Aufgabe 1: Ankreuzfragen (22 Punkte)

1) Einfachauswahlfragen (18 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (⊗) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) Welche Aussage über Prozesszustände ist in einem Monoprozessor-Betriebssystem mit blockierenden Ein-/Ausgabeoperationen richtig? 2 Punkte

- ○ Wenn gerade keine Prozessumschaltung stattfindet und ein Prozess im Zustand *laufend* ist, so gibt es mindestens einen Prozess im Zustand *blockiert*.
- ○ Wenn gerade keine Prozessumschaltung stattfindet und kein Prozess im Zustand *laufend* ist, so ist auch kein Prozess im Zustand *blockiert*.
- ○ Ein Prozess im Zustand *laufend* wird in den Zustand *blockiert* überführt, wenn eine seiner Ein-/Ausgabeoperation nicht sofort abgeschlossen werden kann.
- ○ Es befinden sich bis zu zwei Prozesse im Zustand *laufend* und damit in Ausführung auf dem Prozessor (Vordergrund- und Hintergrundprozess).

b) Welche Aussage zum Thema Adressraumschutz ist richtig? 2 Punkte

- ○ Bei allen Verfahren des Adressraumschutzes führt jeder Zugriff auf eine ungültige Speicheradresse zu einem Trap.
- ○ Adressraumschutz durch Abgrenzung erlaubt es, mehrere Benutzerprozesse voneinander zu isolieren.
- ○ Beim Adressraumschutz durch Abgrenzung wird der logische Adressraum in mehrere Segmente mit unterschiedlicher Semantik unterteilt.
- ○ Adressraumschutz durch Abgrenzung eignet sich besonders für Systeme, die von mehreren Nutzern gleichzeitig verwendet werden.

c) Ein Programm will die drei Zeichenketten `char a[] = "path"; char b[] = "to"; char c[] = "video";` mit der Funktion `sprintf(3)` wie folgt in einen Puffer buffer speichern: `sprintf(buffer, "%s/%s/%s", a, b, c);` Mit welcher Länge (in Bytes) muss der Puffer `buffer` mindestens angelegt werden, damit kein Überlauf entstehen kann? 2 Punkte

- ○ 12
- ○ 13
- ○ 14
- ○ 15

d) Welche Antwort trifft für die Eigenschaften eines Linux-Dateideskriptors zu? 2 Punkte

- ○ Ein Dateideskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine Datei benutzen kann.
- ○ Dateideskriptoren sind Zeiger auf betriebssystem-interne Strukturen, die von den Systemaufrufen ausgewertet werden, um auf Dateien zuzugreifen.
- ○ Ein Dateideskriptor ist eine Integerzahl, die über gemeinsamen Speicher an einen anderen Prozess übergeben werden kann, und von letzterem zum Zugriff auf eine geöffnete Datei verwendet werden kann.
- ○ Beim Öffnen ein und derselben Datei erhält ein Prozess jeweils die gleiche Integerzahl als Dateideskriptor zum Zugriff zurück.

e) Ein laufender Prozess wird durch den Scheduler verdrängt. Welcher Zustandsübergang findet statt? 2 Punkte

- ○ Der Prozess wechselt vom Zustand *laufend* in den Zustand *blockiert*.
- ○ Der Prozess wechselt vom Zustand *bereit* in den Zustand *blockiert*.
- ○ Der Prozess wechselt vom Zustand *laufend* in den Zustand *beendet*.
- ○ Der Prozess wechselt vom Zustand *laufend* in den Zustand *bereit*.

f) In einem Unix/Linux-Dateisystem gibt es symbolische Verknüpfungen (symbolic links) und feste Verknüpfungen (hard links) auf Dateien. Welche Aussage ist richtig? 2 Punkte

- ○ Wird die letzte symbolische Verknüpfung (symbolic link) auf eine Datei gelöscht, so wird auch die Datei selbst gelöscht.
- ○ Eine feste Verknüpfung (hard link) kann nur auf Verzeichnisse, nicht jedoch auf Dateien verweisen.
- ○ Eine symbolische Verknüpfung (symbolic link) kann nicht auf Dateien anderer Dateisysteme verweisen.
- ○ Für jede reguläre Datei existiert mindestens eine feste Verknüpfung (hard link) im selben Dateisystem.

g) Welche Antwort beschreibt ein im Inode (Dateikopf) gespeichertes Attribut einer Datei eines UNIX/Linux-Dateisystems? 2 Punkte

- ○ Anzahl der festen Verknüpfungen (hard links)
- ○ Anzahl der symbolischen Verknüpfungen (symbolic links)
- ○ Dateiname
- ○ Position des Schreib-Lese-Zeigers

h) Welche Aussage zum Thema Systemaufrufe ist richtig?

