Aufgabe 1: Ankreuzfragen (30 Punkte)

1) Einfachauswahlfragen (22 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 der folgenden Aussagen zu statischem bzw. dynamischem Binden ist richtig? 2 Punkte

- ☐ Bei dynamischem Binden werden alle Referenzen spätestens bei der ersten Nutzung zu Adressen aufgelöst.
- ☐ Statisch gebundene Programme können zur Laufzeit durch das Nachladen neuer Programmodule (plug-ins) ergänzt werden.
- ☐ Beim statischen Binden werden alle Adressen zum Ladezeitpunkt aufgelöst.
- ☐ Bei dynamischem Binden müssen zum Übersetzungszeitpunkt alle Adressbezüge vollständig aufgelöst werden.

b) Welche der folgenden Aussagen zum Thema Synchronisation ist richtig? 2 Punkte

- ☐ Der Einsatz von nicht-blockierenden Synchronisationsmechanismen kann zu Verklemmungen (deadlocks) führen.
- ☐ Ein Anwendungsprozess muss bei der Verwendung von Semaphoren Interrupts sperren, um Probleme durch Nebenläufigkeit zu verhindern.
- ☐ Gibt ein Faden einen Mutex frei, den er selbst zuvor nicht angefordert hatte, stellt dies einen Programmierfehler dar; eine Fehlerbehandlung sollte eingeleitet werden.
- ☐ Zur Synchronisation eines kritischen Abschnitts ist passives Warten immer besser geeignet als aktives Warten.

c) Beim Blockieren in einem Monitor muss der Monitor freigegeben werden. Warum? 2 Punkte

- ☐ Weil sonst die Monitordaten inkonsistent sind.
- ☐ Weil kritische Abschnitte immer nur kurz belegt sein dürfen.
- ☐ Weil ein anderer Thread die Blockierungsbedingung nur aufheben kann, wenn er den Monitor vorher betreten kann.
- ☐ Weil der Thread sonst aktiv warten würde.

d) Ein Betriebssystem setzt logische Adressräume auf der Basis von Segmentierung ein. Welche Aussage ist richtig? 2 Punkte

- ☐ Einem Prozess kann mehr Speicher zugeordnet werden als physikalisch vorhanden ist, da ein Segment teilweise ausgelagert werden kann.
- ☐ Adressraumschutz durch Segmentierung erfordert keine Hardwareunterstützung.
- ☐ Die Segmentierung schränkt den logischen Adressraum derart ein, dass nur auf gültige Speicheradressen erfolgreich zugegriffen werden kann.
- ☐ Segmente können verschiedene Länge haben. Die Einhaltung der Längenbegrenzung wird vom C-Compiler überprüft.

e) Beim Einsatz von RAID-Systemen kann durch zusätzliche Festplatten ein fehlertolerierendes Verhalten erzielt werden. Welche Aussage dazu ist richtig? 2 Punkte

- ☐ Der Lesezugriff auf ein gestreiftes Plattensystem, insbesondere auch auf ein RAID 5, ist meist schneller, da mehrere Platten gleichzeitig beauftragt werden können.
- ☐ Bei RAID 4 und 5 darf eine bestimmte Menge von Festplatten nicht überschritten werden, da es sonst nicht mehr möglich ist, die Paritätsinformation zu bilden.
- ☐ Bei einem RAID 5 kommen immer genau 5 Platten zum Einsatz. Die Paritätsinformation wird dabei auf alle 5 Platten gleichmäßig verteilt.
- ☐ Wenn bei einem RAID 4 die Paritätsplatte ausfällt, sind alle Daten verloren.

f) Gegeben sei folgendes Szenario: zwei Fäden werden auf einem Monoprozessorsystem mit der Strategie "First Come First Served" verwaltet. In jedem Faden wird die Anweisung `i++;` auf der gemeinsamen, globalen `volatile` Variablen `i` ausgeführt. Welche der folgenden Aussagen ist richtig? 2 Punkte

- ☐ In einem Monoprozessorsystem ohne Verdrängung ist keinerlei Synchronisation erforderlich.
- ☐ Während der Inkrementoperation müssen Interrupts vorübergehend unterbunden werden.
- ☐ Die Inkrementoperation muss mit einer CAS-Anweisung synchronisiert werden.
- ☐ Die Operation `i++` ist auf einem Monoprozessorsystem immer atomar.

