Aufgabe 1: Ankreuzfragen (20 Punkte)

1) Einfachauswahlfragen (14 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) Ein Prozess wird in den Zustand *bereit* überführt. Welche Aussage passt zu diesem Vorgang? 2 Punkte

- ☐ Ein anderer Prozess blockiert sich an einem Semaphor.
- ☐ Der Prozess wartet auf eine Tastatureingabe.
- ☐ Der Prozess hat auf Daten von der Festplatte gewartet und die Daten stehen nun zur Verfügung.
- ☐ Der Prozess hat einen Seitenfehler für eine Seite, die aber noch im Hauptspeicher vorhanden ist.

b) Wie viele Prozesse existieren nach Ausführung des nachfolgenden Programmausschnitts? Gehen Sie davon aus, dass alle Aufrufe von `fork()` erfolgreich sind. 2 Punkte

```
fork();
fork();
fork();
```

- ☐ 8
- ☐ 3
- ☐ 4
- ☐ 1

c) In einem UNIX-Dateisystem gibt es symbolische Verweise (symbolic links) und feste Verweise (hard links). Welche der folgenden Aussagen ist richtig? 2 Punkte

- ☐ Ein „hard link“ kann nicht auf Dateien, sondern nur auf Verzeichnisse verweisen.
- ☐ Ein symbolischer Verweis kann nicht auf Dateien in anderen Dateisystemen verweisen.
- ☐ Die Anzahl der „hard links“, die auf ein Verzeichnis verweisen, hängt von der Anzahl seiner Unterverzeichnisse ab.
- ☐ Auf ein Verzeichnis verweist immer genau ein symbolischer Verweis.

d) Welche Aussage zu Speicherzuteilungsstrategien ist richtig? 2 Punkte

- ☐ Mit *best-fit* muss bei der Allokation nur einmalig durch die Freispeicherliste iteriert werden, da kein Verschnitt entsteht.
- ☐ Mit *worst-fit* können freie Speicherblöcke besonders einfach verschmolzen werden, da keine besonderen Restriktionen gelten.
- ☐ *next-fit* eignet sich zur Implementierung von `realloc(3)`, da es immer den nächst größeren Block wählt.
- ☐ *first-fit* setzt eine aufsteigende Sortierung der Freispeicherliste nach Adresse der Speicherblöcke voraus.

e) Was passiert, wenn Sie in einem C-Programm über einen ungültigen Zeiger versuchen auf Speicher zuzugreifen? 2 Punkte

- ☐ Die MMU erkennt die ungültige Adresse bei der Adressumsetzung und löst einen Trap aus.
- ☐ 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 Compiler erkennt die problematische Code-Stelle und generiert Code, der zur Laufzeit bei dem Zugriff einen entsprechenden Fehler auslöst.

f) Welche Aussagen zu virtuellem Speicher sind richtig? 2 Punkte

- ☐ Speicher der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber möglicherweise größer als der verfügbare physikalische Hauptspeicher ist.
- ☐ Virtueller Speicher ist in unbegrenzter Menge vorhanden.
- ☐ Da virtuelle Adressräume direkt durch die Hardware (MMU) unterstützt werden, können zur Laufzeit keine Kosten durch die Adressübersetzung entstehen.
- ☐ Adressierbarer Speicher, in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.

g) Welche Aussage zu Prozesszuständen ist richtig? 2 Punkte

- ☐ Beendet sich ein Prozess, wird er vom Betriebssystem vom Zustand bereit in den Zustand *blockiert* überführt.
- ☐ Je Rechenkern kann immer nur ein Prozess im Zustand *bereit* sein.
- ☐ Ein Prozess kann direkt vom Zustand *blockiert* in den Zustand *laufend* versetzt werden.
- ☐ Ein Prozess kann direkt vom Zustand *laufend* in den Zustand *bereit* versetzt werden.

2) Mehrfachauswahlfragen (6 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) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig?

3 Punkte

- ○ Mit Hilfe des Systemaufrufs `execve(2)` (bzw. der Bibliotheksfunktion `exec(3)`) wird das bestehende Programm im aktuell laufenden Prozess ersetzt.
- ○ Der Prozess ist der statische Teil (Rechte, Speicher, etc.), das Programm der aktive Teil (Programmzähler, Register, Stack).
- ○ Der C-Präprozessor übersetzt mehrere C-Quelldateien in Module die zu einem Programm gebunden werden können.
- ○ Ein Prozess kann mit Hilfe von Threads mehrere Programme gleichzeitig ausführen.
- ○ Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- ○ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.

b) Welche der folgenden Aussagen zum Thema Threads sind richtig?

