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 zum Thema Threads ist richtig? 2 Punkte

- ☐ Zur Umschaltung von Userlevel-Threads ist ein Adressraumwechsel erforderlich.
- ☐ Kernel-Threads können Multiprozessoren nicht ausnutzen.
- ☐ Zu jedem Kernel-Thread gehört ein eigener isolierter Adressraum.
- ☐ Die Umschaltung von Userlevel-Threads ist sehr effizient.

b) Welche Aussage über Variablen und Funktionen in C-Programmen ist richtig? 2 Punkte

- ☐ Lokale automatic-Variablen, die auf dem Stack angelegt werden, werden immer mit dem Wert 0 initialisiert.
- ☐ Eine Funktion, die mit dem Schlüsselwort static definiert wird, kann nur innerhalb des Moduls verwendet werden, in dem sie definiert wurde, nicht jedoch aus einem anderen Modul heraus.
- ☐ Es ist nicht möglich, Zeiger als Parameter an Funktionen zu übergeben.
- ☐ Wird dem Parameter einer Funktion innerhalb der Funktion ein neuer Wert zugewiesen, so ändert sich auch der Wert der Variablen, die von der aufrufenden Funktion als Parameter übergeben wurde.

c) Was passiert, wenn Sie in einem C-Programm über einen ungültigen Zeiger versuchen auf Speicher zuzugreifen? 2 Punkte

- ☐ Beim Laden des Programms wird die ungültige Adresse erkannt und der Speicherzugriff durch einen Sprung auf eine Abbruchfunktion ersetzt. Diese Funktion beendet das Programm mit der Meldung ”Segmentation fault”.
- ☐ Die MMU erkennt die ungültige Adresse bei der Adressumsetzung und löst einen Trap aus.
- ☐ Der Compiler erkennt die problematische Code-Stelle und generiert Code, der zur Laufzeit bei dem Zugriff einen entsprechenden Fehler auslöst.
- ☐ Das Betriebssystem erkennt die ungültige Adresse bei der Weitergabe des Befehls an die CPU (partielle Interpretation) und leitet eine Ausnahmebehandlung ein.

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

- ☐ Pro Prozessor kann es stets nur einen laufenden, jedoch mehr als einen bereiten Prozess geben.
- ☐ Ein Prozess kann mit Hilfe von Spezialbefehlen selbst vom Zustand bereit in den Zustand laufend wechseln.
- ☐ Terminiert ein laufender Prozess, so wird er vom Betriebssystem in den Zustand blockiert überführt.
- ☐ Ein Prozess kann nicht direkt vom Zustand laufend in den Zustand bereit überführt werden.

e) Bei der Behandlung von Ausnahmen (Traps oder Interrupts) unterscheidet man zwei Bearbeitungsmodelle. Welche Aussage hierzu ist richtig? 2 Punkte

- ☐ Das Wiederaufnahmemodell ist sowohl für Interrupts als auch für Traps geeignet.
- ☐ Das Betriebssystem kann Interrupts, die in ursächlichem Zusammenhang mit dem gerade laufenden Prozess stehen, nach dem Beendigungsmodell behandeln, wenn eine sinnvolle Fortführung des Prozesses nicht mehr möglich ist.
- ☐ Bei der Behandlung einer Ausnahme nach dem Wiederaufnahmemodell wird der unterbrochene Prozess neu gestartet.
- ☐ Das Beendigungsmodell ist für Interrupts und Traps gleichermaßen geeignet.

f) 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.

g) Was versteht man unter virtuellem Speicher?

2 Punkte

- ☐ Virtueller Speicher kann dynamisch zur Laufzeit von einem Programm erzeugt werden.
- ☐ Speicher, der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber möglicherweise größer als der verfügbare physikalische Hauptspeicher ist.
- ☐ Unter einem virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.
- ☐ Virtueller Speicher ist eine Bezeichnung für die in der konkreten Hardwarekonfiguration nicht vorhandenen Speicherbereiche des physikalischen Adressraums.

h) Welche Aussage über den Rückgabewert von fork() ist richtig?

