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 über lokale auto Variablen ist richtig? 2 Punkte

- ☐ Der die Variable umgebende Block definiert die Lebensdauer.
- ☐ Zeigervariablen die auf mit `malloc(3)` allokierten Speicher zeigen verlieren durch einen `free(3)`-Aufruf ihre Gültigkeit.
- ☐ Die Lebensdauer einer lokalen Variable endet mit ihrer Sichtbarkeit.
- ☐ Durch den Aufruf von `realloc(3)` kann die Lebensdauer einer Variable auf eine andere übertragen werden.

b) Prozessoren kennen oft einen Benutzermodus und einen privilegierten Betriebsmodus. Welche Aussage ist richtig? 2 Punkte

- ☐ Spezielle Befehle zur Ansteuerung externer Geräte (z.B. Festplatte) stehen nur im privilegierten Modus zur Verfügung.
- ☐ Die Verwaltung von Speicher ist nur im privilegierten Betriebsmodus möglich.
- ☐ Der Prozessor bietet für den Benutzer- und Systemmodus je einen anderen Befehlsatz an.
- ☐ Wird zwischen zwei Prozessen des gleichen Benutzers umgeschaltet, so ist kein Wechsel in den privilegierten Modus nötig.

c) 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? `2 * SUB(3, ADD(1, 4))`

- ☐ 12
- ☐ 10
- ☐ -4
- ☐ 9

d) Ein laufender Prozess wird durch den Scheduler verdrängt. Welcher Zustandsübergang findet statt? 2 Punkte

- ☐ Der Prozess wechselt vom Zustand laufend in den Zustand bereit.
- ☐ Der Prozess wechselt vom Zustand bereit in den Zustand blockiert.
- ☐ Der Prozess wechselt vom Zustand laufend in den Zustand beendet.
- ☐ Der Prozess wechselt vom Zustand laufend in den Zustand blockiert.

e) Welche der folgenden Informationen wird typischerweise in dem Seitendeskriptor einer Seite eines virtuellen Adressraums gehalten? 2 Punkte

- ☐ Zugriffsrechte (z. B. lesen, schreiben, ausführen)
- ☐ die Identifikation des Prozesses, dem die Seite zugeordnet ist
- ☐ die Position der Seite im virtuellen Adressraum
- ☐ die Zuordnung zu einem Segment (Text, Daten, Stack, ...)

f) Was versteht man unter virtuellem Speicher? 2 Punkte

- ☐ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.
- ☐ Virtueller Speicher kann größer sein als der physikalisch vorhandene Arbeitsspeicher. Gerade nicht benötigte Speicherbereiche können auf Hintergrundspeicher ausgelagert werden.
- ☐ Virtueller Speicher ist in unbegrenzter Menge vorhanden.
- ☐ Unter einem Virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.

g) Was versteht man unter einem Interrupt? 2 Punkte

- ☐ Mit einer Signalleitung wird dem Prozessor eine Unterbrechung angezeigt. Der Prozessor sichert den aktuellen Zustand bestimmter Register, insbesondere des Programmzählers, und springt eine vordefinierte Behandlungsfunktion an.
- ☐ Durch eine Signalleitung wird der Prozessor veranlasst, die gerade bearbeitete Maschineninstruktion abzubrechen.
- ☐ Eine Signalleitung teilt dem Prozessor mit, dass er den aktuellen Prozess anhalten und auf das Ende der Unterbrechung warten soll.
- ☐ Der Prozessor wird veranlasst eine Unterbrechungsbehandlung durchzuführen. Der gerade laufende Prozess kann die Unterbrechungsbehandlung ignorieren.

h) 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.
- ☐ Mittels `fork(2)` erzeugte Kindprozesse können in einem Multiprozessor-System nur auf dem Prozessor ausgeführt werden, auf dem auch der Elternprozess ausgeführt wird.
- ☐ Die Veränderung von Variablen und Datenstrukturen in einem mittels `fork(2)` erzeugten Kindprozess beeinflusst auch die Datenstrukturen im Elternprozess.
- ☐ Threads, die mittels `pthread_create(3)` erzeugt wurden, besitzen jeweils einen eigenen Adressraum.

i) Welche der folgenden Aussagen zum Thema Betriebsarten ist richtig?

