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 zu Systemaufrufen ist richtig? 2 Punkte

- ☐ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.
- ☐ Das Betriebssystem greift mit Hilfe von Systemaufrufen auf den Adressraum der Benutzerebene zu.
- ☐ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht ausführen dürfte.
- ☐ Durch einen Systemaufruf wechselt das Betriebssystem von der Systemebene auf die Benutzerebene, um unprivilegierte Operationen ausführen zu können.

b) Ein laufender Prozess wird in den Zustand *blockiert* überführt. Welche Aussage passt zu diesem Vorgang? 2 Punkte

- ☐ Der Prozess wartet auf Eingaben von der Tastatur.
- ☐ Der Zustandsübergang ist nur möglich, wenn sich der Prozess vorher im Zustand *bereit* befunden hat.
- ☐ Der Scheduler bewirkt, dass der laufende Prozess durch einen anderen verdrängt wird.
- ☐ Der Prozess hat einen Seitenfehler für eine Seite, die bereits in die Liste freier Seiten eingetragen aber noch im Hauptspeicher vorhanden ist.

c) Gegeben ist das folgende Codefragment. Welche der folgenden Definition von `buffer` stellt sicher, dass kein Überlauf entsteht ohne jedoch Speicherplatz zu verschwenden? 2 Punkte

```
const char *path = "path";
char to[] = "to";
sprintf(buffer, "%s/%s/file", path, to);
```

- ☐ `char buffer[strlen(path)+strlen(to)+strlen("//file") + 1];`
- ☐ `char buffer[sizeof(path)+sizeof(to)+sizeof("//file")];`
- ☐ `char buffer[sizeof(path)+sizeof(to)+sizeof("%s/%s/file")];`
- ☐ `char buffer[strlen(path)+strlen(to)+strlen("%s/%s/file")];`

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

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

e) Man unterscheidet zwischen den Begriffen Programm und Prozess. Welcher der folgenden Aussagen zu diesem Thema ist richtig? 2 Punkte

- ☐ Der Binder erzeugt aus einem oder mehreren Programmteilen einen Prozess.
- ☐ Der UNIX-Systemaufruf `exec(2)` sorgt dafür, dass zwei Prozesse eines Programms ausgeführt werden.
- ☐ Durch Threads kann ein Prozess mehrere Programme ausführen.
- ☐ Der UNIX-Systemaufruf `fork(2)` sorgt dafür, dass ein Programm in zwei Prozessen ausgeführt wird.

f) Welche Aussagen über Variablen und Parameter in C-Programmen ist richtig? 2 Punkte

- ☐ Auf globale „static“ Variablen kann nur innerhalb des selben Moduls über ihren Namen zugegriffen werden.
- ☐ Zugriffe auf uninitaliserte Variablen lösen automatisch einen Trap aus.
- ☐ Variablen der Speicherklasse „automatic“ werden immer mit dem Wert 0 initialisiert.
- ☐ Durch Zeiger lässt sich die Call-by-Value Semantik in C abbilden.

g) In jedem Verzeichnis eines UNIX-Dateissytem gibt es Verzeichnisse mit bestimmten vorgegebenen Namen. Welche der Aussagen ist richtig? 2 Punkte

- ☐ Das Verzeichnis „ .“ verweist auf das aktuelle Verzeichnis des Prozesses. Das Verzeichnis „ ..“ verweist auf dessen übergeordnetes Verzeichnis.
- ☐ Das Verzeichnis „ ..“ verweist auf das Wurzelverzeichnis; das Verzeichnis „ .“ verweist auf das aktuelle Arbeitsverzeichnis.
- ☐ Das Verzeichnis „ .“ verweist auf das Verzeichnis, in dem der Einrag steht; das Verzeichnis „ ..“ auf das im Pfad übergeordnete Verzeichnis.
- ☐ Das Verzeichnis „ ..“ verweist auf das Verzeichnis, in dem der Eintrag steht. Das Verzeichnis „ .“ wird ignoriert.

h) Der Speicher eines UNIX-Prozesses ist konzeptionell in Text-, Daten- Stack-(Stapel-)Segment untergliedert. Welche Aussage zur Platzierung von Teilen eines C-Programms in diesen Segmenten ist richtig?

2 Punkte

- ☐ Statisch lokale Variablen liegen im Daten-Segment
- ☐ Durch den Aufruf von `malloc(3p)` wird Platz im Stack-Segment reserviert.
- ☐ Globale Variablen werden zu Beginn im Stack-Segment initialisiert.
- ☐ Dynamisch allozierte Zeichenketten liegen im Text-Segment

i) Welche der folgenden Antwortmöglichkeiten beschreibt ein Attribut eines Index-knoten („inode“) in einem UNIX-Dateisystem?

