Aufgabe 1.1: Einfachauswahl-Fragen (18 Punkte)

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

Wollen Sie eine Multiple-Choice-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 zum Thema Adressräume ist richtig? 2 Punkte

- ❏ Der einem Prozess zugeteilte Adressraum ist in seiner Größe nicht durch die Hardware beschränkt.
- ❏ Wird in einem System die Nutzung virtueller Adressräume unterstützt, so können Teile des Arbeitsspeichers in den Ablagespeicher (*swap area*) ausgelagert sein.
- ❏ In einem realen Adressraum sind alle Adressen gültig.
- ❏ Das Betriebssystem bildet den realen Adressraum auf logische Adressen ab.

b) In einem UNIX-Dateisystem gibt es symbolische Verweise (*symbolic links*) und feste Verweise (*hard links*) Welche der folgenden Aussagen ist richtig? 2 Punkte

- ❏ Auf ein Verzeichnis verweist immer genau ein *symbolic link*.
- ❏ Ein *symbolic link* kann nicht auf Dateien in anderen Dateisystemen verweisen.
- ❏ Ein *hard link* kann nicht auf Dateien, sondern nur auf Verzeichnisse verweisen.
- ❏ Die Anzahl der *hard links*, die auf ein Verzeichnis verweisen, hängt von der Anzahl seiner Unterverzeichnisse ab.

c) Man unterscheidet zwischen Traps und Interrupts. Welche Aussage ist richtig? 2 Punkte

- ❏ Traps können nicht durch Speicherzugriffe ausgelöst werden.
- ❏ Bei der mehrfachen Ausführung eines unveränderten Programmes mit den selben Eingabedaten treten Traps immer an den selben Stellen auf.
- ❏ Ein Trap wird immer durch das Programm behandelt, welches den Trap ausgelöst hat, Interrupts werden hingegen immer durch das Betriebssystem behandelt.
- ❏ Die Behandlung eines Interrupts führt zwingend zur Beendigung des aktuell laufenden Prozesses.

d) Welche Aussage zum Thema Systemaufrufe ist richtig? 2 Punkte

- ❏ Die Bearbeitung eines Systemaufrufs findet immer im selben Adressraum statt, aus dem heraus der Systemaufruf abgesetzt wurde.
- ❏ Durch einen Systemaufruf wechselt das Betriebssystem in den Adressraum von Maschinenprogrammen auf der Benutzerebene.
- ❏ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht ausführen dürfte.
- ❏ Nach der Bearbeitung eines beliebigen Systemaufrufes ist es für das Betriebssystem nicht mehr möglich, zu dem Programm zu wechseln, welches den Systemaufruf abgesetzt hat.

e) Man unterscheidet die Begriffe *Programm* und *Prozess*. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig? 2 Punkte

- ❏ Ein Programm kann immer nur von einem Prozess gleichzeitig ausgeführt werden.
- ❏ Ein Prozess kann mit Hilfe von Threads mehrere Programme gleichzeitig ausführen.
- ❏ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.
- ❏ Der Übersetzer (Compiler) erzeugt aus mehreren Programmteilen (Modulen) einen Prozess.

f) Welche Aussage über `exec()` ist richtig? 2 Punkte

- ❏ Dem Vater-Prozess wird die Prozess-ID des neu erzeugten Kind-Prozesses zurückgeliefert.
- ❏ Beim Aufruf von `exec()` wird das im aktuellen Prozess laufende Programm durch das angegebene Programm ersetzt.
- ❏ Der an `exec()` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.
- ❏ Nach einem erfolgreichen Aufruf von `exec()` kann weiterhin auf Datenstrukturen im Adressraum des Aufrufers zugegriffen werden.

g) Welche Aussage zu Prozesszuständen ist richtig? 2 Punkte

- ❏ Ein Prozess kann direkt vom Zustand *laufend* in den Zustand *bereit* versetzt werden.
- ❏ Pro Rechenkern kann es beliebig viele Prozesse im Zustand *laufend* geben.
- ❏ Beendet sich ein Prozess, wird er vom Betriebssystem vom Zustand *bereit* in den Zustand *blockiert* überführt.
- ❏ Ein Prozess, der sich im Zustand *blockiert* befindet, kann diesen Zustand durch einen Aufruf der Funktion *free()* wieder verlassen.

h) Welche Aussage zu Zeigern ist richtig? 2 Punkte

- ❏ Der Compiler erkennt bei der Verwendung eines ungültigen Zeigers die problematische Code-Stelle und generiert Code, der zur Laufzeit die Meldung "*Segmentation fault*" ausgibt.
- ❏ Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden.
- ❏ Zeiger können in C nicht als Parameter an Funktionen übergeben werden.
- ❏ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.

