Aufgabe 1.1: Einfachauswahl-Fragen (22 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 der folgenden Aussagen zum Thema Adressräume ist richtig? 2 Punkte

- ❏ Der virtuelle Adressraum eines Prozesses kann nie größer sein als der physikalisch vorhandene Arbeitsspeicher.
- ❏ Die maximale Größe des virtuellen Adressraums kann unabhängig von der verwendeten Hardware frei gewählt werden.
- ❏ Virtuelle Adressräume sind Voraussetzung für die Realisierung logischer Adressräume.
- ❏ Logische Adressräume bieten Schutz vor Berechnungsfehlern.

b) 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

- ❏ Alle lokalen Variablen werden im Stack-Segment abgelegt.
- ❏ Variablen der Speicherklasse *static* liegen im Daten-Segment.
- ❏ Bei einem malloc-Aufruf wird das Stack-Segment dynamisch erweitert.
- ❏ Dynamisch allozierte Zeichenketten liegen im Text-Segment.

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

- ❏ Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzähler, Register, Stack).
- ❏ Der Compiler erzeugt aus mehreren Programmteilen (Module) einen Prozess.
- ❏ Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- ❏ Ein Prozess kann mit Hilfe von Threads mehrere Programme gleichzeitig ausführen.

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

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

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

- ❏ Das Wiederaufnahmemodell ist für Interrupts und Traps gleichermaßen geeignet.
- ❏ Das Beendigungsmodell ist für Interrupts und Traps gleichermaßen geeignet.
- ❏ Bei der Behandlung einer Ausnahme nach dem Wiederaufnahmemodell wird der unterbrochene Prozess neu gestartet.
- ❏ Das Beendigungsmodell sieht das Herunterfahren des Betriebssystems im Falle eines schwerwiegenden Fehlers vor.

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

- ❏ Bei RAID 5 liegen die Paritätsinformationen auf einer dedizierten Platte.
- ❏ Bei RAID 4 werden alle im Verbund beteiligten Platten gleichmäßig beansprucht.
- ❏ Bei RAID 1 wird beim Lesen ein Geschwindigkeitsvorteil erzielt.
- ❏ Bei RAID 0 führt der Ausfall einer der beteiligten Platten nicht zu Datenverlust.

g) Welche Aussage zum Thema Seitenersetzungsstrategien ist richtig? 2 Punkte

- ❏ Bei der Ersetzungsstrategie LRU wird diejenige Seite ersetzt, seit deren letztem Zugriff die geringste Zeit vergangen ist.
- ❏ Bei der Ersetzungsstrategie LFU ist der Zeitpunkt des letzten Zugriffes das ausschlaggebende Kritierum für die Ersetzung einer Seite.
- ❏ Die Ersetzungsstrategie OPT ist in der Praxis nur schwer realisierbar, weil Wissen über das zukünftige Verhalten des Gesamtsystems notwendig ist.
- ❏ Die Ersetzungsstrategie FIFO eignet sich vor allem für Programme, die nichtsequentielle Speicherzugriffsmuster aufweisen.

h) Welche Aussage zu Prozesszuständen ist richtig? 2 Punkte

- ❏ Ein Prozess kann nur durch seine eigene Aktivität vom Zustand *laufend* in den Zustand *blockiert* überführt werden.
- ❏ Ein Prozess, der sich im Zustand *gestoppt* befindet, kann sich selbst durch den Aufruf der Funktion `fork()` in den Zustand *bereit* überführen.
- ❏ Ein Prozess, der sich im Zustand *laufend* befindet, kann im Rahmen der mittelfristigen Einplanung in den Zustand *schwebend-laufend* überführt werden.
- ❏ Ein Prozess kann sich nicht selbst in den Zustand *beendet* überführen.