2 Punkte

- ☐ Name der Datei
- ☐ Anzahl der symbolischen Verweise („symbolic links“)
- ☐ Übergeordnetes Verzeichnis
- ☐ Anzahl der festen Verweise („hard links“)

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 Aussagen über Prozesszustände sind richtig?

4 Punkte

- ○ Es kann mehr Prozesse im Zustand *bereit* geben als Rechenkerne im System.
- ○ Ein Prozess kann nur durch seine eigene Aktivität vom Zustand *laufend* in den Zustand *blockiert* überführt werden.
- ○ Findet in einem Monoprozessor-System zu einem Zeitpunkt keine Prozessum-schaltung statt und kein Prozess befindet sich im Zustand *laufend*, so ist auch kein Prozess im Zustand *bereit*.
- ○ Ein Prozess, der sich im Zustand *blockiert* befindet, kann diesen Zustand durch einen Aufruf der Funktion `fork(3p)` wieder verlassen.
- ○ Durch das Wegfallen der Blockadebedingung geht der Prozess direkt in den Zustand *laufend* über.
- ○ Beendet sich ein Prozess, wird er vom Betriebssystem in den Zustand *blockiert* überführt.
- ○ Ein Prozess kann sich nicht selbst in den Zustand *beendet* überführen.
- ○ Pro Rechenkern in einem System befindet sich zu jedem Zeitpunkt maximal ein Prozess im Zustand *laufend*.

Aufgabe 2: Testzentrum (45 Punkte)

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

Ihre Aufgabe ist es, ein **Testzentrum** zu programmieren, dass es ermöglichen soll, mehrere Tests parallel auf verschiedenen Proben auszuführen. Dazu wird als erster Parameter ein Verzeichnis angegeben, das rekursiv nach gültigen Proben durchsucht werden soll. Alle weiteren Parameter werden als Pfade zu den eigentlichen Test-Programmen interpretiert. Wird nicht mindestens ein Test-Programm angegeben, so soll das Testzentrum eine entsprechende Fehlermeldung ausgeben.

Für jede gültige Probe - das ist jede reguläre Datei, die den Test `is_valid_sample` besteht - wird ein Arbeiterfaden gestartet (Funktion `handle_file`), um die Test-Programme darauf auszuführen.

Beispiel eines Aufrufs und einer möglichen Ausgabe:

```
$ ./testzentrum /tests/probes /tests/test1 /tests/test2
/tests/test1 /tests/probes/rabenstein/j status: 0
/tests/test1 /tests/probes/nguyen/d     status: 0
/tests/test2 /tests/probes/nguyen/d     status: 42
/tests/test2 /tests/probes/rabenstein/j error: 12
SUMMARY: success:2 failure:1 error:1
```

Implementieren Sie die folgenden Funktionen:

`void *handle_file(void *arg)`
: Einstiegsfunktion eines Arbeiterfadens, der Name der Probe wird als Parameter übergeben. Die Funktion führt nacheinander alle Tests auf der übergebenen Probe durch. Jeder Test wird in einem Kindprozess gestartet und erhält den Pfad der Probe als Parameter. Startet ein Faden einen Prozess, so wird auch nur dieser Faden im neuen Prozess ausgeführt. Der Status des beendeten Kindprozesses wird mit der Funktion `report_status` ausgegeben. Über alle Arbeiterfäden hinweg, dürfen nie mehr als 4 (`MAX_PARALLEL`) Tests aktiv sein. Die Zahl der Kindprozesse muss dementsprechend limitiert werden. Hierzu soll die vorgegebene Semaphor-Implementierung verwendet werden. Die `main`-Funktion merkt sich nicht die Referenzen auf die Fäden. Deshalb müssen die Fäden dafür sorgen, dass bei ihrerer Beendigung ihre Ressourcen automatisch freigegeben werden.

`unsigned handle_dir(const char *path)`
: Für jede reguläre Datei im Verzeichnis `path`, die den Test durch `is_valid_sample` besteht, wird ein Arbeiterfaden gestartet, der die Funktion `handle_file` darauf ausführt. Unterverzeichnisse werden rekursiv bearbeitet. Die Funktion gibt die Anzahl der gestarteten Arbeiterfäden zurück.

