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) In jedem Verzeichnis eines UNIX-Dateisystem 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 Verzeichnis, in dem der Eintrag steht. das Verzeichnis „`..`“ auf das im Pfad übergeordnete Verzeichnis.
- ☐ Das Verzeichnis „`..`“ verweist auf das Wurzelverzeichnis. Das Verzeichnis „`.`“ verweist auf das aktuelle Arbeitsverzeichnis.
- ☐ Das Verzeichnis „`..`“ verweist auf das Verzeichnis, in dem der Eintrag steht. Das Verzeichnis „`.`“ wird ignoriert.

b) Welche Aussage über das aktuelle Arbeitsverzeichnis (Current Working Directory) trifft zu? 2 Punkte

- ☐ Mit dem Systemaufruf `chdir()` kann das aktuelle Arbeitsverzeichnis eines Prozesses durch seinen Elternprozess verändert werden.
- ☐ Das aktuelle Arbeitsverzeichnis erbt der Prozess vom aktuellen Benutzer.
- ☐ Besitzt ein UNIX-Prozess kein aktuelles Arbeitsverzeichnis, so beendet sich der Prozess durch einen Segmentation Fault.
- ☐ Jedem UNIX-Prozess ist zu jeder Zeit ein aktuelles Arbeitsverzeichnis zugeordnet.

c) Was passiert auf einem x86-Linux-System, wenn ein C-Programm über einen ungültigen Zeiger versucht 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“.
- ☐ Das Betriebssystem erkennt die ungültige Adresse bei der Weitergabe des Befehls an die CPU (partielle Interpretation) und leitet eine Ausnahmebehandlung ein.
- ☐ Der Übersetzer erkennt die problematische Code-Stelle und generiert Code, der zur Laufzeit bei dem Zugriff einen entsprechenden Fehler auslöst.
- ☐ Die MMU erkennt die ungültige Adresse bei der Adressumsetzung und löst einen Trap aus.

d) In einem UNIX-Dateisystem gibt es symbolische Namen/Verweise (Symbolic Links) und feste Links (Hard Links) auf Dateien. Welche Aussage ist richtig? 2 Punkte

- ☐ Ein Symbolic Link kann nicht auf Dateien anderer Dateisysteme verweisen.
- ☐ Wird der letzte Symbolic Link auf eine Datei gelöscht, so wird auch die Datei selbst gelöscht.
- ☐ Für jede reguläre Datei existiert mindestens ein Hard Link im selben Dateisystem.
- ☐ Ein Hard Link kann nur auf Verzeichnisse, nicht jedoch auf Dateien verweisen.

e) Welche der folgenden Aussagen zu statischem bzw. dynamischem Binden ist richtig? 2 Punkte

- ☐ Statisch gebundene Programmdateien sind kleiner als dynamisch gebundene, da mehrfach genutzte Funktionen in einer gemeinsam genutzten Bibliothek (Shared Library) abgelegt werden und nicht in die ausführbare Datei kopiert werden.
- ☐ Bei dynamischem Binden können Fehlerkorrekturen in Bibliotheken leichter übernommen werden, da nur die Bibliothek selbst neu erzeugt werden muss. Programme, die die Bibliothek verwenden, müssen nicht neu kompiliert und gebunden werden.
- ☐ Bei statischem Binden werden durch den Übersetzer alle Adressbezüge vollständig aufgelöst.
- ☐ Bei dynamischem Binden müssen zum Übersetzungszeitpunkt alle Adressbezüge vollständig aufgelöst werden.

f) Welche Aussage zu einem Monoprozessor-Betriebssystem ist richtig? 2 Punkte

- ☐ Ist zu einem Zeitpunkt kein Prozess im Zustand bereit, so ist auch kein Prozess im Zustand laufend.
- ☐ Ein Prozess im Zustand blockiert muss warten, bis der laufende Prozess den Prozessor abgibt und kann dann in den Zustand laufend überführt werden.
- ☐ Es befindet sich zu jedem Zeitpunkt maximal ein Prozess im Zustand laufend.
- ☐ In den Zustand blockiert gelangen Prozesse nur aus dem Zustand bereit.

g) Wie wird in einem UNIX-Dateisystem das Attribut des Eigentümers einer Datei realisiert? 2 Punkte

- ☐ Der Benutzername des Eigentümers wird als String im Inode der Datei abgespeichert.
- ☐ Die Group-ID (GID) des Eigentümers wird als Zahl im Inode der Datei gespeichert.
- ☐ Die User-ID (UID) des Eigentümers wird als Zahl im Inode der Datei gespeichert.
- ☐ Die User-ID (UID) des Eigentümers steht im Verzeichniseintrag der Datei.

h) 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

- ☐ Lokale Variablen der Speicherklasse „`static`“ liegen im Daten-Segment.
- ☐ Dynamisch allozierte Zeichenketten liegen im Text-Segment.
- ☐ Der Code von Funktionen wird zusammen mit den Variablen der Funktion im Stack-Segment abgelegt.
- ☐ Bei einem `malloc(3)`-Aufruf wird das Stack-Segment dynamisch erweitert.

i) Welche Aussage zu Prozessen und Threads ist richtig?

