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 optionalen Aufgaben (davon gibt es jeweils eine auf den Aufgabenblättern 0–3) sind ein Stück schwerer als die „normalen“ und geben *keine* zusätzlichen Punkte für das jeweilige Aufgabenblatt – aber jeweils ein „Bonus-Sternchen“ ★. Wenn ihr drei Sternchen sammelt, müsst ihr das letzte Aufgabenblatt (A4) nicht bearbeiten!

Aufgabe 3: Deadlock (10 Punkte)

Ziel der Aufgabe ist es, das Entstehen, Erkennen und Auflösen von Deadlocks praktisch umzusetzen.

ACHTUNG: Zu den untenstehenden Aufgaben existiert eine Vorgabe in Form von C-Dateien mit vorimplementierten Code-Rümpfen, die ihr in den Aufgaben 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.

Programmierung und Theoriefragen: Deadlock (10 Punkte)

Eine wichtige Aufgabe von Betriebssystemen ist die Verwaltung von Speicher. Im vierten Übungsblatt werden wir dazu auch unser eigenes Speicherallokationsverfahren implementieren. In dieser Übung wollen wir uns jedoch anhand des Beispiels für wiederverwendbare Betriebsmittel aus Foliensatz 6 zum Thema Verklemmungen auf das Hervorrufen, Erkennen und Beheben von Verklemmungen konzentrieren.
In dieser Übung soll eine Gruppe von Fäden implementiert werden, die eine zufällige Anzahl an Anfragen nach Speicher an das Speichermanagementsystem des Betriebssystems senden. Die dafür nötigen Datenstrukturen und Funktionen werden euch in dieser Übung vorgegeben. Jeder Faden verwaltet dabei seine eigene Datenstruktur, in der er vermerkt, in welcher Reihenfolge er wie viel Speicher anfordert, um danach den gesamten verwendeten Speicher wieder frei zu geben. Sollte eine Anfrage nach mehr Speicher nicht beantwortet werden können, so wird der anfragende Faden schlafen gelegt und wird erst dann wieder aufgeweckt, wenn genug Speicher für seine Anfrage zur Verfügung steht.
Der Hauptfaden soll die Aufgabe eines Überwachers übernehmen und in regelmäßigen Abständen die aktuelle Situation auswerten. Wird ein Deadlock erkannt, so soll er dies dem Nutzer bekannt geben.

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 (`aufgabe3_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 stellt und bildet somit das Grundgerüst für das beschriebene Szenario. Zudem findet ihr eine Makefile in der Vorgabe, sodass ihr das gesamte Projekt schnell mit den Befehlen make und make clean verwalten könnt. Nach der Implementierung jeder einzelnen Datei sollte das gesamte Projekt übersetzbar sein.

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

a) Theoriefragen (3 Punkte)

1. Im Laufe der Aufgabe sollen mehrere Threads Speicher anfragen und sich wieder schlafen legen. Beschreibt in eigenen Worten, wie in dieser Situation ein Deadlock zustande kommen kann. Gebt dabei insbesondere an, wie die vier Bedingungen für einen Deadlock dabei erfüllt werden. (2 Punkte)
2. Gebe eine Möglichkeit an, eine Voraussetzung für einen Deadlock in dieser Situation zu entschärfen. (1 Punkt)

⇒ `aufgabe3.txt`

b) Semaphoren implementieren (1 Punkt)

Die Datei **`aufgabe3_semaphores.c`** beinhaltet den Rumpf für die Funktionen, die die Semaphoren initialisieren und aufräumen sollen. Jeder einzelne Faden besitzt einen eigenen Semaphor (das Attribut **`sem`** der Struktur **`ThreadData`** bzw. das Element **`thread_data[X].sem`**), mit dem er wieder geweckt werden kann, wenn genug Speicher zur Verfügung steht. Der Faden soll sich mithilfe seines Semaphors sofort schlafen legen können, wenn nicht genug Speicher zur Verfügung steht. Initialisiert ihn also mit dem dazu passenden Wert.