3 Punkte

- ○ Zur Umschaltung von federgewichtigen Prozessen ist ein Adressraumwechsel erforderlich.
- ○ Die Umschaltung von leichtgewichtigen Prozessen muss immer im Systemkern erfolgen.
- ○ Federgewichtige Prozesse blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ○ Leichtgewichtige Prozesse setzen den Einsatz von verdrängenden Scheduling-Verfahren voraus.
- ○ Kernel-Threads können Multiprozessoren nicht ausnutzen, da die Synchronisierung zu aufwändig ist.
- ○ Leichtgewichtige Prozesse können Multiprozessoren ausnutzen.

Aufgabe 2: funfisher (45 Punkte)

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

Sie wurden beauftragt das Programm *funfisher*, eine „selektive off-site Backup-Lösung“, zu implementieren. *funfisher* kopiert Dateien, die einem bestimmten Muster entsprechen, auf einen über ein Netzwerk verbundenen Server. Dabei werden, beginnend von einem Pfad (zu einer Datei oder einem Verzeichnis), rekursiv die Namen aller regulären Dateien mit einer Liste von Suchbegriffen verglichen. Falls ein Suchbegriff im Dateinamen enthalten ist, soll diese Datei durch Aufruf eines externen Programms verschickt werden. Die Nutzung sieht aus wie folgt:

```
agent@schlapphut: head -n 3 keywords
terror
schwarzgeld
indymedia
agent@schlapphut: ./funfisher /home/ ./keywords
agent@schlapphut: ./funfisher /var/tmp/secret.txt ./keywords
```

funfisher besteht aus der `main`-Funktion und drei weiteren Komponenten, die von Ihnen implementiert werden sollen:

- `void exfiltrate(const char *path)`
  kopiert die durch den Pfad *path* identifizierte Datei mit Hilfe eines zweiten Programms auf einen entfernten Server. Dazu wird das Programm *funftp* aufgerufen. Ein *funftp*-Aufruf hat folgenden Aufbau: **funftp host mov src dst**. *host* ist die Adresse des Servers (glob. Variable `HOST`). Der konstante String *mov* weist *funftp* an, eine Datei zu kopieren. *src* ist der Pfad der zu kopierenden Datei, während *dst* die auf dem Server zu erstellende Datei ist. *dst* entspricht dem mit einem Prefix (glob. Variable `ID`) versehenen Dateipfad. Nach jedem Aufruf von *funftp* wartet *funfisher* passiv auf dessen Terminierung. Beendet sich *funftp* mit einem Wert ungleich 0, soll auch *funfisher* sich mit einem Fehler beenden.
- `void iter(char *path, char **keywords, size_t n)`
  iteriert rekursiv durch das Dateisystem und prüft für jede reguläre Datei, ob ihr Name einem Suchbegriff entspricht. Falls dies der Fall ist, soll `exfiltrate` mit dem entsprechenden Dateipfad aufgerufen werden. Die Suchbegriffe sind in der Liste „`keywords`“ (siehe `read_list`) der Länge „`n`“ gespeichert. Zum Vergleichen eines Strings mit einem Suchbegriff kann die gestellte Funktion `char *strstr_ign_case(char *, char *)` verwendet werden, die wie die Ihnen bekannte Funktion `strstr(3)` funktioniert, außer dass Groß-/Kleinschreibung missachtet wird. Symbolische Verknüpfungen sollen wie reguläre Dateien behandelt werden.
- `size_t read_list(const char *path, char **buf[])`
  liest die angegebene Datei zeilenweise ein und speichert die Einträge in einem dynamischen Array. Die für den Zeiger des Array vorgesehene Speicherstelle wird als Argument `buf` übergeben. Jede Zeile ist max. 100 Zeichen lang (inkl. ’\n’) und soll ohne das Zeichen ’\n’ gespeichert werden. Der Rückgabewert ist die Anzahl der Einträgen.

Als weitere Besonderheit gilt, dass *funfisher* sich zu Beginn der Ausführung selbst löscht um mögliche Spuren zu verwischen. Die Funktion `unlink(const char *path)` wird mit dem Pfad der zu löschenden Datei aufgerufen - Fehler sollen dabei ignoriert werden.