2 Punkte

- ☐ Der Kind-Prozess bekommt die Prozess-ID des Eltern-Prozesses.
- ☐ Bei erfolgreicher Ausführung kehrt fork() nicht zum Eltern-Prozess zurück.
- ☐ Dem Eltern-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.
- ☐ Der Rückgabewert ist in jedem Prozess (Kind- und Eltern-Prozess) jeweils die eigene Prozess-ID.

i) Welche Antwort trifft für die Eigenschaften eines UNIX/Linux-Filedeskriptors zu?

2 Punkte

- ☐ Ein Filedeskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine Datei, ein Gerät, einen Socket oder eine Pipe benutzen kann.
- ☐ Beim mehrfachen Öffnen ein und derselben Datei erhält ein Prozess jeweils die gleiche Integerzahl als Filedeskriptor zum Zugriff zurück.
- ☐ Ein Filedeskriptor ist ein Zeiger auf eine Datenstruktur des Systemkerns, den der Prozess zum Zugriff auf eine Datei benutzen kann.
- ☐ Ein Filedeskriptor beschreibt die Eigenschaften, wie Größe und Zugriffsrechte, einer Datei.

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) Gegeben sei folgendes Programmfragment:

4 Punkte

```
static int a = 20140214;
int foo(int x) {
  static int b;
  int c;
  int (*d)(const int *) = foo;
  int *e = malloc(800);
  ++x;
  // ...
}
```

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

- ◯ Die Anweisung `++x` ändert den Wert von `x` und beeinflusst somit den Aufrufer.
- ◯ `c` verliert beim Rücksprung aus `foo` seine Gültigkeit.
- ◯ `e` liegt auf dem Stack.
- ◯ Die in `e` nach der Zuweisung enthaltene Speicheradresse kann problemlos verwendet werden.
- ◯ `b` ist mit 0 initialisiert und liegt im BSS-Segment.
- ◯ Das Ergebnis des Aufrufs der Funktion `foo` wird in `d` gespeichert.
- ◯ `e` zeigt auf ein Array, in dem Platz für 800 Ganzzahlen vom Typ **`int`** ist.
- ◯ `c` ist uninitialisiert und enthält einen undefinierten Wert.

Aufgabe 2: sp-exam (45 Punkte)

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

Schreiben Sie ein Programm *sp-exam* (**S**orted **P**rogram **Ex**ecution **A**nd **M**onitoring), das übergebene Verzeichnisse rekursiv nach regulären und ausführbaren Dateien durchsucht und diese in alphabetischer Reihenfolge ausführt. Nachfolgend sehen Sie beispielhaft den Inhalt eines Verzeichnisses *sp* sowie eine mögliche Ausgabe der Ausführung von *sp-exam* auf diesem Verzeichnis.

```
leo@beach:~ %ls -R sp
sp:
00-generate-exam.sh  01-replace-easy-exercises.sh  post

sp/post:
grade.sh
leo@beach:~ %./sp-exam sp
sp/00-generate-exam.sh: exited with status code 0 in 42s.
sp/01-replace-easy-exercises.sh: terminated unexpectedly.
sp/post/grade.sh: exited with status code 3 in 3s.
Executed 3 programs: 1 succeeded, 2 failed.
```

Das Programm soll folgendermaßen arbeiten:

- Das Programm bekommt als Argumente die zu durchsuchenden Verzeichnisse übergeben.

  Die übergebenen Verzeichnisse werden nacheinander rekursiv nach Programmen durchsucht (`find_executables`). Wurden keine Verzeichnisse übergeben, soll das aktuelle Verzeichnis (".") durchsucht werden. Nachdem alle Verzeichnisse durchsucht wurden, werden alle gefundenen Dateien in alphabetischer Reihenfolge sortiert (`qsort(3)`) und in ebendieser Reihenfolge nacheinander ausgeführt (`run`). Am Ende soll die Anzahl der insgesamt gefundenen Programme sowie die Anzahl der erfolgreichen und gescheiterten Ausführungen ausgegeben werden (siehe Beispiel). Als erfolgreich zählen nur Programme, die mit Exitcode 0 beendet wurden.

- Funktion `void find_executables(const char *path)`

  Durchsucht das übergebene Verzeichnis (`path`) rekursiv nach regulären und **für den Besitzer** ausführbaren Dateien (Überprüfung mit einer Funktion der `stat(2)`-Familie) und speichert diese in einem globalen Puffer. Es sind keine Annahmen über die maximale Anzahl an Programmen möglich. Die Funktion muss in der Lage sein, den Puffer gegebenenfalls zu vergrößern.

