Weitere Hinweise zu den Betriebssysteme-Übungen

- 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 und Auflösen von Deadlocks praktisch umzusetzen.

ACHTUNG: Zur Programmieraufgabe existiert eine Vorgabe in Form von C-Dateien mit vorimplementierten Code-Rümpfen, die ihr erweitern sollt. Diese Vorgabe ist von der Veranstaltungswebseite herunterzuladen, zu entpacken und zu vervollständigen. Die Datei `vorgabe-A3.tar.gz` lässt sich mittels `tar -xzvf vorgabe-A3.tar.gz` entpacken.

In der Küche der Mensa arbeiten einige Köche, die durch Kombination von verschiedenen Zutaten versuchen, neue leckere Gerichte für die Mensa zu finden. Jeder Koch hat eine eigene Küchenzeile und kann sich vom zentralen Kühlschrank Zutatenkisten holen.
Ein Koch kann nur eine Zutatenkiste gleichzeitig holen und verwendet diese für sein Gericht, bevor er die nächste holt. Nachdem er sein Gericht fertig gestellt hat, bringt der Koch alle Zutatenkisten wieder zurück zum Kühlschrank. Er wird vor Fertigstellung seines Gerichts keine Kisten abgeben, die er bereits geholt hat.
Alle Köche müssen sich aus dem zentralen Kühlschrank bedienen. Die Anzahl an Zutatenkisten ist aber begrenzt, weswegen es vorkommen kann, dass eine Zutat nicht direkt zur Verfügung steht. Die Köche warten dann, bis ein anderer Koch eine passende Kiste zurückbringt.
Wir gehen davon aus, dass in den Gerichten nur geringe Mengen der einzelnen Zutaten benötigt werden, da die Köche jeweils nur eine Portion des Gerichts kochen und die Mensa einen großen Vorrat an Zutaten besitzt. Leere Zutatenkisten gibt es in diesem Szenario daher nicht.

Das oben beschriebene Szenario soll in den folgenden Aufgaben mithilfe von Semaphoren und POSIX-Threads implementiert werden. Die Implementierung soll in mehreren Schritten erfolgen, die in separaten Dateien (`a3_X.c`) abgegeben werden. Auf der Veranstaltungswebseite findet ihr zu dieser Aufgabe eine Vorgabe (`vorgabe-A3.tar.gz`). Sie enthält eine Reihe von C-Dateien, die bereits einen Rahmen für euer Programm zur Verfügung stellen und bildet somit das Grundgerüst für das beschriebene Szenario. Zudem findet ihr ein Makefile in der Vorgabe, sodass ihr das gesamte Projekt schnell mit den Befehlen `make` und `make clean` verwalten könnt. Beachtet, dass das Projekt erst nach Implementierung aller Teilaufgaben vollständig kompilierbar ist.

Theoriefragen (1+2+2 Punkte)

In der Mensaküche können leider Verklemmungen entstehen. Diese wollen wir uns hier näher anschauen.

1. Lest euch die obige Beschreibung der Ausgangssituation durch. Was ist das Betriebsmittel? Ist es konsumierbar oder wiederverwendbar? (1P)
2. Damit Verklemmungen entstehen können, müssen bestimmte Vorbedingungen erfüllt sein: *mutual exclusion*, *hold and wait* und *no preemption*. Beschreibt, wodurch diese Bedingungen in der Mensa erfüllt werden.

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

Damit wirklich eine Verklemmung entsteht, muss zur Laufzeit noch *circular wait* eintreten. Beschreibt kurz eine Beispielsituation in der Küche, bei der eine Verklemmung auftritt. (2P)

3. Zur Verklemmungsvorbeugung wird in Teil b) der Programmieraufgabe eine der Bedingungen für Verklemmungen entkräftet. Um welche Bedingung handelt es sich? Erklärt zudem anhand des Szenarios, warum nun keine Verklemmungen mehr auftreten können. Geht außerdem darauf ein, wieso diese Variante der Verklemmungsvorbeugung ineffizient ist. (2P)

⇒ `antworten.txt`

Programmieraufgaben (5 Punkte)

Wir wollen nun die Mensaküche durch ein C-Programm darstellen. Dazu stellt die Vorgabe folgende Hilfsmittel bereit (Schnittstellen, Konstanten und globale Variablen siehe `vorgabe.h`):