2 Punkte

- ☐ 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.
- ☐ Beim Stapelbetrieb können keine globalen Variablen existieren, weil alle Daten im Stapel-Segment (Stack) abgelegt sind.
- ☐ Mehrprogrammbetrieb ermöglicht die simultane Ausführung mehrerer Programme innerhalb desselben Prozesses.

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 Programmfragment:

4 Punkte

```
static int a;
int *foo(int *y) {
    static int b;
    int *(*d)(int) = 0x56343bbd9000;
    int *e = malloc(a * 4);
    ++y;
    // ...
    return &b;
}
```

Welche der folgenden Aussagen zu den Variablen im Programm sind richtig?

- ○ Die Variable `a` wird bei Programmstart auf den Wert 0 gesetzt.
- ○ Die Änderung an `y` hat keinerlei Auswirkung auf den Aufrufer der Funktion, da C keine *Call-by-Reference*-Parameterübergabe bereitstellt.
- ○ Die Zuweisung an den Funktionszeiger `d` führt zu einem Übersetzerfehler, da ein nummerischer Wert keine Funktion darstellt.
- ○ Da `b` eine statisch lokale Variable ist, muss auch beim nebenläufigen Lesen und Schreiben keine synchronisation stattfinden.
- ○ `a` liegt im Daten-Segment.
- ○ Da statisch lokale Variablen eine Lebensdauer über die gesamte Laufzeit des Programms haben, kann auf sie mit dem `::`-Operator zugegriffen werden (z.B. `foo::b`).
- ○ Die Lebensdauer von `e` weicht von der des durch `e` referenzierten Speichers ab.
- ○ `return &b;` führt zu undefiniertem Verhalten, da hier die Adresse einer Stackvariable als Rückgabewert verwendet wird.

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

Aufgabe 2: `Faden Wanne` (45 Punkte)

1) `faden` (26 Punkte)

Schreiben Sie einen Threadpool, eine Gruppe von dauerhaft bestehenden Threads, die sich aus einer Aufgabenliste Arbeitspakete nehmen und diese bearbeiten. Dabei soll der Threadpool als eigenes Modul, also als Teil einer Bibliothek fungieren.

Folgende Funktionen sollen angeboten werden:
`int tp_init(unsigned int nt)` initialisiert den Threadpool, wobei das Argument `nt` die Anzahl der gewünschten Threads beschreibt. Zur Verwaltung wird die globale Variable `tp` verwendet.
`void *tp_routine(void *)` ist die *start_routine* der Threads, in der passiv auf neue Arbeitspakete in der Liste `tp.q` gewartet wird. Sobald ein Arbeitspaket vorhanden ist, wird es am Kopf der Liste entnommen. Ein Arbeitspaket besteht aus einem Funktionszeiger (`void (*task)(void *)`) der mit dem Zeiger `void *pkg` des Arbeitspakets als Argument aufgerufen werden muss. Nach Bearbeitung wird das Paket freigegeben.
`void tp_add(struct tp_work *work)` hängt ein Arbeitspaket ans Ende von `tp.q` ein. Dabei wird den Arbeiter-Threads durch ein Semaphor signalisiert, dass ein neuer Auftrag vorhanden ist.
`int tp_join(void)` signalisiert mit Hilfe eines bestimmten Arbeitspakets (`.task = NULL`), dass sich die Threads beenden sollen. Anschließend wartet die Funktion passiv auf die Beendigung der Threads und sammelt diese auf.

Im Fehlerfall setzen `tp_init` und `tp_join` die errno, geben `-1` zurück und geben keine Ressourcen frei.

Zur Synchronisation steht Ihnen das aus der Übung bekannte Semaphor-Modul zur Verfügung. Mit **`SEM *semCreate(int i)`** kann ein auf *i* initialisierte Semaphor erstellt werden, die mit mit **`void semDestroy(SEM *)`** wieder freigegeben wird. `semCreate` gibt im Erfolgsfall das Semaphor zurück, andernfalls `NULL` und die `errno` wird gesetzt. Mit **`void P(SEM *)`** wird das Semaphor dekrementiert und mit **`void V(SEM *)`** wird es inkrementiert.

```c
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>
#include "sem.h"
#include "tp.h"

struct tp_work { /* in tp.h */
  void *pkg;
  void (*task)(void *);
  struct tp_work *next;
};
static struct {
  struct tp_work *q; /* queue */
  SEM *m; /* mutex */
  SEM *c; /* counter */
  pthread_t *p; /* pool */
  size_t cnt;
} tp;
// Vorausdeklaration
static void * tp_routine(void *unused);
```



```c
// tp_init
```

```
// tp_add
```

```
// tp_routine
```



```
// tp_join
```

F:

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

2) `wanne` (19 Punkte)

Schreiben Sie ein Programm, das auf die zuvor beschriebene Threadpool-Bibliothek zurückgreift, um die md5-Prüfsumme mehrerer Dateien zu berechnen und auszugeben.