⇒ `aufgabe3_semaphores.c`

c) Speicherverwaltung implementieren (2 Punkte)

Die Datei **`aufgabe3_memory.c`** beinhaltet den Rumpf für die Funktionen, die den Speicher verwalten sollen und die Fäden bei Bedarf schlafen legen oder aufwecken. Die Funktion **`request_memory`** ist noch leer, während die Funktion **`free_all_memory`** bereits vorgegeben ist. Implementiert die **`request_memory`**-Funktion so, dass sie die übergebene Datenstruktur des Threads, der Speicher anfordert, anpasst und den Thread mit Hilfe seines Semaphors schlafen legt, wenn nicht genug Speicher vorhanden ist. Wenn ihr die Bonusaufgabe bearbeitet, verwendet bei eurer Überprüfung bitte bereits hier die Funktion **`check_save_allocate`**. Beachtet zudem, dass Zugriffe auf die Datenstruktur der Speicherverwaltung zwar mit Hilfe der dort enthaltenen Mutexvariable geschützt werden müssen, ein wartender Thread dadurch aber nicht alle anderen Speicheroperationen blockieren darf.

⇒ `aufgabe3_memory.c`

d) Fäden implementieren (2 Punkte)

Die Datei **`aufgabe3_threads.c`** beinhaltet den Rumpf für die Funktionen, die die konkurrierenden Fäden starten und implementieren sollen. Die Funktion **`start_threads`** ist vorgegeben. Die Funktion **`thread_function`** sollt ihr implementieren. Ein Faden soll dabei seine Anfragen (das Attribut **`requests`** der **`ThreadData`** Struktur) mit Hilfe der Funktion **`request_memory`** abgeben und zwischen jeder Anfrage eine Sekunde warten. Wenn alle Anfragen beantwortet wurden, soll der gesamte verwendete Speicher mittels **`free_all_memory`** wieder freigegeben werden. Diese beiden Schritte sollen in einer Endlosschleife wiederholt werden.

Zusätzlich soll die Funktion **`cleanup`** implementiert werden, die alle laufenden Threads abbricht und das Programm geregelt beenden soll. Setzt dazu in dieser Funktion die globale Variable **`running`** so, dass der Hauptfaden das Aufräumen und Beenden des restlichen Programms übernehmen kann.

⇒ `aufgabe3_threads.c`

e) Deadlock erkennen (2 Punkte)

Die Datei **`aufgabe3_deadlock.c`** beinhaltet den Rumpf für die Funktion, die der Hauptfaden ausführen soll, um einen Deadlock zu erkennen. Implementiert diese Funktion. Der Hauptfaden legt sich nach Vorgabe nach jeder Überprüfung für ein paar Sekunden schlafen, um nicht aktiv auf das Eintreten eines Deadlocks zu warten. Die Methode soll den Deadlock nur erkennen, nicht auflösen! Gebt also eine entsprechende Nachricht an den Benutzer aus und benutzt die Funktion **`cleanup`** aus Aufgabenteil c) um das Programm zu beenden. Zur Fehlersuche könnt ihr zudem die Funktion **`print_thread_data`** verwenden.

⇒ `aufgabe3_deadlock.c`

f) Deadlocks verhindern (★)

Die Datei aufgabe3_sanity.c beinhaltet den Rumpf einer Funktion, die überprüfen soll, ob eine Allokation von Speicher „sicher“ ist. Anstatt das Programm beim Eintreten eines Deadlocks zu beenden, soll stattdessen ein Deadlock gar nicht erst eintreten können. Dazu sollt ihr die Funktion **`check_save_allocate`** um die Funktionalität erweitern, **`false`** zurück zu geben, wenn eine Allokation von Speicher für einen anfragenden Faden in der Zukunft zu einem Deadlock führen könnte. Es muss dabei jedoch gewährleistet werden, dass immer mindestens ein Faden weiterlaufen kann.

⇒ `aufgabe3_sanity.c`

Tipps zu den Programmieraufgaben:

- Bei der Kompilierung des Codes die Option **`-pthread`** für die Verwendung der posix-Threads nicht vergessen!
- Kommentiert euren Quellcode **ausführlich**, so dass wir bei Programmierfehlern noch Punkte vergeben können!
- Die Programme sollen dem C11- und POSIX-Standard entsprechen und sich mit dem gcc auf den Linux-Rechnern im IRB-Pool übersetzen lassen.
- Alternativ könnt ihr die Programme auch in C++ schreiben.

Abgabe bis Donnerstag 15. Juni 10:00 Uhr (Übungsgruppen in ungeraden Kalenderwochen) bzw. Dienstag 20. Juni 10:00 Uhr (Übungsgruppen in geraden Kalenderwochen)