2 Punkte

- ☐ Der Aufruf von `fork(2)` gibt im Elternprozess die Prozess-ID des Kindprozesses zurück, im Kindprozess hingegen den Wert 0.
- ☐ Die Veränderung von Variablen und Datenstrukturen in einem mittels `fork(2)` erzeugten Kindprozess beeinflusst auch die Datenstrukturen im Elternprozess.
- ☐ Mittels `fork(2)` erzeugte Kindprozesse können in einem Multiprozessor-System nur auf dem Prozessor ausgeführt werden, auf dem auch der Elternprozess ausgeführt wird.
- ☐ Threads, die mittels `pthread_create(3)` erzeugt wurden, besitzen jeweils einen eigenen Adressraum.

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) Ausnahmesituationen bei einer Programmausführung werden in die beiden Kategorien Trap und Interrupt unterteilt. Welche der folgenden Aussagen sind zutreffend?

4 Punkte

- ○ Bei vollständiger Kenntnis der Programmeingabe, lässt sich vorhersagen an welcher Stelle es bei der Programmausführung zu Interrupts kommt.
- ○ Traps treten immer synchron zum unterbrochenen Prozess aus. Die Prozessausführung ist ursächlich für das Auftreten des Traps.
- ○ Wird ein Prozess durch einen Interrupt unterbrochen, so ist dieser Prozess zwingend die Ursache für das Auftreten des Interrupts.
- ○ Ein Systemaufruf im Anwendungsprogramm ist der Kategorie Interrupt zuzuordnen.
- ○ Bei einem Trap wird die gerade in Bearbeitung befindliche Instruktion immer noch vollständig zu Ende ausgeführt, bevor mit der Trapbehandlung begonnen wird.
- ○ Nach der Behandlung eines Traps kann das unterbrochene Programm gegebenenfalls fortgesetzt werden.
- ○ Die Ausführung einer Ganzzahl-Rechenoperation (z.B. Addition, Division) kann zu einem Trap führen.
- ○ Die CPU sichert bei einem Interrupt einen Teil des Prozessorzustands.

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

Aufgabe 2: `unplag` (45 Punkte)

Schreiben Sie ein Programm, dass einen parallelen Plagiatscheck zwischen einer Datei (im folgenden *Pivot-Datei* genannt) und einer Menge anderer Dateien implementiert.

Als Argument erwartet das Programm den Namen der Pivot-Datei, ein Shell-Wildcard-Muster und eine nicht-leere Menge von Pfaden. Die Pfade werden parallel von mehreren Threads bearbeitet. Für Pfade, die dem Wildcard-Muster entsprechen und bei denen es sich um eine reguläre Datei handelt, wird der Plagiats-Score mithilfe der vorgegebenen Funktion `get_plagscore()` berechnet. Pfade, die mindestens eine dieser Bedingungen nicht erfüllen, haben den Plagiats-Score `-1`. Abschließend werden alle Pfade, zusammen mit ihrem Plagiats-Score, sortiert ausgegeben.

Ein beispielhafter Aufruf mit `pivot.c` als Pivot-Datei, `*.c` als Wildcard-Muster, sowie den regulären Dateien `Makefile`, `dir/cc.c`, `.dd.c`, `bb.c` und `aa.c`, wäre damit:

Aufruf: `$ ./unplag pivot.c '*.c' Makefile dir/cc.c .dd.c bb.c aa.c`
Ausgabe: `dir/cc.c 23, .dd.c 2, aa.c 0, bb.c 0, Makefile -1,`

Implementieren Sie die folgenden Funktionen:

`int main(int argc, char *argv[])` Zunächst werden die globalen Datenstrukturen initialisiert (u.a. die zur Pivot-Datei durch einen passenden Aufruf von `load_file()`) und die übergebenen Pfade in einem `struct pathscore`-Array abgelegt. Danach werden 5 (`NTHR`) Arbeiterthreads gestartet die `thread()` ausführen und jeweils einen gleich großen Anteil der Pfade bearbeiten. Gehen Sie vereinfachend davon aus, dass die Anzahl der Pfade immer ein Vielfaches von `NTHR` ist. Wurde für jeden der Pfade der Plagiats-Score bestimmt, werden diese durch `qsort(3)` mit der Vergleichsfunktion `compare_pathscores()` sortiert. Anschließend werden die Pfade zusammen mit den entsprechenden Plagiats-Scores auf der Standardausgabe als Nutzdaten ausgegeben. Abschließend werden alle allokierten Ressourcen freigegeben.

