

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 Aussage zum Thema Systemaufrufe ist richtig?

2 Punkte

- ☐ Durch die Bereitstellung von Systemaufrufen, kann ein Benutzerprogramm das Betriebssystem um eigene Funktionen erweitern.
- ☐ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht selbst ausführen dürfte.
- ☐ Die Bearbeitung eines Systemaufrufs findet immer im selben Adressraum statt, aus dem heraus der Systemaufruf abgesetzt wurde.
- ☐ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.

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

2 Punkte

- ☐ Nach dem Beendigungsmodell werden Interrupts bearbeitet. Gibt man z. B. CTRL-C unter UNIX über die Tastatur ein, wird ein Interrupt-Signal an den gerade laufenden Prozess gesendet und dieser dadurch beendet.
- ☐ Das Beendigungsmodell sieht das Herunterfahren des Betriebssystems im Falle eines schwerwiegenden Fehlers vor.
- ☐ Das Wiederaufnahmmodell ist für Interrupts und Traps gleichermaßen geeignet.
- ☐ Interrupts sollten nach dem Beendigungsmodell behandelt werden, weil ein Zusammenhang zwischen dem unterbrochenen Prozess und dem Grund des Interrupts bestehen kann.

c) Welche Aussage über die Koordinierung von kritischen Abschnitten unter Unix ist richtig?

2 Punkte

- ☐ Für die Synchronisation zwischen dem Hauptprogramm und einer Signalbehandlungsfunktion sind Schlossvariablen (Locks) ungeeignet.
- ☐ Ein Unix-Prozess kann durch das Sperren von Unterbrechungen (Interrupts) den Speicherzugriff in einem kritische Abschnitte synchronisieren.
- ☐ In einem Unix-Prozess kann es keinen kritischen Abschnitt geben, da immer nur ein Aktivitätsträger pro Prozess aktiv ist.
- ☐ Kritische Abschnitte können unter Unix nur mit Semaphoren synchronisiert werden.

d) Welche Aussage zum Thema Programme und Prozesse ist richtig?

2 Punkte

- ☐ Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- ☐ In einem Prozess kann immer nur ein Programm ausgeführt werden.
- ☐ Ein Prozess kann gleichzeitig mehrere verschiedene Programme ausführen.
- ☐ Der Compiler erzeugt aus mehreren Programmteilen (Module) einen Prozess.

e) Was ist ein Stack-Frame?

2 Punkte

- ☐ Ein Fehler, der bei unberechtigten Zugriffen auf den Stack-Speicher entsteht.
- ☐ Der Speicherbereich, in dem der Programmcode einer Funktion abgelegt ist.
- ☐ Ein Bereich des Speichers, in dem u.a. lokale automatic-Variablen einer Funktion abgelegt sind.
- ☐ Ein spezieller Registersatz des Prozessors zur Bearbeitung von Funktionen.

f) Ein laufender Prozess wird in den Zustand *bereit* überführt. Welche Aussage passt zu diesem Vorgang?

2 Punkte

- ☐ Es ist kein direkter Übergang von laufend nach *bereit* möglich.
- ☐ Der Prozess wird durch einen anderen Prozess verdrängt, oder gibt die CPU freiwillig ab.
- ☐ Der Prozess wartet auf Daten von der Festplatte.
- ☐ Der Prozess wartet mit dem Systemaufruf `waitpid(3)` auf die Beendigung eines anderen Prozesses.

g) Welche Aussage zur Verklemmungsverhinderung ist richtig?

