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 zwischen Traps und Interrupts. Welche Aussage ist richtig? 2 Punkte

- ☐ Normale Ganzzahlarithmetik-Operationen können zu einem Interrupt führen.
- ☐ Bei der mehrfachen Ausführung eines unveränderten Programms mit gleicher Eingabe treten Interrupts immer an den gleichen Stellen auf.
- ☐ Traps können nicht durch Speicherzugriffe ausgelöst werden.
- ☐ Traps stehen immer in ursächlichem Zusammenhang zu der Ausführung eines Maschinenbefehls.

b) 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.

c) Welche Aussage zum Thema Betriebsarten ist richtig? 2 Punkte

- ☐ Beim Stapelbetrieb können keine globalen Variablen existieren, weil alle Daten im Stapel-Segment (Stack) abgelegt sind.
- ☐ Echtzeitsysteme findet man hauptsächlich auf großen Serversystemen, die eine enorme Menge an Anfragen zu bearbeiten haben.
- ☐ Mehrzugangsbetrieb ist nur in Verbindung mit CPU- und Speicherschutz sinnvoll realisierbar.
- ☐ Mehrprogrammbetrieb ermöglicht die simultane Ausführung mehrerer Programme innerhalb desselben Prozesses.

d) 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 benutzen kann.
- ☐ Ein Filedeskriptor beschreibt die Eigenschaften, wie Größe und Zugriffsrechte, einer Datei.
- ☐ Beim mehrfachen Öffnen ein und derselben Datei erhält ein Prozess jeweils die gleiche Integerzahl als Filedeskriptor zum Zugriff zurück.
- ☐ Filedeskriptoren sind Zeiger auf Betriebssystemstrukturen, die von den Systemaufrufen ausgewertet werden, um auf Dateien zuzugreifen.

e) Nehmen Sie an, der Ihnen bekannte Systemaufruf stat(2) wäre analog zu der Funktion readdir(3) mit folgender Schnittstelle implementiert: `struct stat *stat(const char *path);` Welche Aussage ist richtig? 2 Punkte

- ☐ Der Systemaufruf liefert einen Zeiger zurück, über den die aufrufende Funktion direkt auf eine Datenstruktur zugreifen kann, die die Dateiattribute enthält.
- ☐ Der Aufrufer muss sicherstellen, dass er den zurückgelieferten Speicher mit free(3) wieder freigibt, wenn er die Dateiattribute nicht mehr weiter benötigt.
- ☐ Ein Zugriff über den zurückgelieferten Zeiger liefert völlig zufällige Ergebnisse oder einen Segmentation fault.
- ☐ Solch eine Schnittstelle ist nicht schön, da dadurch die aufrufende Funktion auf internen Speicher des Betriebssystems zugreifen könnte.

f) Welche Aussage zum Thema Adressraumverwaltung ist richtig? 2 Punkte

- ☐ Da das Laufzeitsystem auf die Betriebssystemschnittstelle zur Speicherverwaltung zurückgreift, ist die Granularität der von malloc() zurückgegebenen Speicherblöcke vom Betriebssystem vorgegeben.
- ☐ Ein Speicherbereich, der mit Hilfe der Funktion free() freigegeben wurde, verbleibt im logischen Adressraum des zugehörigen Prozesses.
- ☐ Mit Hilfe des Systemaufrufes malloc() kann ein Programm zusätzliche Speicherblöcke von sehr feinkörniger Struktur vom Betriebssystem anfordern.
- ☐ Mit malloc() angeforderter Speicher, welcher vor Programmende nicht freigegeben wurde, kann vom Betriebssystem nicht mehr an andere Prozesse herausgegeben werden und ist damit bis zum Neustart des Systems verloren.

g) Welche Aussage über das aktuelle Arbeitsverzeichnis (Current Working Directory) trifft zu?

2 Punkte

- ☐ Jedem UNIX-Benutzer ist zu jeder Zeit ein aktuelles Verzeichnis zugeordnet.
- ☐ Besitzt ein UNIX-Prozess kein Current Working Directory, so beendet sich der Prozess mit einem Segmentation Fault.
- ☐ Mit dem Systemaufruf `chdir()` kann das aktuelle Arbeitsverzeichnis eines Prozesses durch seinen Vaterprozess verändert werden.
- ☐ Pfadnamen, die nicht mit dem Zeichen '/' beginnen, werden relativ zu dem aktuellen Arbeitsverzeichnis interpretiert.

h) Welche der folgenden Aussagen zum Thema Dateisysteme sind richtig?