i) Welche Aussage zu nicht-blockierender Synchronisation ist richtig? 2 Punkte

- ❏ Verfahren zur nicht-blockierenden Synchronisation greifen auf spezielle Prozessor-Instruktionen wie CAS zurück und lassen sich daher nicht in reinem C99 implementieren.
- ❏ Nicht-blockierende Synchronisationsverfahren setzen besondere Unterstützung durch das Betriebssystem voraus.
- ❏ Auch bei nicht-blockierender Synchronisation kann es zu Verklemmungen (*Deadlocks*) kommen.
- ❏ Nicht-blockierende Verfahren sind intransparent für den Scheduler und daher anfällig für Prioritätsverletzung und -umkehr.

j) In welcher der folgenden Situationen wird ein Prozess vom Zustand *laufend* in den Zustand *bereit* übeführt? 2 Punkte

- ❏ Der Prozess greift lesend auf eine Datei zu und der entsprechende Datenblock ist noch nicht im Hauptspeicher vorhanden.
- ❏ Der Scheduler bewirkt, dass der Prozess durch einen anderen Prozess verdrängt wird.
- ❏ Der Prozess ruft eine P-Operation auf einen Semaphor auf, welcher den Wert 0 hat.
- ❏ Der Prozess ruft die Bibliotheksfunktion `exit(3)` auf.

k) Welche Aussage zum Aufbau einer Kommunikationsverbindung zwischen einem Client und Server über eine Socket-Schnittstelle ist richtig? 2 Punkte

- ❏ Der Server erzeugt einen Socket und ruft anschließend `listen(2)` auf - der Client kann daraufhin mit `connect(2)` eine Verbindung herstellen und sofort Daten übertragen.
- ❏ Der Client kann erst `connect(2)` aufrufen, nachdem der Server `accept(2)` aufgerufen hat - vorher würde der Verbindungsversuch mit der Meldung "connection rejected" abgewiesen werden.
- ❏ Der Server signalisiert durch den Aufruf von `connect(2)`, dass er zur Annahme von Verbindungen bereit ist; ein Client kann dies durch `accept(2)` annehmen.
- ❏ Der Server richtet am Socket eine Warteschlange für ankommende Verbindungen ein und kann dann mit `accept(2)` eine konkrete Verbindung annehmen. `accept(2)` blockiert so lange die Warteschlange leer ist.

Aufgabe 1.2: Mehrfachauswahl-Fragen (8 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 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

- ❍ Zur Anzeige des Inhaltes einer Datei ist es notwendig, das Leserecht auf dem übergeordneten Verzeichnis zu besitzen.
- ❍ Für das Löschen einer Datei sind die Rechte-Informationen im Dateikopf (*Inode*) der Datei irrelevant.
- ❍ Nach dem Löschen eines Dateikopfes (*Inode*) und der dazugehörigen Datenblöcke ist es möglich, dass weiterhin *hard links* auf den Inode verweisen.
- ❍ Ein Pfadname, der nicht mit einem '`/`'-Zeichen beginnt, wird relativ zum Home-Verzeichnis des Benutzers interpretiert.
- ❍ Innerhalb eines UNIX-Dateisystembaumes können die Inhalte mehrerer Festplatten eingebunden sein.
- ❍ In einem Namensraum mit hierarchischer Struktur ist die Verwendung von gleichen Namen in unterschiedlichen Kontexten möglich.
- ❍ Der Name einer Datei wird getrennt von ihrem Dateikopf (*Inode*) gespeichert.
- ❍ Auf jedes Verzeichnis verweisen immer mindestens zwei *hard-links*.

b) Welche der folgenden Aussagen zum Thema Speicherverwaltung sind richtig? 4 Punkte

