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

- ☐ Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- ☐ Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzähler, Register, Stack).
- ☐ Der UNIX-Systemaufruf `fork(2)` lädt eine Programmdatei in einen neu erzeugten Prozess.
- ☐ Ein Prozess kann durch mehrere Programme ausgeführt werden.

b) Welche Aussage über `exec(3)` ist richtig? 2 Punkte

- ☐ `exec(3)` erzeugt einen neuen Kind-Prozess und startet darin das angegebene Programm.
- ☐ Das im aktuellen Prozess laufende Programm wird durch das angegebene Programm ersetzt.
- ☐ Dem Vater-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.
- ☐ Der an `exec(3)` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.

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

- ☐ Bei der Behandlung einer Ausnahme nach dem Wiederaufnahmemodell wird der unterbrochene Prozess neu gestartet.
- ☐ Interrupts dürfen auf keinen Fall nach dem Beendigungsmodell behandelt werden, weil überhaupt kein Zusammenhang zwischen dem unterbrochenen Prozess und dem Grund des Interrupts besteht.
- ☐ Das Beendigungsmodell sieht das Herunterfahren des Betriebssystems im Falle eines schwerwiegenden Fehlers vor.
- ☐ 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.

d) Gegeben seien die folgenden Präprozessor-Makros: 2 Punkte

```
#define ADD(x, y) x + y
#define SUB(x, y) x - y
```

Was ist das Ergebnis des folgenden Ausdrucks? `SUB(3, ADD(1, 4)) * 2`

- ☐ 10
- ☐ -4
- ☐ -7
- ☐ 12

e) Welche Aussage zu Zeigern ist richtig? 2 Punkte

- ☐ Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden.
- ☐ Die Übergabesemantik für Zeiger als Funktionsparameter ist call-by-reference.
- ☐ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.
- ☐ Zeiger vom Typ `void*` existieren in C nicht, da solche „Zeiger auf Nichts“ keinen sinnvollen Einsatzzweck hätten.

f) Was versteht man unter virtuellem Speicher? 2 Punkte

- ☐ 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.
- ☐ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.
- ☐ Virtueller Speicher kann dynamisch zur Laufzeit von einem Programm erzeugt werden.
- ☐ Unter einem virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.

g) Welche Aussage zu Semaphoren ist richtig? 2 Punkte

- ☐ Die P-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- ☐ Die V-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- ☐ Die V-Operation eines Semaphors kann ausschließlich von einem Thread aufgerufen werden, der zuvor mindestens eine P-Operation auf dem selben Semaphor aufgerufen hat.
- ☐ Ein Semaphor kann nur zur Signalisierung von Ereignissen, nicht jedoch zum Erreichen gegenseitigen Ausschlusses verwendet werden.

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

2 Punkte

- ☐ Das Betriebssystem erkennt die ungültige Adresse bei der Weitergabe des Befehls an die CPU (partielle Interpretation) und leitet eine Ausnahmebehandlung ein.
- ☐ Der Compiler erkennt die problematische Code-Stelle und generiert Code, der zur Laufzeit bei dem Zugriff einen entsprechenden Fehler auslöst.
- ☐ 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.

i) Der Speicher eines UNIX-Prozesses ist in Text-, Daten- und Stack-(Stapel-)Segment untergliedert. Welche Aussage zur Platzierung von Daten in diesen Segmenten ist richtig?

2 Punkte

- ☐ Bei einem Aufruf von `malloc(3)` wird das Stack-Segment dynamisch erweitert.
- ☐ Alle globalen Variablen werden im Stack-Segment abgelegt.
- ☐ Der Code von Funktionen wird zusammen mit den Variablen der Funktion im Stack-Segment abgelegt.
- ☐ Lokale Variablen der Speicherklasse static liegen im Daten-Segment.

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 zu UNIX-Dateisystemen sind richtig?

4 Punkte

- ○ Auf jedes Verzeichnis verweisen immer mindestens zwei *hard links*.
- ○ Ein Dateikopf ist eine Verwaltungsstruktur, die vorne in der Datei gespeichert wird.
- ○ Der Name einer Datei wird getrennt von ihrem Dateikopf (Inode) gespeichert.
- ○ Nach dem Löschen eines Dateikopfes (Inode) und der dazugehörigen Datenblöcke ist es möglich, dass weiterhin *hard links* auf den Inode verweisen.
- ○ In einem Namensraum mit hierarchischer Struktur ist die Verwendung von gleichen Namen in unterschiedlichen Kontexten möglich.
- ○ Obwohl eine Datei gelöscht wurde, kann es *symbolic links* geben, die noch auf sie verweisen.
- ○ Zur Anzeige des Inhaltes einer Datei ist es notwendig, das Leserecht auf dem übergeordneten Verzeichnis zu besitzen.
- ○ Beim Anlegen einer Datei wird die maximale Größe festgelegt. Wird sie bei einer Schreiboperation überschritten, wird ein Fehler gemeldet.