`char **load_file(const char *path, size_t *sout)` Generische Funktion zum Laden des Inhalts einer übergebenen Datei. Die Zeilen der Datei werden als String-Array zurückgegeben und die Zeilenanzahl im Ausgabeparameter `sout` gespeichert. Besteht eine Zeile (inklusive eventueller Newline) aus mehr als 1024 (`LMAX`) Zeichen, beendet die Funktion das Programm mit einer Fehlermeldung.

`void *thread(void *arg)` Erhält als Argument einen Zeiger auf die erste von diesem Thread zu bearbeitende `struct pathscore`. Der Thread prüft für jeden der ihm zugeteilten Pfade, ob diese dem Wildcard-Muster entsprechen und ob es sich um eine echte reguläre Datei handelt (kein symbolischer Link). Ist dies der Fall, wird durch einen Aufruf der vorgegebenen Funktion `int get_plagscore(char **pivot, size_t lines, const char *path)` der Plagiats-Score bestimmt und in die `struct pathscore` eingetragen.

`int compare_pathscores(const void *a, const void *b)` Die Pfade werden nach ihrem Plagiats-Score absteigend sortiert. Pfade mit dem gleichen Score werden alphabetisch aufsteigend sortiert.

Hinweise:

- Das Wildcard-Muster und der Dateityp müssen nicht für die Pivot-Datei geprüft werden.
- Die Ausgabe muss nicht mit einer Newline enden. Fehler bei der Ausgabe von Nutzdaten sollen zum Programmabbruch führen.
- Bei der Wildcard-Muster-Prüfung soll `/` und `.` keine spezielle Bedeutung zukommen.
- Der Aufruf von `get_plagscore()` erfordert keine Fehlerbehandlung.

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

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <fnmatch.h>
#include <sys/stat.h>

// Anzahl Threads
#define NTHR 5

// Maximale Zeilenlänge
#define LMAX 1024

// Vorausdeklarationen:
static char **load_file(const char *path, size_t *sout);
static void *thread(void *arg);
static int compare_pathscores(const void *a, const void *b);


// Vorgegebene Funktionen & Strukturen:

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

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

static void usage_err(const char *program_name) {
    fprintf(stderr, "Usage: %s PIVOTFILE PATTERN [PATHS...]\n",
            program_name);
    exit(EXIT_FAILURE);
}

static int get_plagscore(char **pivot, size_t lines, const char *path) {
    size_t path_lines;
    char **path_content = load_file(path, &path_lines);
    /* ..., deallokiert path_content auch wieder. */
}

struct pathscore {
    char *path;
    int score;
};
```

```
// Globale Variablen

int main(int argc, char *argv[]) {
    // Argumente prüfen.

    // Globalen Zustand initialisieren.

    // struct pathscore-Array anlegen und initialisieren.

    // Array parallel bearbeiten lassen.
```



```
    // Array sortieren und ausgeben.

    // Ressourcen freigeben.

}
```

M:

```
static char **load_file(const char *path, size_t *sout) {
```

```
}
```

L:


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

```
}
```

T:

```
static int compare_pathscores(const void *a, const void *b) {
```

```
}
```

C:

Aufgabe 3: Adressräume (15 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)

3) Warum kann es nötig werden, dass ein Benutzerprogramms seinen virtuellen/logischen Adressraum zur Laufzeit erweitert? Durch welche Schnittstellen wird dies vom Betriebssystem ermöglicht und warum werden diese dazu von normalen Benutzerprogrammen nur in Spezialfällen direkt verwendet? Welche Schnittstelle wird von Benutzerprogrammen stattdessen normalerweise verwendet? (4 Punkte)

Aufgabe 4: Synchronisation (8 Punkte)

1) Erläutern Sie das Konzept Semaphor. Welche Operationen sind auf Semaphoren definiert und welche Eigenschaften haben diese Operationen? (5 Punkte)

2) Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie mithilfe eines zählenden Semaphors das folgende Szenario korrekt synchronisiert werden kann: Zu jedem Zeitpunkt darf maximal ein Thread die Funktion `writeResult()` ausführen. Die Funktion `doWork()` soll jedoch von allen 8 Threads parallel ausgeführt werden. Ihnen stehen dabei folgende Semaphor-Funktionen zur Verfügung: (3 Punkte)

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

Kennzeichnen Sie durch „`NOP`“, wenn Ihre Lösung in einer freien Zeile keine Operation benötigt. Jede korrekt beschriftete Zeile gibt einen halben Punkt, jede falsch beschriftete einen halben Punkt Abzug. Die Teilaufgabe wird mindestens mit 0 Punkten bewertet. Fehlerbehandlung und das freigeben von Resourcen ist in der Aufgabe **nicht** notwendig.

Hauptthread:

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

-------------------------------------
  for (size_t i = 0; i < 8; i++) {

-------------------------------------
    startWorkerThread(threadFunc);

-------------------------------------
  }
}
```

Arbeiterthread:

```
void threadFunc(void) {

-------------------------------------
  int result = doWork();

-------------------------------------
  writeResult(result);

-------------------------------------
}
```