2 Punkte

- ☐ In einem UNIX-Dateisystem ist es möglich, dass derselbe Inode unter mehreren Dateinamen erreichbar ist.
- ☐ UNIX-Betriebssysteme unterstützen keine Dateisysteme mit hierarchischem Namensraum.
- ☐ Symbolische Links können nur auf Dateien innerhalb des selben Dateisystems verweisen.
- ☐ Ein Inode enthält Metadaten über eine Datei: Größe, Eigentümer, Zugriffsrechte, Dateiname usw.

i) Welche der folgenden Aussagen zum Thema Prozesszustände ist korrekt?

2 Punkte

- ☐ Der Planer (*scheduler*) kann einen Prozess in den Zustand „blockiert" überführen, indem er einen anderen Prozess einlastet.
- ☐ In einem Vierkernsystem können sich maximal vier Prozesse gleichzeitig im Zustand „bereit" befinden.
- ☐ Blockierte Prozesse können nicht in den Zustand „laufend" überführt werden, solange nicht alle benötigten Betriebsmittel zur Verfügung stehen.
- ☐ In einem Vierkernsystem gibt es stets genau vier laufende Prozesse.

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 Programm:

4 Punkte

```
#include <stdlib.h>
#include <stdio.h>

#define PI 3.1415

extern int x;

int main(int argc, char *argv[]) {
    static int a;
    int b = PI;

    x = a + b;

    printf("%f\n", b);

    return EXIT_SUCCESS;
}
```

Welche der folgenden Aussagen bzgl. dieses Programms sind korrekt?

- ○ Die Variable `a` ist uninitialisiert und enthält daher einen zufälligen Wert.
- ○ `argv` ist ein Array aus Zeigern, die jeweils auf ein Array aus `char`s zeigen.
- ○ Beim Binden des Programms kann ein Fehler auftreten.
- ○ Das Programm bekommt die Anzahl der Befehlszeilenparameter in `argv` übergeben.
- ○ Die globale Variable `PI` enthält den Wert `3.1415`.
- ○ Der Inhalt der Datei *stdlib.h* wird vor dem Übersetzen an die Stelle des includes einkopiert.
- ○ An Index `0` des `argv`-Arrays liegt ein Zeiger auf den Programmnamen oder -pfad.
- ○ Der Aufruf von `printf` in Zeile 14 gibt den Wert `3.1415` auf `stdout` aus.

Aufgabe 2: COUCH - COncUrrent Command Host (45 Punkte)

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

Schreiben Sie ein Programm *COUCH*, 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 übergebenen werden.

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

```
./couch 2 "sleep 5" "ls -ash /" "ps aux" "tar -xvf file.tar"
```

- 2 → maximal 2 Befehle werden gleichzeitig ausgeführt
- "sleep 5" → 1. Befehl
- "ls -ash /" → 2. Befehl
- "ps aux" → 3. Befehl
- "tar -xvf file.tar" → 4. Befehl

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 Prozesses gewartet werden. *Hinweis:* Es steht Ihnen frei, die Definition der Struktur um zusätzliche Einträge zu erweitern.

- Funktion `pid_t run(char *cmdline)`: Führt die übergebene Befehlszeile aus. Dazu werden das auszuführende Programm und die Parameter aus der Befehlszeile extrahiert und mithilfe einer Funktion der `exec()`-Familie ausgeführt. Tritt hierbei ein Fehler auf, dann wird eine aussagekräftige Fehlermeldung ausgegeben und mit der Ausführung des nächsten Befehlt fortgefahren. 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 (Exitcode / Signalnummer) und die Ausführdauer aus.

Hinweis: Nutzen Sie die Funktion `time()` (siehe Manpage) zur Bestimmung der Ausführdauer.

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>


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

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

struct process {
    pid_t  pid;     // PID des Prozesses

    // Eigene Mitglieder






};


// Makros, Funktionsdeklarationen, globale Variablen
```

```
// Funktion main()

    // 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 (8 Punkte)

Schreiben Sie ein Makefile, welches die Targets `all` und `clean` unterstützt. Ebenfalls soll ein Target `couch` unterstützt werden, welches das Programm `couch` baut. Greifen Sie dabei stets auf Zwischenprodukte (z.B. couch.o) zurück.

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `couch` 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
```

Mk:

Aufgabe 3: Ausnahmen (11 Punkte)

1) Nennen Sie die zwei Modelle der Ausnahmebehandlung. Wie unterscheiden sich diese Modelle? Nennen Sie für jedes der Modelle je einen Auslösungsgrund. (6 Punkte)

2) Sie führen untenstehenden Code auf einem x86-Linux-System aus. (5 Punkte)