2 Punkte

- ☐ Bei Verklemmungsverhinderung wird dafür gesorgt, dass eine der vier notwendigen Bedingungen für Verklemmungen nicht auftreten kann.
- ☐ Bei einem Zyklus im erweiterten Betriebsmittelgraph liegt ein unsicherer Zustand vor.
- ☐ Mit Hilfe des Bankiers-Algorithmus wird beim Belegen eines Betriebsmittels geprüft, ob mit erfolgreicher Belegung ein unsicherer Zustand eintreten würde.
- ☐ Das System überprüft vor dem Freigeben von Betriebsmitteln, ob ein unsicherer Zustand eintreten würde.

h) Welche Seitennummer und welcher Offset gehören bei einstufiger Seitennummierung und einer Seitengröße von 1024 ($= 2^{10}$) Bytes zu folgender logischer Adresse: 0x0802?

2 Punkte

- ☐ Seitennummer 0x2, Offset 0x2
- ☐ Seitennummer 0x8, Offset 0x8
- ☐ Seitennummer 0x2, Offset 0x8
- ☐ Seitennummer 0x8, Offset 0x2

i) Welche Aussage zu UNIX/Linux-Dateideskriptoren ist korrekt?

2 Punkte

- ☐ Nach dem Aufruf von `fork(2)` teilen sich Eltern und Kindprozess die den gemeinsamen Dateideskriptoren zu Grunde liegenden Kernel-Datenstrukturen.
- ☐ Da Dateideskriptoren Zeiger auf Betriebssystem-interne Strukturen sind, können diese zwischen Prozessen geteilt werden.
- ☐ Der Dateideskriptor enthält die nötigen Metadaten einer Datei und ist auf der Festplatte gespeichert.
- ☐ Das Flag `FD_CLOFORK` eines Dateideskriptors sorgt dafür, dass der Dateideskriptor bei einem Aufruf von `fork(2)` automatisch geschlossen wird.

j) Beim Einsatz von RAID-Systemen wird durch zusätzliche Festplatten ein fehler-tolerierendes Verhalten erzielt. Welche Aussage dazu ist richtig?

2 Punkte

- ☐ Bei RAID 4 Systemen wird die Paritätsinformation gleichmäßig über alle beteiligten Platten verteilt.
- ☐ Bei RAID 5 Systemen sind mindestens 5 Festplatten nötig.
- ☐ 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.
- ☐ Der Lesezugriff auf ein gestreiftes Plattensystem, insbesondere auch auf ein RAID 5 System, ist schneller, da mehrere Platten gleichzeitig beauftragt werden können.

k) Welche der folgenden Aussagen zum Thema persistente Datenspeicherung ist richtig?

2 Punkte

- ☐ Bei indizierter Speicherung von Dateien müssen unter Umständen mehrere Blöcke geladen werden, bevor der Dateiinhalt gelesen werden kann.
- ☐ Bei kontinuierlicher Speicherung ist es immer problemlos möglich, bestehende Dateien zu vergrößern.
- ☐ Beim Einsatz von RAID 0 kann eine der beteiligten Platten ausfallen, ohne dass das Gesamtsystem ausfällt.
- ☐ Bei verketteter Speicherung dauert der wahlfreie Zugriff auf eine bestimmte Dateiposition immer gleich lang, wenn Cachingeffekte außer Acht gelassen werden.

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) 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.
- ☐ In einem Verzeichnis darf es keinen Eintrag geben, der auf das Verzeichnis selbst verweist, da andernfalls eine Endlosschleife bei dem Auslesen auftreten könnte.
- ☐ Für das Löschen einer Datei sind die Rechte-Informationen im Dateikopf (*Inode*) der Datei irrelevant.
- ☐ Der selbe *Inode* kann im Dateisystem mehrfach über verschiedene Pfade referenziert werden.
- ☐ Beim lesenden Zugriff auf eine Datei über einen *symbolic link* kann ein Prozess den Fehler *Permission denied* erhalten, obwohl er das Leserecht auf dem *symbolic link* besitzt.
- ☐ Innerhalb eines Verzeichnisses können mehrere Verweise auf den selben *Inode* existieren, sofern diese unterschiedliche Namen haben.
- ☐ Auf eine Datei in einem Dateisystem verweisen immer mindestens zwei *hard-links*.
- ☐ Wird eine reguläre Datei gelöscht, so kann auf deren Inhalt über *symbolic links*, die auf diese Datei verweisen, noch zugegriffen werden, da diese nicht gelöscht werden.

b) Welche der Aussagen zu folgenden Programmfragment sind richtig?