- Eine Zutat wird durch die Struktur `struct ingredient` mit dem Typalias `ingredient_t` dargestellt. Sie enthält den Namen der Zutat (`name`), die Verarbeitungsdauer in Sekunden (`time_needed`), die Anzahl der insgesamt vorhandenen Kisten mit dieser Zutat (`boxes`) und eine Semaphore, die die aktuell im Kühlschrank verfügbare Kistenzahl angibt (`sem`).
- Die Zutaten befinden sich im globalen Array `ingredients`; ihre Anzahl ist durch das Makro `INGREDIENT_NUM` gegeben.
- Zu Beginn sind alle Kisten im Kühlschrank. Das Holen einer Zutatenkiste wird durch Belegen der Semaphore simuliert, das Zurückbringen durch Freigabe.
- Gerichte werden durch die Funktion `get_meal` generiert und als Array von Zeigern auf die enthaltenen Zutaten dargestellt. Rezepte haben die Länge `MEAL_SIZE` und enthalten jede Zutat höchstens einmal.
- Jeder Koch wird durch einen Thread dargestellt. Das Makro `CHEF_NUM` gibt die Anzahl an Köchen an.

a) Fäden implementieren (3 Punkte)

Implementiert die Funktion `work()`, die von den Koch-Threads ausgeführt wird. Der Ablauf soll folgender sein:

- Gericht generieren: in `work()` ein Array `ingredient_t *meal[MEAL_SIZE]` anlegen, dann `get_meal(meal);` aufrufen.
- Für jede der enthaltenen Zutaten:
  - Die Zutatenkiste aus dem Vorrat holen (Semaphore `meal[i]->sem` belegen).
  - Zutat verarbeiten (Warten der entsprechenden Zeit `meal[i]->time_needed`).
- Sobald alle Zutaten aus dem Gericht verarbeitet wurden: Alle Zutatenkisten wieder zurückbringen (Semaphore freigeben).
- Dieser Ablauf soll `MEAL_PER_CHEF` mal durchgeführt werden, bevor der Koch nach Hause geht (Thread beenden).

⇒ `a3_a.c`

Ihr könnt das Programm mit `make a` kompilieren und anschließend mit `./a3_a` ausführen.

b) Verklemmungsvorbeugung (2 Punkte)

Um Deadlocks vorzubeugen, ändern die Köche nun ihre Vorgehensweise:

- Zunächst wird der *gesamte* Kühlschrank reserviert (Mutex `lock` aus `vorgabe.h`).
- Dann holt der Koch sich *alle benötigten Kisten* direkt nacheinander. Wenn eine Kiste nicht verfügbar ist, wartet er ganz normal.
- Dann gibt er den Kühlschrank wieder frei und fängt erst dann an, die Zutaten zu verarbeiten.
- Schließlich gibt er die Kisten zurück. Hierfür muss der Kühlschrank nicht reserviert werden.

Benutzt die globale Mutex-Variable `lock`, um das Reservieren des Kühlschranks zu simulieren.

Kopiert für diese Aufgabe eure Lösung für Aufgabe b) und passt sie an.

⇒ `a3_b.c`

Ihr könnt das Programm mit `make b` kompilieren. Die ausführbare Datei wird als `a3_b` abgelegt.

c) Zusatzaufgabe: Verklemmungsauflösung (2 Punkte)

Die Köche probieren diesmal eine andere Methode, um das Problem zu beheben: Sie warten immer nur eine bestimmte Zeit auf eine Zutatenkiste. Wenn diese nicht rechtzeitig verfügbar wird, wird die gesamte Küche zurückgesetzt.

Kopiert dazu eure Lösung für Aufgabe b) erneut und passt sie so an, dass die Köche nur so lange warten, wie ein maximal zeitintensives Gericht (max. Anzahl an Zutaten mal längste Verarbeitungsdauer der Zutaten) brauchen würde.

Nach Ablauf der Zeit soll die globale Variable `restart = 1` gesetzt werden (dadurch startet die main-Funktion die Threads neu, nachdem alle beendet sind), alle anderen Threads abgebrochen und schließlich auch der aktuelle Koch-Thread beendet werden.

Nehmt dazu ggf. die Funktionen **sem_timedwait(3)**, **clock_gettime(3)**, **pthread_cancel(3)**, **pthread_self(3)** und **pthread_equal(3)** zu Hilfe.

⇒ `a3_c.c`

Ihr könnt das Programm mit `make c` kompilieren. Die ausführbare Datei wird als `a3_c` abgelegt.

Tipps zu den Programmieraufgaben:

- Kommentiert euren Quellcode ausführlich, so dass wir bei Programmierfehlern 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`
- Alternativ kann auch der GNU C++-Compiler (g++) verwendet werden.

Abgabe bis Donnerstag 18. Juni 8:00 Uhr (Übungsgruppen in geraden Kalenderwochen) bzw. Dienstag 23. Juni 8:00 Uhr (Übungsgruppen in ungeraden Kalenderwochen)