g) Gegeben seien die folgenden Präprozessor-Makros: 2 Punkte

```
#define MUL(x, y) x * y
#define SUB(x, y) x - y
```

Was ist das Ergebnis der Präprozessorauswertung des folgenden Ausdrucks?

```
SUB(3, (MUL(4, 2))) * 2
```

- ☐ `3 - 4 * 2 * 2`
- ☐ `3 - (4 * 2) * 2`
- ☐ `(3 - (4 * 2) * 2)`
- ☐ `((3 - (4 * 2)) * 2)`

h) In einem UNIX-Dateisystem gibt es symbolische Verweise (Symbolic Links) und feste Verweise (Hard Links) auf Dateien. Welche Aussage ist richtig.

2 Punkte

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

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

2 Punkte

- ☐ Mit Hilfe des Systemaufrufs `exec(3)` wird das bestehende Programm im aktuell laufenden Prozess ersetzt.
- ☐ Der Binder erzeugt aus einer oder mehreren Objekt-Dateien einen Prozess.
- ☐ Der UNIX-Systemaufruf `fork(2)` lädt eine Programmdatei in einen neu erzeugten Prozess.
- ☐ Mit Hilfe von Threads kann ein Prozess mehrere Programme gleichzeitig ausführen.

j) Man unterscheidet zwischen privilegierten und nicht-privilegierten Maschinenbefehlen. Welche Aussage ist richtig?

2 Punkte

- ☐ Privilegierte Maschinenbefehle dürfen in Anwendungsprogrammen grundsätzlich nicht verwendet werden.
- ☐ Privilegierte Maschinenbefehle können durch Betriebssystemprogramme implementiert werden.
- ☐ Mit nicht-privilegierten Befehlen ist der Zugriff auf Geräteregister grundsätzlich nicht möglich.
- ☐ Die Benutzung eines privilegierten Maschinenbefehls in einem Anwendungsprogramm führt zu einer asynchronen Programmunterbrechung.

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

2 Punkte

- ☐ 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.
- ☐ Der Speicher schickt an die CPU einen Interrupt. Hierdurch wird das Betriebssystem angesprungen, das dem gerade laufenden Prozess das Signal SIGSEGV (Segmentation Violation) zustellt.
- ☐ Beim Zugriff muss die MMU, soweit vorhanden, die erforderliche Adressumsetzung vornehmen, erkennt die ungültige Adresse und löst einen Trap aus.

2) Mehrfachauswahlfragen (8 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) Der Speicher eines UNIX-Prozesses ist in Text-, Daten- und Stack-Segment untergliedert. Welche Aussagen bezogen auf C-Programme sind richtig?

4 Punkte

- ○ Die Sichtbarkeit globaler static-Variablen ist auf das jeweilige Modul beschränkt.
- ○ Funktionspointer werden nicht immer im Text-Segment gespeichert.
- ○ Globale schreibbare Variablen liegen im Daten-Segment.
- ○ Dynamisch allozierte Zeichenketten liegen im Text-Segment.
- ○ Bei einem Aufruf von `malloc(3)` wird das Stack-Segment dynamisch erweitert.
- ○ Der Code von Funktionen wird zusammen mit den Variablen der Funktion im Stack-Segment abgelegt.
- ○ Variablen der Speicherklasse static liegen im Daten-Segment.
- ○ Lokale static-Variablen werden bei jedem Betreten der zugehörigen Funktion neu initialisiert.

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

4 Punkte

- ○ Der Zugriff auf eine physikalische Speicheradresse kann zu einem Trap führen.
- ○ Ein durch einen Interrupt unterbrochenes Programm darf je nach der Interruptursache entweder abgebrochen oder fortgesetzt werden.
- ○ Die CPU sichert bei einem Interrupt einen Teil des Prozessorzustands.
- ○ Da Traps immer synchron auftreten, kann es im Rahmen ihrer Behandlung nicht zu Wettlaufsituationen mit dem unterbrochenen einfädigen Programm kommen.
- ○ Ein Systemaufruf im Anwendungsprogramm ist der Kategorie Interrupt zuzuordnen.
- ○ Ein Trap steht nicht zwangsläufig in ursächlichem Zusammenhang mit dem unterbrochenen Programm.
- ○ Die Ausführung einer Ganzzahl-Rechenoperation (z. B. Addition, Division) kann zu einem Trap führen.
- ○ Ein Trap signalisiert einen schwerwiegenden Fehler und führt deshalb immer zur Beendigung des unterbrochenen Programms.