4 Punkte

```
static int a = 2022;
void f1 (const int *y) {
    static int b;
    int c;
    char *d = malloc(0x802);
    void (*e)(const int *) = f1;
    y++;
    // ...
}
```

- ☐ e liegt im Stacksegment und zeigt in das Textsegment.
- ☐ c ist mit dem Wert 0 initialisiert.
- ☐ Die Anweisung y++ führt zu einem Laufzeitfehler, da y konstant ist.
- ☐ d ist ein Zeiger, der in den Heap zeigt.
- ☐ a liegt im Datensegment.
- ☐ y liegt im Stacksegment.
- ☐ b liegt im Stacksegment.
- ☐ Die Speicherstelle, auf die d zeigt, verliert beim Rücksprung aus der Funktion f1() ihre Gültigkeit.

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

Aufgabe 2: SemServer (60 Punkte)

Die Implementierung von Semaphoren basiert darauf, dass die zu synchronisierenden Prozesse auf gemeinsamen Speicher zugreifen können. Sollen aber Prozesse auf verschiedenen Rechnern im Netz synchronisiert werden, ist eine mögliche Lösung, einen zentralen Semaphor-Server bereit zu stellen. Solch ein Serverprogramm soll hier implementiert werden. Es nimmt auf TCP-Port 7076 Verbindungen entgegen, über die zeilenweise Kommandos zum Erzeugen von Semaphoren sowie für P- und V-Operationen gesendet werden können. Das erste Zeichen jeder Zeile gibt dabei die gewünschte Operation an, direkt gefolgt von einem, von der Operation abhängigen Argument. Neben den Operationen **P** und **V** gibt es noch die Initialisierung eines Semaphors. Semaphore werden über ihre Speicheradressen identifiziert. Um die Gültigkeit eines Semaphors prüfen zu können, werden diese in einer globalen Instanz der gegebenen Listenimplementierung gespeichert.

Implementieren Sie folgende Funktionen:

int main(int argc, char *argv[]) Initialisiert den globalen Zustand des Prozesses, nimmt Verbindungen an und erzeugt zu jeder eine **struct client** (**clientCreate()**). Die weitere Bearbeitung erfolgt in einem eigenen Thread, der über die Funktion **workerStart()** gestartet wird. Dieser werden die Bearbeitungsfunktion **clientProcess()** sowie die **client**-Struktur übergeben. Fehler werden auf **stderr** ausgegeben, falls möglich wird die Ausführung des Prozesses aber fortgesetzt.

int workerStart(void (*fn)(void*), void *arg) Startet **fn** mit **arg** als Argument in einem neuen Faden, dessen Ressourcen automatisch bei dessen Beendigung freigegeben werden. Im Fehlerfall wird **errno** auf einen passend Wert gesetzt und -1 statt 0 zurückgegeben.

struct client *clientCreate(int fd) Erzeugt eine **struct client** für die Verbindung **fd**. Im Fehlerfall wird **errno** gesetzt, NULL zurückgegeben und belegte Ressourcen freigegeben. In jedem Fall darf **fd** vom Aufrufer nicht weiter verwendet werden.

void *clientProcess(void *arg) Liest Zeilen mit Arbeitsaufträgen von der in **arg** als **struct client** übergebenen Verbindung ein und startet Fäden zu deren Bearbeitung. Zeilen mit mehr als 20 (**MAX_LINE_LEN**) Zeichen werden ohne Meldung verworfen. Für alle anderen wird eine **struct request** erzeugt und in einem neuen Faden durch die Funktion **handleRequest()** bearbeitet. Treten dabei Fehler auf, wird der Fehlergrund (**errno**) mit der Funktion **reply()** an den Clienten geschickt (**strerror(3)**). Wurden alle Zeilen eingelesen, muss mit dem Aufräumen der **struct client** gewartet werden, bis alle Anfragen bearbeitet wurden, wozu der Semaphor **requests** in der Struktur vorgesehen ist.