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

- ❏ Beim Einsatz von Adressraumschutz durch Eingrenzung können keine Zugriffsfehler (*segmentation faults*) auftreten.
- ❏ In einem segmentierten Adressraum kann zur Laufzeit kein weiterer Speicher mehr dynamisch nachgefordert werden.
- ❏ Adressraumschutz durch Abteilung erlaubt es, mehrere Benutzerprozesse voneinander zu isolieren.
- ❏ Beim Einsatz von Segmentierung ist es möglich, dass die selbe logische Adresse in unterschiedlichen logischen Adressräumen auf unterschiedliche physikalische Adresse verweist.

Aufgabe 1.2: Mehrfachauswahl-Fragen (4 Punkte)

Bei den Multiple-Choice-Fragen in dieser Aufgabe sind jeweils *m* Aussagen angegeben, *n* ($0 \leq n \leq m$) Aussagen davon sind richtig. Kreuzen Sie **alle richtigen** Aussagen an. Jede korrekte Antwort in einer Teilaufgabe gibt einen halben Punkt, jede falsche Antwort einen halben Minuspunkt. Eine Teilaufgabse 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) Gegeben sei folgendes Programmfragment: 4 Punkte

```
const int a = 20160719;
static int bar(int x) {
    int b;
    int *c[800];
    c[0] = &b;
    ...
    x = x + a;
    return x + b;
}
```

Welche der folgenden Aussagen zu den Variablen im Programm sind richtig?

- ❍ `a` liegt auf dem Stack.
- ❍ `b` ist uninitialisiert und enthält einen zufälligen Wert.
- ❍ `bar` kann aus anderen Modulen heraus aufgerufen werden.
- ❍ Die Anweisung `x = x + a;` ändert den Wert von `x` und beeinflusst somit auch Wert im Aufrufer.
- ❍ `c` bezeichnet ein Array, in dem Platz für 800 Zeiger auf Ganzzahlen vom Typ `int` ist.
- ❍ Die Anweisung `c[0] = &b;` kann einen *Segmentation fault* auslösen.
- ❍ `b` verliert beim Rücksprung aus `bar` seine Gültigkeit.
- ❍ Auf den Wert von `a` kann von anderen Modulen aus zugegriffen werden.

Aufgabe 2: fpid (46 Punkte)

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

Schreiben Sie ein Programm **`fpid`** (**f**iles **p**er **id**), das die Anzahl der Dateien und Verzeichnisse, die einem Nutzer mit einer übergebenen Benutzer-ID gehören, in allen übergebenen Verzeichnisbäume parallel ermittelt und anschließend ausgibt. Der Grad der Parallelität ist durch die Konstante `MAXTHREADS` vorgegeben.

Das Programm bekommt auf der Befehlszeile vom Benutzer als ersten Parameter die Benutzer-ID, nach der gesucht werden soll. Für diesen Parameter sollen nur nichtnegative Ganzzahlen gültig sein. Die folgenden Parameter sind die Namen der zu durchsuchenden Verzeichnisbäume.

Das Programm soll folgendermaßen arbeiten:

- Der Haupt-Thread startet nach der Prüfung der Parameter und der Initialisierung aller relevanten Datenstrukturen für jeden als Parameter übergebenen Verzeichnisbaum die Funktion `tstart` in einem eigenen Thread. Ist die maximale Anzahl an gleichzeitig laufenden Threads erreicht, wartet der Haupt-Thread passiv, bis wieder ein neuer Thread gestartet werden kann.
  Nachdem für jeden Verzeichnisbaum die Anzahl der passenden Verzeichniseinträge parallel ermittelt wurde, gibt der Hauptthread eine Liste der Verzeichnisnamen zusammen mit der ermittelten Gesamtsumme aus.
  Trat bei der Bearbeitung eines Verzeichnisbaums ein Fehler auf, so wird als Ergebnis -1 angezeigt.
- Funktion **`void *tstart(void *arg)`**: Berechnet für das per Struktur übergebene Verzeichnis die Anzahl der Einträge mit Hilfe der rekursiven Funktion `getEntriesForID` und legt das Ergebnis in der Struktur ab. Die Parameter-Struktur ist von Ihnen passend zu definieren.
- Funktion **`int getEntriesForID(char *path, int uid)`**: Durchsucht das als Parameter übergebene Verzeichnis und summiert die Anzahl aller Einträge, die dem Benutzer mit der Benutzer-ID **`uid`** gehören, rekursiv auf. Dabei sollen nur Verzeichnisse und reguläre Dateien berücksichtigt werden.
  Hinweis: Verzeichnisse, die dem gesuchten Benutzer nicht gehören, sollen zwar nicht in das Gesamtergebnis aufgenommen werden, müssen aber dennoch rekursiv durchsucht werden!
  Tritt bei der Bearbeitung ein Fehler auf, wird die Rekursion abgebrochen und -1 zurückgegeben.

Ihnen steht das aus der Übung bekannte Semaphoren-Modul zur Verfügung. Die Schnittstelle finden Sie im folgenen Programmgerüst nach den #include-Anweisungen.

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.


```
#include <dirent.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#include "sem.h"

SEM *semCreate(int initVal);
void semDestroy(SEM *sem);
void P(SEM *sem);
void V(SEM *sem);

static const int MAXTHREADS = 4;

static void die(const char message[]) {
        perror(message);
        exit(EXIT_FAILURE);
}

// Makros, Funktionsdeklarationen, Strukturen, globale Variablen
```

```
// Funktion main()
```

```
    // Argumente auswerten und weitere Initialisierungen
```

A:


```
//  Verzeichnisse parallel bearbeiten
```

```
// Auf Terminierung der Threads warten, Ergebnis ausgeben
```

```
// Ende Funktion main
```

T:

```
// Funktion tstart()
```

```
// Ende Funktion tstart
// Funktion getEntriesForID()
```

```
   // Verzeichnis-Eintraege lesen
```


```
   // Relevante Eintraege ermitteln und aufsummieren
```

```
// Ende Funktion getEntriesForID
```

D:

Aufgabe 3: (23 Punkte)

a) Virtuelle Adressräume unterscheiden sich von logischen Adressräumen dadurch, dass nicht jede virtuelle Adresse auf ein Datum im Hauptspeicher abgebildet sein muss. Beschreiben Sie was passiert, wenn bei der Ausführung eines Programms solch eine nicht abgebildete Adresse benutzt wird. (3 Punkte)

b) Speicherzuteilung erfolgt typischerweise auf verschiedenen Ebenen. Zum einen durch das Laufzeitsystem im Anwendungsprogramm (Maschinenprogramm), zum anderen auf Betriebssystemebene. Welche Art von Speicher wird auf diesen Ebenen jeweils verwaltet, was sind jeweils Ziele und Kriterien bei der Verwaltung und in welcher Granularität wird der Speicher gehandhabt. (7 Punkte)

c) Wie arbeitet ein Anwendungsprogramm mit der Speicherzuteilung und wie arbeiten die beiden Ebenen in der Praxis zusammen (erläutern Sie dies anhand der typischen Schnittstellenfunktionen in einem UNIX-ähnlichen System). (6 Punkte)

d) Gegeben ist ein System mit seitennummeriertem Adressraum (paged address space), Adresslänge = 16Bit, Seitengröße = 4096 Bytes, Hauptspeichergröße = 64 KiBi Bytes (zur Erinnerung: $4096_{10} = 1000_{16}$).
Die Seite an der virtuellen Adresse 8192 ist auf die Kachel (den Seitenrahmen) an der Adresse 4096 im physikalischen Hauptspeicher abgebildet.
Skizzieren und benennen Sie die Datenstrukturen, die Betriebssystem und MMU für die Abbildungsfunktion benötigen, und tragen Sie die Abbildung der virtuellen Adresse 0x2160 auf die physikalische Adresse ein. (7 Punkte)