Aufgabe 2: intbox (60 Punkte)

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

Schreiben Sie ein Programm `intbox`, das auf TCP/IPv6-Port 71 einen Dienst anbietet, der den Inhalt von Verzeichnissen liefert. Zur Abarbeitung parallel eintreffender Anfragen wird ein Threadpool verwendet; die Threads werden über einen Ringpuffer mit Verbindungen versorgt.

Ein Client sendet pro Verbindung **einen** Pfad inklusive Newline mit einer Maximallänge von 1337 Zeichen an den Server (gehen Sie davon aus, dass längere Zeilen nicht auftreten). Der Server liefert daraufhin den Verzeichnisinhalt im Textformat an den Client zurück und schließt die Verbindung. Um Anfragen zu beschleunigen werden die fertig formatierten Verzeichnisinhalte in einem Cache gespeichert, der als einfach verkettete Liste implementiert ist. Parallele Zugriffe werden **nichtblockierend** synchronisiert; doppelte Cacheeinträge durch Wettlaufsituationen dürfen ignoriert werden. Die Einträge des Caches bleiben dauerhaft bestehen und werden **nicht** wieder freigegeben.

Der Server soll sich **nicht** ohne Fehlerausgabe beenden, falls Clients die Verbindung trennen, bevor der Server antworten konnte.

Sobald das `SIGINT`-Signal (`Ctrl-C`) auftritt, soll der Server **sofort** die Annahme neuer Verbindungen einstellen, selbst wenn das Programm gerade in `accept()` auf Verbindungen wartet. Alle bereits angenommenen Verbindungen werden fertig behandelt. Zur Deinitialisierung legt der Hauptthread für jeden Arbeiterthread einen speziellen Wert in den Ringpuffer (`POISON`). Bei der Entnahme dieses Wertes beendet sich der jeweilige Arbeiterthread. Der Hauptthread wartet **passiv** bis sich alle Arbeiterthreads beendet haben und beendet sich anschließend.

Das Programm soll folgendermaßen strukturiert sein:

- Das Hauptprogramm initialisiert alle benötigten Datenstrukturen, konfiguriert die Signalbehandlungen, legt einen Ringpuffer mit **42** Elementen an (`bb_create()`), startet **23** Arbeiterthreads und nimmt Socket-Verbindungen entgegen. Alle Anfragen werden per `bb_put()` in den Ringpuffer eingefügt.
- **`void *worker_thread(void *)`** dient als Startroutine der Arbeiterthreads und entnimmt Verbindungen aus dem Ringpuffer (`bb_get()`) und ruft in einer Endlosschleife `handle_request()` auf. Um sicherzustellen, dass das `SIGINT`-Signal nur dem Hauptthread zugestellt wird, blockieren alle Arbeiterthreads `SIGINT` mit `pthread_sigmask()` (verhält sich analog zu `sigprocmask()`).
- **`void handle_request(FILE *read, FILE *write)`** liest für jede Verbindung den Pfad aus der Anfrage aus. Beachten Sie, dass das Newline-Zeichen (`'\n'`) nicht Teil des Pfades ist. Falls der angefragte Pfad im Cache vorhanden ist (`cache_get()`) wird die Anfrage direkt aus dem Cache beantwortet. Existiert noch kein passender Eintrag, dann wird der Verzeichnisinhalt mittels der **vorgegebenen** Funktion `format_directory()` formatiert, in den Cache geschrieben (`cache_put()`) und anschließend an den Client gesendet.
- **`const char *cache_get(const char *path)`** sucht den Verzeichnisinhalt für `path` im Cache und gibt diesen zurück oder `NULL`, falls nicht vorhanden.
- **`void cache_put(const char *path, const char *data)`** fügt den Verzeichnisinhalt für `path` dem Cache hinzu. Dazu wird der Kopf der Liste mittels Compare-and-Swap (CAS, mit dem **vorgegebenen** Makro) mit dem neuen Element getauscht. Das für die Liste zu verwendende `struct` ist auf der folgenden Seite gegeben.