Hinweise:

- Im Fehlerfall dürfen Sie die Ausführung (bspw. mithilfe von `die()`) abbrechen.
- Achten Sie darauf, im Erfolgsfall alle angeforderten Ressourcen wieder freizugeben.
- *funftp* ist unter einem in der PATH-Variable definierten Pfad verfügbar.

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

```c
#include <ctype.h>
#include <dirent.h>
#include <errno.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

static const char * const HOST = "77.87.229.75";
static const char * const ID = "3ff8dbf81799795483c9ac50c68a27c9";

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

```c
/* Die Funktion sucht das Vorkommen des Teilstring "needle" in "haystack"
 * Rückgabewert:
 * - NULL, wenn der Teilstring nicht gefunden wurde
 * - sonst, Zeiger auf die entsprechende Position innerhalb "haystack"
 */
static char *
strstr_ign_case(char *haystack, char *needle) {
  // ...
}
```



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

}

// Funktion zum Durchsuchen der Verzeichnisstruktur

  // Infos zu Datei/Verzeichnis sammeln
```

```
// Verzeichnisse durchsuchen
```

```
// Dateien prüfen
```

M:


```
// Funktion zum Erstellen der Liste
```

H:

```
// Funktion zum exfiltrieren von Dateien
```

E:

Aufgabe 3: Adressräume (11 Punkte)

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

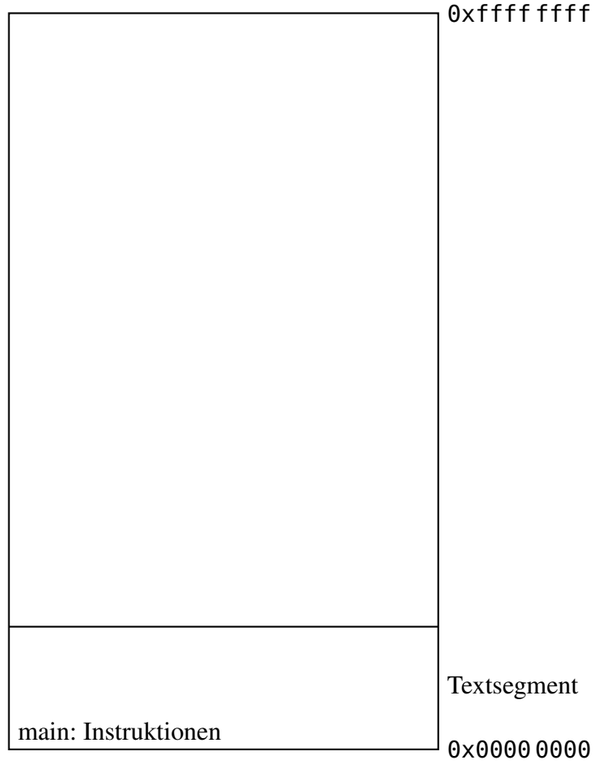
1) Tragen Sie die Segmente und deren Namen (analog zum schon vorgegebenen Textsegment) in unten stehende Zeichnung ein. Unterscheiden Sie hierbei die Bereiche zur Speicherung von initialisierten und nicht initialisierten Variablen. Zeichnen Sie für jede Variable ein, wo diese ungefähr im logischen Adressraum zu finden sein wird und welchen Wert sie enthält. 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. (11 Punkte)

```c
static int a;
static int *b = &a;
int c = 4;

static int foo(void);

int main(void) {
    static const char *s = "Hello World\n";
    int g = 5;
    int *h = &g;
    void *arr = malloc(5);
    int (*bar)(void) = foo;
}
```



Aufgabe 4: Unterbrechungen (14 Punkte)

Bei Ausnahmesituationen, wie sie bei der Ausführung eines Programms auftreten können, werden *Traps* und *Interrupts* unterschieden.

1) Beschreiben Sie diese beiden Arten:

   – wodurch entstehen diese Ausnahmesituationen?

   – wodurch unterscheiden sie sich?

   – geben Sie jeweils zwei Beispiele an

(8 Punkte)

2) Man unterscheidet bei der Ausnahmebehandlung Wiederaufnahmemodell und Terminierungsmodell. Was versteht man darunter und bei welchen Ausnahmen kann wie verfahren werden? (6 Punkte)