- ❍ Bei Segmentierung kann externe Fragmentierung des Arbeitsspeichers auftreten.
- ❍ Bei der Adressumsetzung in einem seitennummerierten Adressraum kann ein Zugriffsfehler auftreten, obwohl das *present-bit* im Seitendeskriptor gesetzt ist.
- ❍ Bitkarten eignen sich zur Verwaltung von segmentierten Adressräumen besser als zur Verwaltung von seitenbasierten Adressräumen.
- ❍ Bei der Platzierungsstrategie *best-fit* muss die Freispeicherliste bei einer Allokation gegebenenfalls zweimal durchlaufen werden.
- ❍ Bei der Platzierungsstrategie *first-fit* ist die Verschmelzung von freien Speicherbereichen einfacher als bei *worst-fit*.
- ❍ Beim Buddy-Verfahren können zwei aneinandergrenzende Blöcke gleicher Größe immer verschmolzen werden.
- ❍ Der Systemaufruf `free(2)` sorgt dafür, dass die angegebene Seite im Freiseitenpuffer landet.
- ❍ Bei der Auslagerung einer Seite ist keine Anpassung des TLBs erforderlich.

Aufgabe 2: tesa (62 Punkte)

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

Schreiben Sie ein Programm `tesa` (**Te**stsuite **S**hell **A**dvanced), welches eine interaktive Shell zum Einlesen und Ausführen einer Testfolge (Testsuite) anbietet. Die unterstützten Kommandos lauten wie folgt:

`readTestsuite`: Sucht im aktuellen Arbeitsverzeichnis nach Testfällen und fügt diese der Testfolge hinzu.

`runTestsuite`: Führt die vorher eingelesene Testfolge aus und gibt zu jedem Testfall eine kurze Zusammenfassung aus.

Das Programm soll folgendermaßen strukturiert sein:

- Das Hauptprogramm liest wiederholt Zeilen von der Standardeingabe ein. Sie können davon ausgehen, dass keine der eingegebenen Zeilen mehr als 100 (Nutz-)Zeichen beeinhaltet; siehe `MAX_LINE_LEN`. Enthält die Zeile ein unbekanntes Kommando, so soll sie ignoriert werden und die Fehlermeldung "`Unknown command`" ausgegeben werden. Leere Zeilen werden ohne Fehlermeldung ignoriert.
  Ist die Eingabe ein unterstütztes Kommando, so wird die entsprechende Funktion aufgerufen.
- Kommando `readTestsuite`, implementiert durch die Funktion `void readTestsuite(void)`: Die Funktion extrahiert aus dem aktuellen Arbeitsverzeichnis die Namen aller regulären Dateien, die ausführbar (siehe Makro `S_ISEXEC`) sind, und trägt den Testfall (`struct testcase`) in die Verwaltungsstruktur der Testfolge (`struct testsuite`) ein. Sie können davon ausgehen, dass die Namen der Testfälle kürzer sind als 256 (=`MAX_PROGNAME`) Zeichen. Sollten mehr als 100 (=`MAX_TESTCASES`) Testfälle im Verzeichnis vorhanden sein, werden die überzähligen ohne Fehlermeldung ignoriert. Sollte bei der Bearbeitung in Fehler auftreten, ist eine aussagekräftige Fehlermeldung auszugeben und der Verzeichniseintrag zu ignorieren.
- Kommando `runTestsuite`, implementiert durch die Funktion `void runTestsuite(void)`: Die Funktion startet alle Testfälle der vorher eingelesenen Testfolge und wartet anschließend passiv auf die Beendigung aller Testfälle. Alle Zombies sollen sofort aufgesammelt werden.
  Durch Drücken von Strg-C (Signal `SIGINT`) kann der Benutzer die Ausführung der Testfälle vorzeitig abbrechen. Dies soll allerdings nicht zum Abburch der `tesa` führen. Nach Auftreten des Signals `SIGINT` haben die Testfälle 3 (=`TESTCASE_TIMEOUT`) Sekunden Zeit, sich zu beenden. Testfälle, die nach Ablauf dieser Frist noch nicht terminiert sind, sollen durch das Signal `SIGKILL` unverzüglich beendet werden. Zur Realisierung dieses Timeouts steht die Funktion `alarm(2)` zur Verfügung, deren genaue Funktionsweise der beigefügten Manual-Page zu entnehmen ist.
  Nachdem alle Testfälle beendet wurden, wird zu jedem Testfall eine kurze Zusammenfassung abhängig von der Art der Prozessbeendigung ausgegeben. Dies kann durch Aufruf der vorgegebenen Funktion `void printSummary(const struct testsuite *)` erfolgen.
- Die als implementiert anzunehmende Funktion `void printSummary(const struct testsuite[])` gibt für alle (Eintrag `numberOfTestcases` der `struct testsuite`) Testfälle der Testfolge den Status (Eintrag `status` in der `struct testcase`) des Kindprozesses aus.