Achten Sie auf korrekte Speicherverwaltung. Nutzen Sie für den Ringpuffer die bekannten Schnittstellen, deren Signaturen auf der folgenden Seite dargestellt sind; der Ringpuffer ist **nicht** selbst zu implementieren. Zur Vereinfachung dürfen alle Fehler nach der üblichen Fehlerausgabe zu einer **Beendigung** des Programms führen. Das Formatieren des Verzeichnisinhalts müssen Sie **nicht** selbst implementieren; nutzen Sie dafür die vorgegebene Funktion `format_directory()`.

Hinweis: Falls `accept()` auf eine Verbindung wartet und der dazugehörige Serversocket währenddessen geschlossen wird, so kehrt `accept()` sofort zurück und setzt die `errno` auf `EBADF`.

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

```c
#include <dirent.h>
#include <errno.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdatomic.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define POISON (-1)

// Atomic compare-and-swap operation, in pseudo-code:
//     if (*object_ptr == *expected_ptr) {
//         *object_ptr = desired;
//     }
// Returns true if CAS was successful, false otherwise.
#define CAS(object_ptr, expected_ptr, desired) \
    atomic_compare_exchange_strong(object_ptr, expected_ptr, desired)

// API: bounded buffer
struct bbuf; // forward declaration
// Create new bounded buffer; returns NULL on error and sets errno.
struct bbuf *bb_create(size_t size);
// Add entry to bounded buffer; blocks if full, cannot fail.
void bb_put(struct bbuf *bb, int value);
// Retrieve element from bounded buffer, waiting until one is available;
// cannot fail.
int bb_get(struct bbuf *bb);

// Return the formatted directory content of the path as allocated string;
// cannot fail.
const char *format_directory(const char *path);

// Struct for a single cache entry
struct cache {
    struct cache *next;
    const char *path;
    const char *data;
};

// Helper function
static void die(const char *s) {
    perror(s);
    exit(EXIT_FAILURE);
}
```

```
// Globale Variablen; Vorwärtsdeklarationen von Funktionen
// dürfen weggelassen werden

// Funktion zur Signalbehandlung
```



```
// Hauptfunktion (main)
int main(void) {
    // Datenstrukturen initialisieren

    // Signalbehandlungen initialisieren

    // Threads starten
```

```
    // Socket erstellen
```

```
    // Verbindungen annehmen
```



```
    // Threads signalisieren und auf Beendigung dieser warten
}
```

M:

```
// Funktion für Arbeiterthread
```

W:


```
// Funktion für Requests
```

R:

```
// Funktion zum Lesen aus dem Cache
```

```
// Funktion zum Schreiben in den Cache
```

C:

Aufgabe 3: Adressräume & Freispeicherverwaltung (15 Punkte)

1) Ein in der Praxis häufig eingesetztes Verfahren zur Verwaltung von freiem Speicher ist das *Buddy*-Verfahren.

Nehmen Sie einen Speicher von 1024 Bytes an und gehen Sie davon aus, dass die Freispeicher-Verwaltungsstrukturen separat liegen. Initial ist bereits ein Datenblock der Größe 128 Bytes vergeben worden. Ein Programm führt nacheinander die im folgenden Bild angegebenen Anweisungen aus. (11 Punkte)

⓪ p0 = malloc(100); // 0 (initial vergebener Block)

① p1 = malloc(50);

② p2 = malloc(110);

③ p3 = malloc(32);

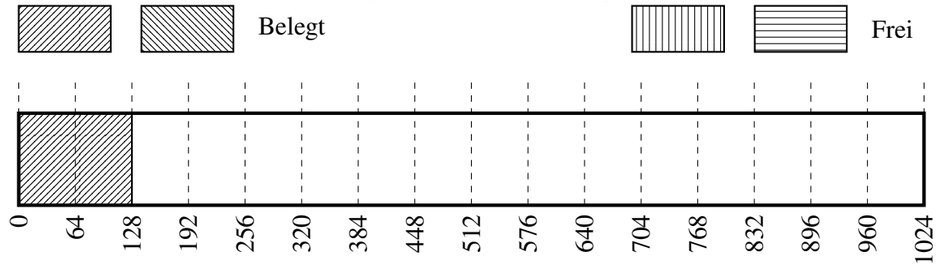
④ free(p1);