`int main(int argc, char *argv[])`
: Wertet zu Beginn die übergebenen Parameter aus und initialisiert die globalen Variablen. Falls ein Verzeichnis mit Proben sowie mindestens ein Testprogramm übergeben wurden, so wird mit der Suche von Proben (`handle_dir`) begonnen. Bevor eine Zusammenfassung der Ergebnisse (`report_summary`) ausgegeben wird, muss sichergestellt sein, dass alle gestarteten Arbeiterfäden ihre Tests abgeschlossen haben. Verwenden Sie dazu ebenfalls die vorgegebene Semaphor-Implementierung. Anschließend werden eventuell noch belegte Ressourcen freigegeben.

Hinweise:

- Fehler außerhalb von `main` sollen nicht zum Abbruch des Programms führen, sondern mit `report_error` gemeldet werden.
- Tritt ein Fehler ohne Bezug zu einem Test auf, so soll die fehlgeschlagene Funktion als Test angegeben werden.
- Sie können davon ausgehen, dass der Wertebereich der Semaphore immer ausreicht.
- Für die zu implementierenden Funktionen sind keine Vorausdeklaration nötig.

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

```c
/*****************************************************\
|* Alle Funktionen auf dieser Seite sind vorgegeben *|
|* und müssen *nicht* selbst implementiert werden!  *|
\*****************************************************/
#include <dirent.h>
#include <errno.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#define MAX_PARALLEL  4

static void die(const char *msg) {
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(EXIT_FAILURE);
}

/* Meldet/Registriert den Fehler @error beim Erzeugen
 * des Kindes von @test auf @sample */
static void report_error(const char *test, const char *sample, int error);

/* Meldet/Registriert den Status @status des Kindprozesses
 * für die Ausführung von @test auf @sample */
static void report_status(const char *test, const char *sample, int status);

/* Schließt die Sammlung von Testergebnissen über report_error
 * und report_status ab und gibt eine Übersicht/Statistik zu
 * den Ausgeführten Tests aus */
static void report_summary(void);

/* Überprüft ob der Status einer Datei die Bedingungen für eine
 * gültige Probe erfüllt. Falls alle Bedingungen erfüllt sind, wir die
 * Funktion einen Wert ungleich 0 zurückliefern. */
static int is_valid_sample(struct stat *st);

/* Seamphoren-Modul wie aus der Übung bekannt */
typedef struct SEM SEM;
SEM *semCreate(int);
void semDestroy(SEM *);
void P(SEM *);
void V(SEM *);
```

```
// Globale Variablen und Definitionen

G:

int main(int argc, const char *argv[]) {
  // Initalisierung und Parametercheck

  // Proben in Verzeichnis suchen und Tests starten

  // Aufräumen

}
```

M:


```
static void *handle_file(void *arg) {

}
```

F:

```
static unsigned handle_dir(const char *dirname) {

  // Verzeichniseinträge auslesen
```



```
    // Rekursiver Abstieg

    // Faden für Testdatei starten

}
```

D:

Aufgabe 3: Adressräume (11 Punkte)

1) Beschreiben Sie die drei Begriffe *realer*, *logischer* und *virtueller* Adressraum. (7 Punkte)

2) Nennen und beschreiben Sie eine Möglichkeit, wie zur Laufzeit eine Abbildung auf den realen Adressraum vorgenommen werden kann. (4 Punkte)

Aufgabe 4: Fehler und Ausnahmen (12 Punkte)

Gegeben ist folgende C-Funktion, die Programmierfehler enthält.

```
unsigned *create_array(unsigned size) {
    int *array = calloc(size, sizeof(*array));
    for (unsigned i = 0; i <= size; ++i)
        array[i] = i;
    return array;
}
```

1) Welcher der Fehler wird -wenn er auftritt- zuverlässig von der Hardware erkannt? (1 Punkt)

2) Welche Hardwarekomponente ist dafür verantwortlich und wie wird die Ausnahme dem Betriebssystem signalisiert? (2 Punkte)

3) Aunahmesituationen lassen sich in die Kategorien *Trap* und *Interrupt* einteilen. (6 Punkte)

a) Beschreiben Sie, wodurch Ausnahmesituationen der einzelnen Kategorien entstehen. (2 Punkte)

b) Geben Sie je ein Beispiel pro Kategorie. (2 Punkte)

c) Nennen Sie zwei Eigenschaften, in denen sich die Kategorien unterscheiden. (2 Punkte)

4) Ein weiterer Fehler kann nicht so zuverlässig durch die Hardware entdeckt werden. Nennen Sie die fehlerhafte Stelle und begründen Sie mit einem Beispiel weshalb der Fehler von der Hardware unerkannt bleiben kann. (3 Punkte)