Das Programm erwartet eine beliebige Anzahl von Verzeichnispfaden als Aufrufargument. Der Threadpool soll dabei so viele Threads bereit stellen, wie Argumente übergeben wurden. Je Verzeichnispfad sollen in **`void make_work_pkg(char *path)`** für jede reguläre Datei ein Arbeitspaket erstellt werden, das an den Threadpool weitergereicht wird. Die Pakete bestehen aus der Funktion **`void work_routine(char *path)`** und dem Pfad zu der jeweiligen Datei. Unterverzeichnisse werden ignoriert.

Die Funktion `work_routine` berechnet mittels der Funktion **`int md5sum(char *file, char *hash)`** die md5-Prüfsumme einer Datei, welche über den Dateipfad im Argument `file` identifiziert wird. Eine Eigenschaft der md5-Prüfsumme ist, dass sie für jede Eingabe 16 Byte lang ist und geringfügige Änderungen an der Eingabe zu einer anderen Prüfsumme führen. Die Prüfsumme wird dabei in einen String in Hexadezimaldarstellung, also 33 Zeichen (inkl. dem „\0“-Zeichen), konvertiert und in dem von `hash` referenzierten Speicher abgelegt.
Das Array `blacklist` beinhaltet Prüfsummen in Form von 33 Zeichen langen Hexadezimal-Strings. Sollte die Prüfsumme einer Datei mit einer in `blacklist` übereinstimmen, ist eine Ausgabe mit der Prüfsumme und dem Pfad der Datei zu tätigen. In `blacklist_len` ist die Anzahl der Einträge des Array `blacklist` gespeichert.

Ein beispielhafter Aufruf an `wanne` und eine mögliche Ausgabe sehen aus wie folgt:

```
dust@i4sp:/proj/i4sp1> /usr/bin/wanne ./ /tmp/
e0ff8a0af309ec10709aecb77ec3bb27: /tmp/clash
```

Mit Ausnahme der in Threads ausgeführten Funktionen darf sich das Programm im Fehlerfall durch Aufruf der Funktion **`void die(char *)`** beenden.

```c
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include "tp.h"

static void die(const char *const msg) { perror(msg); exit(EXIT_FAILURE); }
static int md5sum(char *file, char *hash) { /* ... */ }
static void work_routine(char *path);
static void make_work_pkg(char *path);

static char blacklist[][33] = {
  "e0ff8a0af309ec10709aecb77ec3bb27",
  "9864796996a03bad1713c101905ec254"
};
static unsigned int blacklist_len = 2;
```



```c
// Hauptfunktion (main)
int main(int argc, char **argv) {
```

```c
// make_work_pkg
```

```
// work_routine
```

W:

Aufgabe 3: Adressräume (11 Punkte)

Gegeben sei das nachfolgende Programm. Skizzieren Sie den Aufbau des logischen Adressraums eines Prozesses, der dieses Programm ausführt.

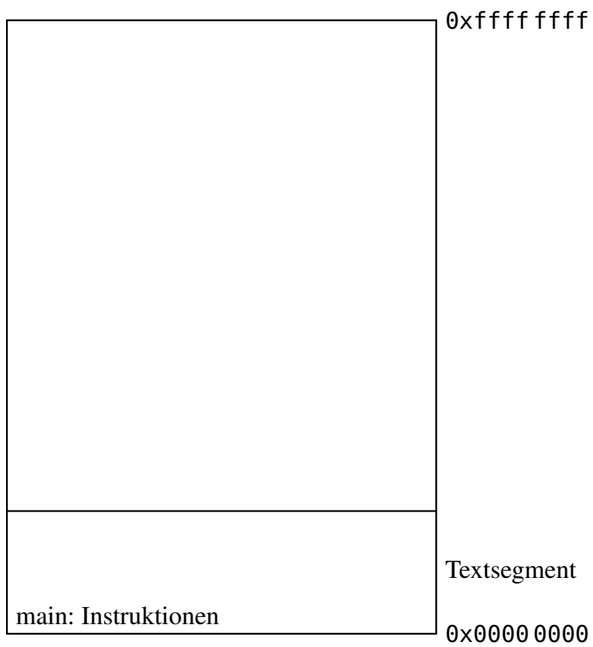
1) Tragen Sie die Segmente und deren Namen (analog zum schon vorgegebenen Textsegment) in unten stehende Zeichnung ein. Unterscheiden Sie hierbei die Bereiche zur Speicherung von initialisierten und nicht initialisierten Variablen. Zeichnen Sie für jede gültige Variable ein, wo diese ungefähr im logischen Adressraum zu finden sein wird und welchen Wert sie enthält, wenn die Programmausführung bis unmittelbar vor der **return**-Anweisung der `main`-Funktion fortgeschritten ist. Illustrieren Sie im Falle von Zeigervariablen mittels Pfeil, auf welches Datum die Variable jeweils zeigt.