2 Punkte

- ○ Nach der Bearbeitung eines beliebigen Systemaufrufes ist es für das Betriebssystem nicht mehr möglich, zu dem Prozess zu wechseln, welcher den Systemaufruf abgesetzt hat.
- ○ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht selbst ausführen dürfte.
- ○ Durch einen Systemaufruf wechselt das Betriebssystem von der Systemebene auf die Benutzerebene, um unprivilegierte Operationen ausführen zu können.
- ○ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.

i) Wie viele Prozesse existieren nach Ausführung des nachfolgenden Programmausschnitts? Gehen Sie davon aus, dass alle Aufrufe von fork() erfolgreich sind.

2 Punkte

```
fork(); fork(); fork();
```

- ○ 3
- ○ 8
- ○ 4
- ○ 1

2) Mehrfachauswahlfragen (4 Punkte)

Bei den Mehrfachauswahlfragen in dieser Aufgabe sind jeweils m Aussagen angegeben, davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie alle richtigen Aussagen an.
Jede korrekte Antwort in einer Teilaufgabe gibt einen Punkt, jede falsche Antwort einen Minuspunkt. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, d. h. falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Welche der folgenden Aussagen zum Thema Threads ist richtig?

4 Punkte

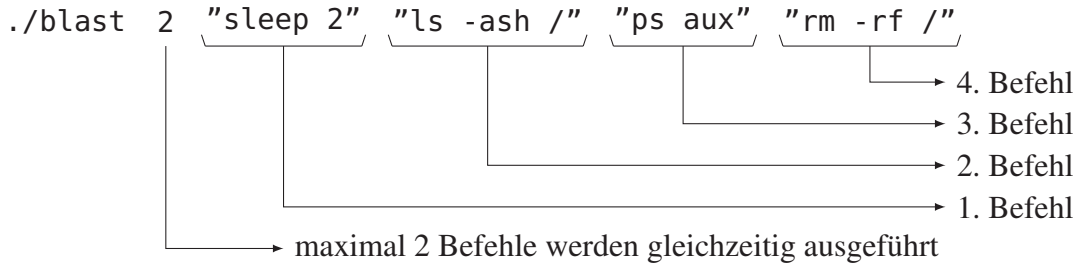
- ☐ *Kernel-Level Threads* setzen den Einsatz von verdrängenden Schedulingverfahren voraus.
- ☐ Bei *Kernel-Level Threads* ist die Schedulingstrategie durch das Betriebssystem implementiert.
- ☐ *Kernel-Level Threads* blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ☐ Die Umschaltung von *User-* und *Kernel-Level Threads* muss immer im Systemkern erfolgen (privilegierter Maschinenbefehl).
- ☐ Die Umschaltung von *User-Level Threads* ist sehr effizient.
- ☐ *Kernel-Level Threads* können Multiprozessoren ausnutzen.
- ☐ Bei *User-Level Threads* ist die Schedulingstrategie durch die Anwendung bzw. die von ihr genutzten Bibliotheken vorgegeben.
- ☐ *Kernel-Level Threads* teilen sich den kompletten Adressraum und verwenden daher denselben Stack.

Aufgabe 2: BLAST - Batch Launcher for Asynchronous Shell Tasks (45 Punkte)

Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!

Schreiben Sie ein Programm *BLAST*, das per Befehlszeile übergebene Befehle parallel ausführt. Dabei soll darauf geachtet werden, dass maximal n Befehle gleichzeitig laufen. Die Obergrenze n soll dabei ebenfalls per Befehlszeile übergeben werden.

Beispielhafter Aufruf von *BLAST* (4 Befehle, davon maximal 2 parallel):



Das Programm soll folgendermaßen strukturiert sein:

- Funktion `main()`: Initialisiert zunächst alle benötigten Datenstrukturen und prüft die Befehlszeilenargumente. Nutzen Sie zum Umwandeln der Obergrenze n die Funktion `strtol`, um eine Fehlerprüfung zu ermöglichen.

  Im Anschluss daran werden die als Befehlszeilenargumente übergebenen Befehle parallel ausgeführt. Dabei sollen, solange noch nicht ausgeführte Befehle vorhanden sind, stets genau n Befehle parallel laufen. Zur Verwaltung der gestarteten Prozesse soll ein **`struct`** `process`-Array verwendet werden. Nachdem alle Befehle gestartet wurden, soll auf die noch laufenden Prozesse gewartet werden. *Hinweis:* Es steht Ihnen frei, die Definition der Struktur um zusätzliche Einträge zu erweitern.

- Funktion `pid_t run(`**`const char`** `*cmdline)`: Führt die übergebene Befehlszeile aus. Dazu werden das auszuführende Programm und die Parameter mittels `strtok` aus der Befehlszeile extrahiert und mithilfe einer Funktion der `exec`-Familie ausgeführt. Tritt hierbei ein Fehler auf, wird eine aussagekräftige Fehlermeldung ausgegeben und *BLAST* beendet. Achten Sie auf die passende Dimensionierung eventuell benötigter Arrays. Zur Vereinfachung dürfen Sie annehmen, dass die zu extrahierenden Parameter durch Leerzeichen voneinander getrennt sind und sich innerhalb der einzelnen Parameter keine weiteren Leerzeichen befinden.