```c
int *ptr = NULL;
*ptr = 42;
```

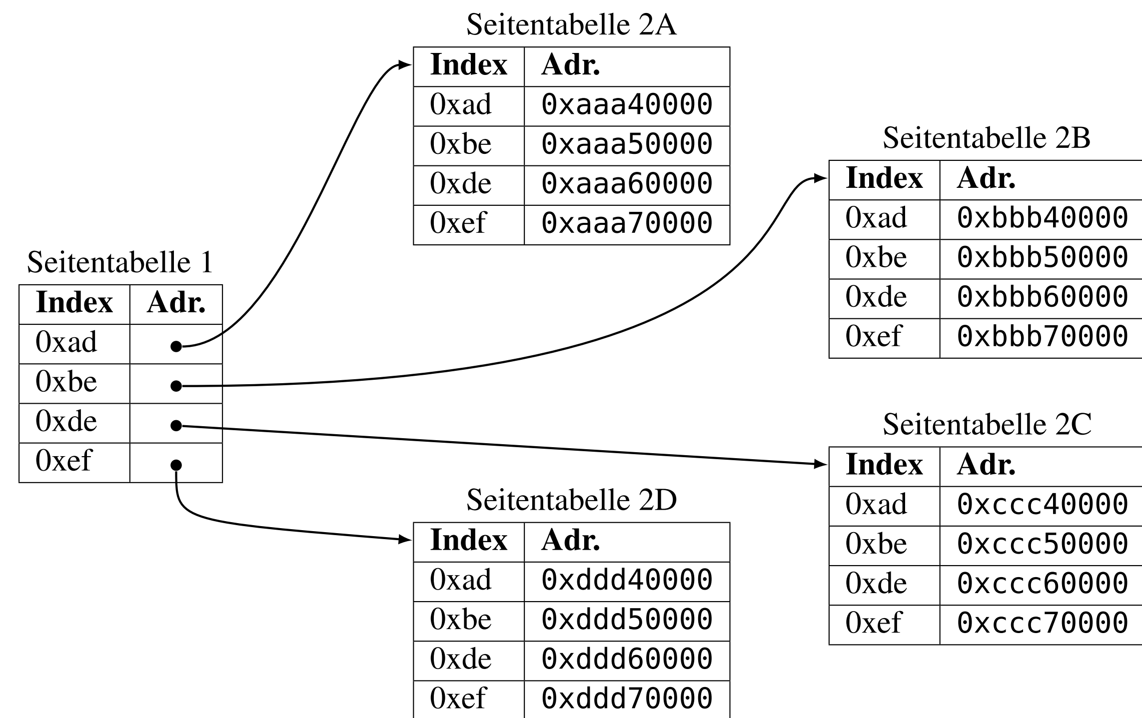
a) Was für ein Fehler tritt auf? Warum tritt dieser Fehler auf? (2 Punkte)

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

c) Nach welchem Ausnahmebehandlungsmodell wird dieser Fehler (durch das Betriebssystem) behandelt? (1 Punkt)

Aufgabe 4: Paging (12 Punkte)

Gegeben sei unten dargestellte Hierarchie zweistufiger Seitentabellen. Die Adresslänge des genutzten Systems sei 32 Bit, die Größe einer Seite 64 kByte. Für die Indizierung der zweistufigen Abbildung werden pro Stufe 8 Bit genutzt.



Seitentabelle 1

Index	Adr.
0xad	→ Seitentabelle 2A
0xbe	→ Seitentabelle 2B
0xde	→ Seitentabelle 2C
0xef	→ Seitentabelle 2D

Seitentabelle 2A

Index	Adr.
0xad	0xaaa40000
0xbe	0xaaa50000
0xde	0xaaa60000
0xef	0xaaa70000

Seitentabelle 2B

Index	Adr.
0xad	0xbbb40000
0xbe	0xbbb50000
0xde	0xbbb60000
0xef	0xbbb70000

Seitentabelle 2C

Index	Adr.
0xad	0xccc40000
0xbe	0xccc50000
0xde	0xccc60000
0xef	0xccc70000

Seitentabelle 2D

Index	Adr.
0xad	0xddd40000
0xbe	0xddd50000
0xde	0xddd60000
0xef	0xddd70000

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

2) Was passiert, wenn ein Prozess auf eine ausgelagerte Seite zugreift? (2 Punkte)

3) Zeichnen Sie den Aufbau einer Hierarchie von zweistufigen Seitentabelle für die Übersetzung der unten angegebenen Adressen. (6 Punkte)

Analog zu Teilaufgabe 1 wird ein System mit 32 Bit-Zeigern und 64 kByte großen Seiten angenommen, pro Indizierung werden pro Stufe 8 Bit genutzt.

Nutzen Sie hierfür die untenstehende Vorgabe. Dabei sollen die zur Übersetzung notwendigen Einträge für die folgenden Adressen (je eine Seite) eingezeichnet werden:

Logische Adresse	Reale Adresse
0x12340000	0x11110000
0x12660000	0x11350000
0xabcd0012	0x22220012

Seitentabelle 1

Index	Adr.