Weitere Hinweise zu den Übungen Betriebssysteme

- 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 optionalen Aufgaben (davon wird es jeweils eine auf den Aufgabenblättern 0–3 geben) 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!
- Am Besten arbeitet ihr in diesem Blatt zuerst die Theoriefragen durch; die Programmieraufgaben bauen z.T. stark auf dem Theorieteil auf.
- Führt den C-Code-Schnipsel aus der Theorieaufgabe 4 nicht aus! Eine theoretische Betrachtung ist zum Lösen der Aufgabe mehr als ausreichend.

Aufgabe 1: Prozessverwaltung und fork() (10 Punkte)

Lernziel dieser Aufgabe ist die Verwendung der UNIX-Systemschnittstelle zum Erzeugen und Verwalten von Prozessen.

Theoriefragen: Prozessverwaltung (5 Punkte)

Bitte gebt diesen Aufgabenteil, wie auch in Aufgabe 0, in der Datei `antworten.txt` ab. Die Antworten sind in eigenen Worten zu formulieren.

1) Erklärt in eigenen Worten die Ausgabe des Kommandos `ls | wc -l`.

2) Wie unterscheiden sich Includes, die mit Anführungszeichen eingebunden werden, und Includes, die mit spitzen Klammern eingebunden werden?

3) Weshalb kann man in C den Inhalt von Strings nicht mit `==` vergleichen?

4) Betrachtet folgendes C-Code-Schnipsel: **/!\ nicht ausführen /!**

for(;;) fork();

a) Beschreibt das Programmverhalten nach 1, 2, 3 und n „Generationen“. Legt zu Grunde, dass alle Prozesse (hier insbesondere obige for-Schleife) parallel ausgeführt werden. Betrachtet die Gesamtheit der parallelen Schleifendurchläufe als eine „Generation“.

b) Das Verhalten eines solchen Programms kann zu Problemen führen und mittels */etc/security/limits.conf* beschränkt werden. Welche Probleme sind dies und warum sind Gegenmaßnahmen sinnvoll? Gebt ferner beispielhaft Einträge für diese Datei an, die den Benutzer „tux“ hart auf 100 und die Gruppe „guests“ hart auf 2000 Prozesse limitieren (*man 5 limits.conf*).

Programmierung: Eine eigene Shell (2+1+1+1 Punkte)

An dieser Stelle wollen wir eine eigene kleine Shell programmieren. Aus dieser sollen beliebige Programme gestartet werden können. Weiterhin soll die Shell dem Nutzer seinen Namen, den Namen des aktuellen Verzeichnisses und den Rechnernamen anzeigen.

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

a) Systeminformationen und Einlesen des Befehls Implementiert zuerst mittels **printf(3)** eine einfache Aufforderung an den Benutzer, die folgendermaßen aufgebaut ist :

```
Nutzername@Rechnername Verzeichnispfad$
```

Daraufhin sollen mittels **scanf(3)** genau ein Befehl und ein Argument eingelesen werden. Die Shell soll nun den Befehl sowie das übergebene Argument an den Nutzer ausgeben und anschließend wieder die Aufforderung anzeigen, um auf die nächste Eingabe zu warten. Dies soll innerhalb einer Endlosschleife realisiert werden. Die Ausführung eures Programms könnt ihr mit der Tastenkombination <**Strg**>+**C** abbrechen.

Eine Beispieldurchlauf des Programmes könnte folgendermaßen aussehen:

```
christoph@NCC-1742-G /home/christoph$ ls -al
Befehl: ls
Argument: -al
christoph@NCC-1742-G /home/christoph$
```

Hinweise:

- Achtet darauf, dass die Standardeingabe immer geleert wird, bevor der nächste Befehl eingelesen wird
- Setzt für alle Strings eine maximale Größe. Eine gute maximale Größe sind 256 Zeichen. Denkt daran, dass Strings in C nullterminiert sind
- In der Datei `limits.h` sind bereits maximale Größen für den aktuellen Pfad und den Rechnernamen hinterlegt (`PATH_MAX` bzw. `HOST_NAME_MAX`)
- Nutzername, Rechnername und das aktuelle Verzeichnis lassen sich mit den folgenden Funktionen ermitteln:
  - **getpwuid(3)**
  - **gethostname(2)**
  - **getcwd(3)**
- Falls ihr euer Programm mit der Option `-std=c11` kompiliert, müsst ihr zusätzlich noch `-D_GNU_SOURCE` angeben, um die Funktion **gethostname(2)** verwenden zu können.
- Fügt geeignete Fehlerabfragen hinzu! *(Dies gilt auch für alle weiteren Aufgaben)*
- Die Implementierung soll in der Datei `aufgabe1_a.c` abgegeben werden

b) Starten von Programmen Erweitert euer Programm soweit, dass die Shell den Befehl mit dem übergebenen Argument mittels **execlp(3)** ausführt.

Hinweis:

- Wenn ihr die erste Teilaufgabe nicht gelöst habt, könnt ihr davon ausgehen, dass als Befehl `ls` und als Argument `-al` übergeben wurde.
- Die Implementierung soll in der Datei `aufgabe1_b.c` abgegeben werden

c) Weitere Programme starten Ein Problem mit der Realisierung aus Ausgabenteil **b)** ist, dass die Shell beendet wird, sobald sich das aufgerufene Programm beendet. Sorgt mittels **fork(2)** und **wait(2)** oder **waitpid(2)** dafür, dass die Shell nach dem Ende des aufgerufenen Programms neue Eingaben verarbeiten kann.

- Die Implementierung soll in der Datei `aufgabe1_c.c` abgegeben werden

d) Eingebaute Shell-Befehle

Zusätzlich soll es nun möglich sein, Verzeichnisse zu wechseln und die Shell zu beenden. Falls als Befehl **cd** eingegeben wurde, soll nun kein Programm gestartet, sondern das Verzeichnis abhängig von dem Argument gewechselt werden. Ist der Befehl **exit**, soll sich die Shell mit dem als Argument übergebenen Rückgabewert beenden.

Hinweis:

- **strtol(3)** wandelt einen String in eine Zahl um
- Die Implementierung soll in der Datei `aufgabe1_d.c` abgegeben werden

e) Ausführen im Hintergrund optional (★) Es soll nun auch möglich sein, Programme im Hintergrund auszuführen. Sobald ein Befehl mit einem „`!`“ beginnt, soll die Shell nicht auf die Beendigung des Prozesses warten, sondern direkt weitere Eingaben von dem Benutzer erwarten. Achtet dabei darauf, dass auch diese Prozesse abgeräumt werden müssen, um Zombies zu vermeiden. Fügt dafür an geeigneter Stelle einen Aufruf an **waitpid(2)** ein.

- Die Implementierung soll in der Datei `aufgabe1_e.c` abgegeben werden

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 dazu mit folgenden Parametern aufzurufen: *gcc -Wall -o ziel datei1.c* Weitere (nicht zwingend zu verwendende) Compilerflags, die dafür sorgen, dass man sich enger an die Standards hält, sind: `-std=c11 -Wpedantic -Werror -D_GNU_SOURCE`
- Alternativ kann auch der GNU C++-Compiler (*g++*) verwendet werden.

Abgabe: bis spätestens Donnerstag, 18. Mai 10:00 (Übungsgruppen mit ungerader Kalenderwoche) bzw. Dienstag, 23. Mai 10:00 (Übungsgruppen mit gerader Kalenderwoche).