- Funktion **`void`** `waitProcess(`**`struct`** `process *processes, size_t size)`: Wartet passiv auf einen beliebigen der zuvor gestarteten Befehle. Nach Terminierung des Prozesses gibt `waitProcess` die PID, die Befehlszeile, den Terminierungsgrund (falls verfügbar) und die Ausführungsdauer aus.

Alle Ausgaben des Programms sollen auf `stderr` geschrieben werden (ein manuelles Spülen via `fflush` und die Fehlerbehandlung der Ausgabe ist nicht erforderlich). Nutzen Sie die Funktion `time` (siehe Manpage) zur Bestimmung der Ausführungsdauer. Die Vorausdeklaration der Funktionen `run` und `waitProcess` ist nicht notwendig.

Auf den folgenden Seiten finden Sie ein Gerüst für das beschriebene Programm. In den Kommentaren sind nur die wesentlichen Aufgaben der einzelnen zu ergänzenden Programmteile beschrieben, um Ihnen eine gewisse Leitlinie zu geben. Es ist überall sehr großzügig Platz gelassen, damit Sie auch weitere notwendige Programmanweisungen entsprechend Ihrer Programmierung einfügen können.

Einige wichtige Manual-Seiten liegen bei – es kann aber durchaus sein, dass Sie bei Ihrer Lösung nicht alle diese Funktionen oder gegebenenfalls auch weitere Funktionen benötigen.

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>


static void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

static void usage(void) {
    fprintf(stderr, "Usage: blast <max. parallel processes> <cmdlines>\n");
    exit(EXIT_FAILURE);
}
```

```c
struct process {
    pid_t  pid;     // PID des Prozesses

    // Eigene Mitglieder

};
```

```
// Funktion main()
int main(int argc, char *argv[]) {

    // Argumente prüfen und bearbeiten

    // Übergebene Befehle ausführen
```

```
}
// Ende Funktion main()
```

M:

```
// Funktion run()
```

```
// Ende Funktion run()
```

R:

```
// Funktion waitProcess()
```

```
// Ende Funktion waitProcess()
```

W:

2) Makefile (7 Punkte)

Schreiben Sie ein Makefile, welches die Targets `all` und `clean` konventionskonform unterstützt. `all` soll hierbei das Defaulttarget sein. Ebenfalls soll ein Target `blast` unterstützt werden, welches das Programm `blast` aus der Quelldatei `blast.c` baut. Greifen Sie dabei stets auf Zwischenprodukte (z.B. `blast.o`) zurück.

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `blast` löschen.

Nutzen Sie dabei die Variablen `CC` und `CFLAGS` konventionskonform. Achten Sie darauf, dass das Makefile ohne eingebaute Regeln (Aufruf von `make -Rr`) funktioniert!

```
# Makefile
CC=gcc
CFLAGS=-std=c11 -pedantic -Wall -Werror \
       -fsanitize=undefined -fno-sanitize-recover \
       -g -D_XOPEN_SOURCE=700
```

Mk:

Aufgabe 3: Ausnahmen (12 Punkte)

1) Sie haben in der Vorlesung zwei Arten von Ausnahmen von der normalen Programmausführung kennengelernt. Wie lauten diese verschiedenen Arten von Ausnahmen? Nennen sie zwei Eigenschaften, durch die sie sich voneinander unterscheiden. (3 Punkte)

2) Nennen Sie die zwei Modelle der Ausnahmebehandlung. Wie unterscheiden sich diese Modelle? Nennen Sie für jedes der Modelle je einen Auslösungsgrund. (5 Punkte)

3) Untenstehender Code wird auf einem x86_64-Linux-System ausgeführt. (4 Punkte)

```
char *ptr = NULL;
*ptr = 'c';
```

a) Wodurch tritt hier ein Fehler auf? Was wird der Anwendung signalisiert? (2 Punkte)

b) Welche Hardwarekomponente entdeckt diesen Fehler? Wie signalisiert diese Komponente den Fehler an das Betriebssystem? (2 Punkte)

Aufgabe 4: Adressräume (6 Punkte)

1) Warum kann es nötig werden, dass ein Benutzerprogramm seinen virtuellen/logischen Adressraum zur Laufzeit erweitert? Durch welche Schnittstellen wird dies vom Betriebssystem ermöglicht und warum werden diese dazu von normalen Benutzerprogrammen nur in Spezialfällen direkt verwendet? Welche Schnittstelle wird von Benutzerprogrammen stattdessen normalerweise verwendet? (4 Punkte)

2) Erklären Sie einen Vorteil und einen Nachteil von virtuellem Speicher. (2 Punkte)

Aufgabe 5: Synchronisation (5 Punkte)

1) Erläutern Sie das Konzept Semaphor. Welche Operationen sind auf Semaphoren definiert und welche Eigenschaften haben diese Operationen? (5 Punkte)