- Funktion `void run(const char *prog)`

  Führt das übergebene Programm (`prog`) mit einer Funktion der `exec(3)`-Familie in einem Kindprozess aus und wartet im Elternprozess auf dessen Beendigung. Das Programm wird ohne Übergabe von Argumenten gestartet. Wurde ein Programm regulär beendet, soll dessen Exitstatus sowie die benötigte Zeit (`time(2)`) ausgegeben werden. In allen anderen Fällen soll stattdessen eine entsprechende Fehlermeldung ausgegeben werden. Eine mögliche Ausgabe ist im Beispiel gezeigt.

Hinweise:

- Im Fehlerfall dürfen Sie die Ausführung (bspw. mithilfe von `die()`) abbrechen.
- Symbolische Links sollen beim Durchsuchen des Verzeichnisses nicht aufgelöst werden.
- **Dynamische Speicheranforderung** soll **ausschließlich** in `find_executables` erfolgen.
- Achten Sie darauf, im Erfolgsfall alle angeforderten Ressourcen auch wieder freizugeben.

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

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

// Funktions- & Strukturdekl., globale Variablen, etc.
```

```
// Hauptfunktion
int main(int argc, char *argv[]) {
  // Programme suchen




  // Programme sortieren, ausführen & Statusausgabe




  return 0;
} // Ende der Hauptfunktion
```

M:

```
// Hilfsfunktion für qsort
```

```
// Funktion find_executables
```

```
  // Verzeichniseinträge lesen
```

```
// Überprüfung ob Datei und Verzeichnis
```

F:

```
// Funktion run
```

R:

2) Makefile (6 Punkte)

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

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `sp-exam` 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 -D_XOPEN_SOURCE=700
```

Mk:

Aufgabe 3: Synchronisation (5 Punkte)

Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie mit Hilfe von zählenden Semaphoren das folgende Szenario korrekt synchronisiert werden kann: Zu jedem Zeitpunkt müssen so viele Threads wie möglich, maximal jedoch 4, die Funktion `threadFunc` ausführen. Dabei soll die Ergebnisausgabe in der jeweils ausgeführten Funktion `doWork` zusätzlich serialisiert werden. Ihnen stehen dabei folgende Semaphor-Funktionen zur Verfügung:

- `SEM *semCreate(int);`
- `void P(SEM *);`
- `void V(SEM *);`

Beachten Sie, dass nicht unbedingt alle freien Zeilen für eine korrekte Lösung nötig sind. Kennzeichnen Sie durch `/`, wenn Ihre Lösung in einer freie Zeile keine Operation benötigt.

Hauptthread:

```
static SEM *s;
static SEM *p;
int main(void){

-------------------------------------

-------------------------------------

  while(1) {

-------------------------------------

  startWorkerThread(threadFunc);

-------------------------------------

  }

-------------------------------------

}
```

Arbeiterthread:

```
void threadFunc(void) {

-------------------------------------

  doWork();

-------------------------------------

}

void doWork(void) {

-------------------------------------

  int result = doCalculations();

-------------------------------------

  printf("Result: %d\n", res);

-------------------------------------

}
```

Aufgabe 4: Unterbrechungen (11 Punkte)

Bei Ausnahmesituationen, wie sie bei der Ausführung eines Programms auftreten können, werden *Traps* und *Interrupts* unterschieden.

1) Beschreiben Sie diese beiden Arten:
   - Wodurch entstehen diese Ausnahmesituationen?
   - Wodurch unterscheiden sie sich?
   - Geben Sie jeweils zwei Beispiele an

(8 Punkte)

2) Beschreiben Sie was geschieht, wenn ein Prozess auf eine ausgelagerte Seite zugreift. (3 Punkte)

Aufgabe 5: Prozesszustände (7 Punkte)

Beschreiben Sie die Prozesszustände bei der Einplanung von Prozessen sowie die Ereignisse, die jeweils zu Zustandsübergängen führen (Skizze mit kurzer Erläuterung der Zustände und Übergänge).