=c11 -Wall -o ziel datei.c`
  Weitere (nicht zwingend zu verwendende) nützliche Compilerflags sind: `-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 spätestens Donnerstag, den 30. Mai 10:00 (Übungen in ungerader Kalenderwoche) bzw. Dienstag, den 4. Juni 10:00 (Übungen in gerader Kalenderwoche).