Aufgabe 2: sp-exam (46 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: Prozesse (14 Punkte)

1) 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). (7 Punkte)

2) Programmausführungen können durch Ausnahmen jederzeit unterbrochen werden. Sie haben in der Vorlesung zwei Arten dieser Programmunterbrechungen kennengelernt. Nennen Sie diese, sowie mindestens zwei Eigenschaften, in denen diese Arten sich unterscheiden. (3 Punkte)

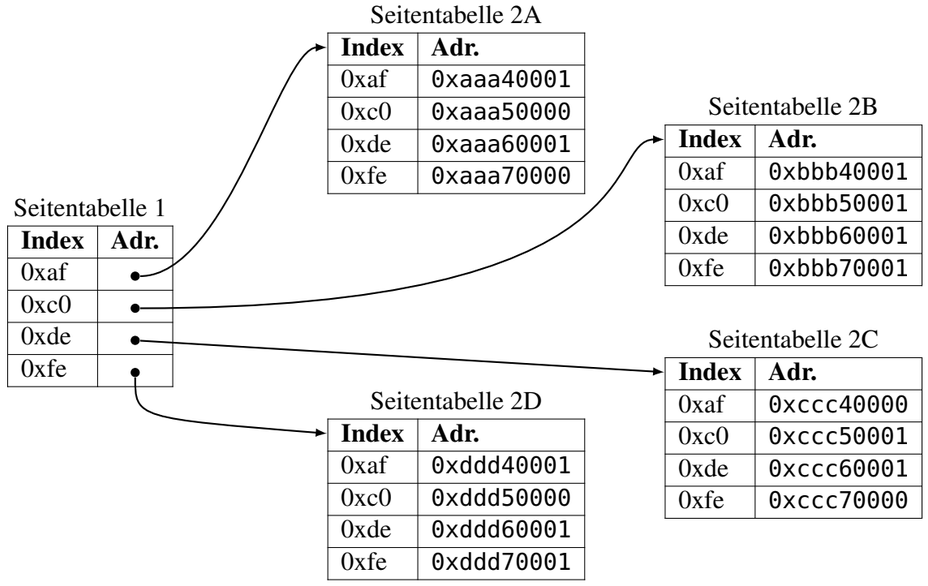
3) Das folgende Programmstück zeigt einen Ausschnitt aus dem Code eines Produzenten. Ordnen Sie die verwendeten Betriebsmittel *buffer* und *data* nach der Ihnen bekannten Klassifikation ein. Welche Konsequenzen in Bezug auf Koordinierungsmaßnahmen ergeben sich aus der Klassifikation für diese Betriebsmittel? (4 Punkte)

```
char *buffer;

void produce(char *data) {
  //...
  buffer = strdup(data);
  //...
  free(data);
  //...
}
```


Aufgabe 4: Paging (8 Punkte)

Gegeben sei unten dargestellte Hierarchie zweistufiger Seitentabellen. Die Adresslänge des genutzten Systems sei 32 Bit, die Größe einer Seite 64 Kibibyte. Für die Indizierung der zweistufigen Abbildung werden pro Stufe 8 Bit genutzt. Das niederwertigste Bit eines Seitentabelleneintrags fungiert als Anwesenheitsbit: Ist es auf 1 gesetzt, ist die Seite anwesend.



1) Bestimmen Sie die **reale Adresse** zur logischen Adresse `0xc0deaffe`. Geben Sie hierbei die zur Bestimmung notwendigen Zwischenschritte stichpunktartig an! (4 Punkte)

2) Was passiert in Hardware und im Betriebssystem bei einem Zugriff auf die Adresse `0xaffec0de`? (2 Punkte)

3) Wieso ist es nicht möglich, den Ihnen bekannten Systemaufruf `stat(2)` analog zu der Funktion `readdir(3)` mit folgender Schnittstelle zu implementieren:
`struct stat *stat(const char *path)`? Welches Verhalten ließe sich bei einem Zugriff auf den zurückgelieferten Speicher beobachten? (2 Punkte)