⑤ p4 = malloc(386);

⑥ free(p3);

⑦ free(p4);

Tragen Sie hinter den obigen Anweisungen jeweils ein, welches Ergebnis die `malloc()` - Aufrufe zurückliefern. Skizzieren Sie in der folgenden Grafik, wie der Speicher nach **Schritt ⑤** aussieht, und tragen Sie in der Tabelle den aktuellen Zustand der Lochliste nach **jedem** Schritt ein. Für Löcher gleicher Größe schreiben Sie die Adressen einfach nebeneinander in die Tabellenzeile (es ist nicht notwendig, verkettete Buddys wie in der Vorlesung beschrieben einzutragen).



Größe der Buckets	initial ⓪	①	②	③	④	⑤	⑥	⑦
2^5								
2^6								
2^7	128							
2^8	256							
2^9	512							
2^{10}								

2) Man unterscheidet bei Adressraumkonzepten und bei Zuteilungsverfahren zwischen externer und interner Fragmentierung. Erläutern Sie den Unterschied. Was kann man jeweils gegen die beiden Arten der Fragmentierung tun? (4 Punkte)

Aufgabe 4: Dateisystem (15 Punkte)

1) Beschreiben Sie kurz (in Stichworten) die grundsätzliche Funktionsweise eines *Journaling-File-Systems.* (3 Punkte)

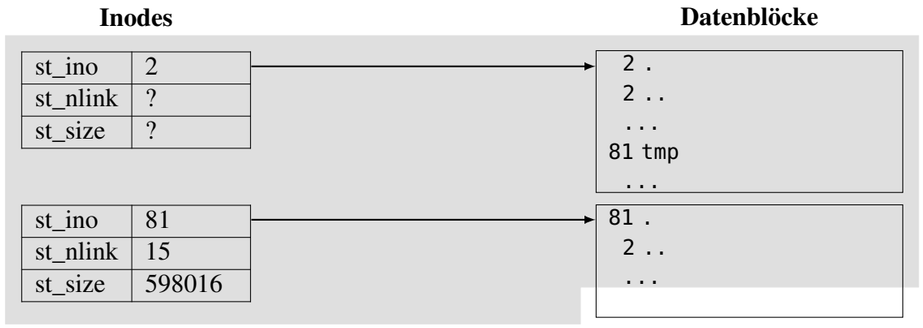
2) Gegeben ist die folgende Ausgabe des Kommandos `ls -aoRi /tmp/sp` auf einem Linux-System; eine rekursiv absteigende Ausgabe aller Dateien und Verzeichnisse unter `/tmp/sp` mit Angabe der Inode-Nummer (erste Spalte), des Referenzzählers (dritte Spalte) und der Dateigröße (fünfte Spalte). (12 Punkte)

```
simon@faui0sr0:~$ ls -aoRi /tmp/sp
/tmp/sp:
total 596
39 drwxr-xr-x  3 simon   4096 Jul 26 12:45 ./
81 drwxrwxrwt 15 root  598016 Jul 26 12:34 ../
45 drwxr-xr-x  3 simon   4096 Jul 26 12:43 folder/
67 lrwxrwxrwx  1 simon     12 Jul 26 12:35 important -> folder/file3

/tmp/sp/folder:
total 16
45 drwxr-xr-x 3 simon 4096 Jul 26 12:43 ./
39 drwxr-xr-x 3 simon 4096 Jul 26 12:45 ../
75 drwxr-xr-x 2 simon 4096 Jul 26 12:43 deeper/
49 -rw-r--r-- 2 simon   42 Jul 26 12:38 file1
73 lrwxrwxrwx 1 simon   14 Jul 26 12:43 file2 -> deeper/thefile

/tmp/sp/folder/deeper:
total 16
75 drwxr-xr-x 2 simon 4096 Jul 26 12:43 ./
45 drwxr-xr-x 3 simon 4096 Jul 26 12:43 ../
49 -rw-r--r-- 2 simon   42 Jul 26 12:38 fileX
53 -rw-r--r-- 1 simon  666 Jul 26 12:43 thefile
```

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.



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	