Tipp: Die Verwendung von Funktionen der dynamischen Speicherverwaltung ist bei dieser Aufgabe nicht notwendig.

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 Anweisungen 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 <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <signal.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/wait.h>

// Konstanten
#define MAX_LINE_LEN 100+2
#define MAX_PROGNAME 256
#define MAX_TESTCASES 100
#define TESTCASE_TIMEOUT 3

// Struktur-Deklarationen
struct testcase {
   char progName[MAX_PROGNAME];
   int status;
   pid_t pid;
};

struct testsuite {
   int numberOfTestcases;
   struct testcase testcases[MAX_TESTCASES];
};

// Hilfsfunktion die
static int die(const char message[]) {
   perror(message);
   exit(EXIT_FAILURE);
}

// Prüft, ob Datei ausführbar ist
#define S_ISEXEC(mode) (((S_IXUSR|S_IXGRP|S_IXOTH)&mode) > 0 )

// Hilfsfunktion zur Ausgabe der Testfälle einer Testfolge
static void printSummary(const struct testsuite ts[]);
```

```
// Globale Variablen, Funktionsdeklarationen usw.
```

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

```
    // Allgemeine Vorbereitungen
```

I:


```
    // Zeile einlesen und Kommando interpretieren
```

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

D:

```
// Funktion readTestsuite
```


```
// Prüfen, ob Verzeichniseintrag ein Testfall ist
```

```
// Ende Funktion readTestsuite
```

E:

```
// Funktion runTestsuite
```

```
   // Testfälle starten
```


```
   // Auf die Beendigung der Testfälle warten
```

```
   // Zusammenfassung ausgeben
```

```
// Ende der Funktion runTestsuite
```

A:

```
// Signalbehandlung SIGALRM
```

```
// Signalbehandlung SIGINT
```

```
// Signalbehandlung SIGCHLD
```

S:

Aufgabe 3: (10 Punkte)

Bei virtuellen Adressräumen können Teile des Speichers auf Hintergrundspeicher ausgelagert sein.

a) Wie wird im System bei einem Speicherzugriff erkannt, dass der entsprechende Speicherbereich ausgelagert ist? (2 Punkte)

b) Was läuft im System nach dieser Erkennung ab? Beschreiben Sie die einzelnen Schritte, die im Betriebssystem abgewickelt werden, um den Speicher verfügbar zu machen, und geben Sie an, welche Prozesszustände der auslösende Prozess dabei jeweils einnimmt. (8 Punkte)

Aufgabe 4: (18 Punkte)

Zur Koordinierung von nebenläufigen Vorgängen, die auf gemeinsame Betriebsmittel zugreifen, unterscheidet man zwischen einseitiger und mehrseitiger Synchronisation.

a) Beschreiben Sie die Unterschiede zwischen einseitiger und mehrseitiger Synchronisation. (2 Punkte).

b) Betriebsmittel lassen sich in zwei Kategorien einteilen. Nennen und beschreiben Sie diese und geben Sie je zwei Beispiele. (6 Punkte)

c) Zur Koordination des Zugriffs auf Betriebsmittel können Semaphore eingesetzt werden. Skizzieren Sie in Pseudocode für die beiden Kategorien von Betriebsmitteln je ein Zugriffsmuster und dessen korrekte Koordinierung mit Hilfe eines Semaphors. Nennen Sie für jedes Zugriffsmuster ein Beispiel für ein konkretes Problem, welches dadurch gelöst wird. Geben Sie jeweils an, ob es sich um einseitige oder mehrseitige Synchronisation handelt. (10 Punkte)