Vermerken Sie zudem, in welche Richtung Segmente variabler Größe wachsen. Gehen Sie hierbei von einem x86-System aus. (8 Punkte)

```
static char *s = "Hello World\n";

static int * weird(int x) {
    int *arr = malloc(x);
    if (!arr) {exit(1);}
    return arr;
}

void main(void) {
    int g = 8;
    static int h;
    int *(*func)(int) = weird;
    int *w = func(g);
    return;
}
```



2) Beschreiben Sie was geschieht, wenn ein Prozess auf eine ausgelagerte Seite zugreift. (3 Punkte)

Aufgabe 4: Dateisystem (12 Punkte)

Gegeben ist die folgende Ausgabe des Kommandos `ls -aoRi /tmp/sp/` (rekursiv absteigende Ausgabe aller Dateien und Verzeichnisse unter `/tmp/sp/` mit Angabe der Inode-Nummer, des Referenzzählers und der Dateigröße) auf einem Linux-System.

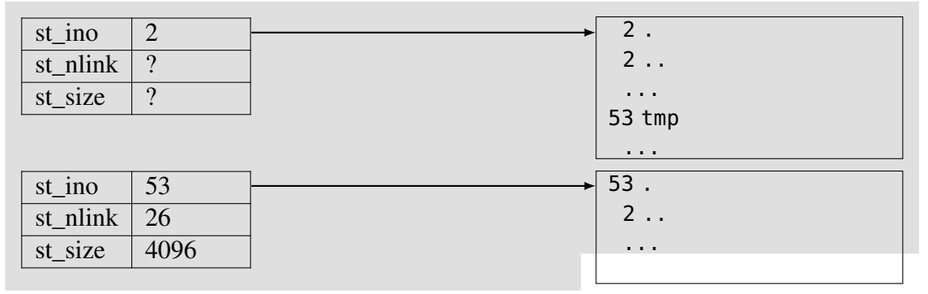
```
dust@lucy:~$ ls -aoRi /tmp/sp
/tmp/sp:
total 0
12 drwxr-xr-x  4 dust 4096 Jul 17 11:50 .
53 drwxrwxrwt 26 root  620 Jul 17 11:42 ..
43 drwxr-xr-x  2 dust 4096 Jul 17 11:25 tief
94 drwxr-xr-x  2 dust 4096 Jul 17 11:32 wichtig

/tmp/sp/tief:
total 8
43 drwxr-xr-x 2 dust 4096 Jul 17 11:25 .
12 drwxr-xr-x 4 dust 4096 Jul 17 11:50 ..
63 lrwxrwxrwx 1 dust    7 Jul 17 14:11 leere -> ./void/
20 -rw-r--r-- 2 dust 5432 Jul 17 11:23 proviant
04 lrwxrwxrwx 1 dust    2 Jul 17 11:19 tiefer -> ./

/tmp/sp/wichtig:
total 12
94 drwxr-xr-x 2 dust 4096 Jul 17 11:32 .
12 drwxr-xr-x 4 dust 4096 Jul 17 11:50 ..
36 -rw-r--r-- 1 dust 1337 Jul 17 11:31 kaffee
20 -rw-r--r-- 2 dust 5432 Jul 17 11:23 kekse
```

Ergänzen Sie im weißen Bereich die auf der folgenden Seite im grauen Bereich bereits angefangene Skizze der Inodes und Datenblöcke des Linux-Dateisystems um alle entsprechenden Informationen, die aus obiger Ausgabe entnommen werden können.

Inodes



st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	