void clientDestroy(struct client *client) Gibt die Ressourcen von **client** frei.

void *handleRequest(void *arg) Bearbeitet das als **arg** übergebene **struct request**-Objekt. Abhängig vom ersten Zeichen in **line** wird **handleI()** oder **handlePV()** ausgeführt. Geben diese einen Wert ungleich 0 zurück, wird der Fehlergrund (**errno**) als Antwort verwendet. Andernfalls haben die Funktion selbst bereits eine Antwort geschickt. Auf ungültige Operationen wird mit "unknown_operation" geantwortet (**reply()**).

int handleI(const struct request *rq) Initialisiert einen neuen Semaphor mit dem in der Anfragezeile angegebenen Initialwert (**parseNumber()**). Im Erfolgsfall wird die Speicheradresse des Semaphors in die Liste der gültigen Semaphore eingetragen (**listAdd()**) und mit **reply()** als Antwort geschickt.

int handlePV(const struct request *rq, void (*fn)(SEM *)) Prüft, ob die in der Anfragezeile gegebene Adresse (**parsePointer()**) gültig ist (**listContains()**) und führt die Funktion **fn** darauf aus. Für ungültige Adressen wird **errno=ENOENT** gesetzt und -1 zurückgegeben, andernfalls "success" als Antwort (**reply()**) gesendet.

Hinweise:

- Die Funktion **strerror(3)** gibt eine passende Zeichenkette für einen Fehlercode zurück.
- Die vorgegebenen Listenimplementierung ist *thread-safe*.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <string.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#define MAX_LINE_LEN 20

struct client {
    FILE *rx, *tx;
    SEM *requests;
};
struct request {
    struct client *client;
    char line[];
};

// vorgegebene Funktionen:
void die(const char *name) {
    perror(name);
    exit(EXIT_FAILURE);
}

// parse the string @str as number or pointer into *@dst.
// returns 0 on success or in case of an error -1 and sets errno.
int parseNumber(const char *str, int *dst);
int parsePointer(const char *str, void **dst);

// send @fmt to @f as reply to request @line
void reply(FILE *f, const char *line, const char *fmt, ...);

struct list;
// create and return a new list or NULL; sets errno in case of an error
struct list *listCreate(void);
// add @sem to @list or return -1; sets errno in case of an error
int listAppend(struct list *list, SEM *sem);
// returns 1 if @list contains @sem, 0 otherwise
int listContains(const struct list *list, SEM *sem);

typedef struct SEM SEM;
// create and return a new semaphore or NULL and set errno in case of an error
SEM *semCreate(int initVal);
// destroy a semaphore and free all associated resources.
void semDestroy(SEM *sem);
// P- and V-operations never fail.
void P(SEM *sem);
void V(SEM *sem);
```

// Globale Variablen

int main(int argc, char *argv[]) {

G:

5

5

7

M:

7

W:

```
struct client *clientCreate(int fd) {
```

10

}

```
void *clientProcess(void *arg) {
```

```
// Zeilenlänge prüfen
```

5

```
// Anfrage erstellen und zur Bearbeitung auslagern
```

7

}

```
void clientDestroy(struct client *client) {
```

}

```
void * handleRequest(void *arg) {
```

}

10

10

C: $\therefore$

10

11


```
int handleI(const struct request *rq) {
```

5

7

1

7

}

```
int handlePV(const struct request *rq, void (*fn)(SEM *)) {
```

7

9

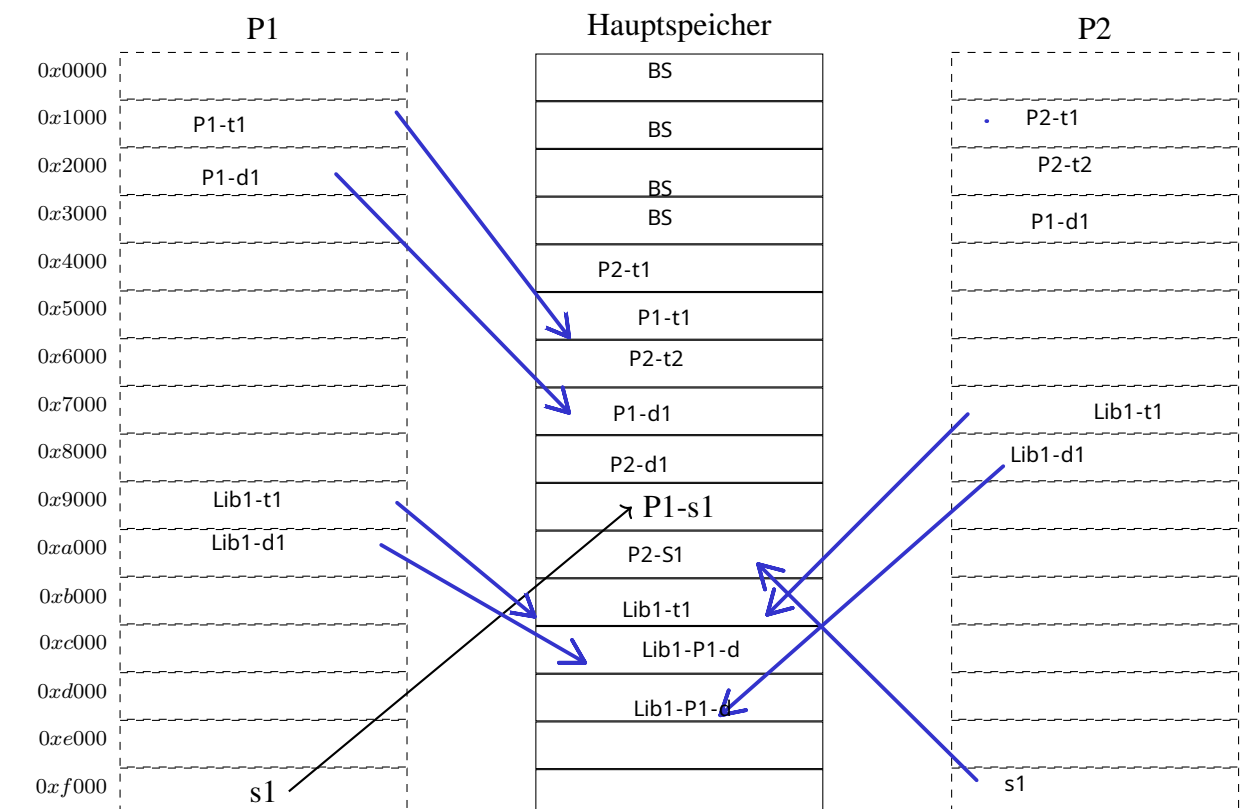
7

H:

Aufgabe 3: Adressräume (17 Punkte)

Für die nachfolgende Aufgabe wird ein System mit Seitenadressierung angenommen. Eine Adresse besteht aus 16 Bit und die 64 Kibibyte Hauptspeicher werden über Seiten der Größe 4 Kibibyte verwaltet. (Zur Erinnerung: $4096_{10} = 1000_{16}$)

Das (sehr kleine) Betriebssystem (*BS*) belegt dabei die ersten 4 Seitenrahmen und stellt sicher, dass die erste Seite des virtuellen Adressraums jedes Prozesses nicht genutzt wird. Werden neue Seiten im Hauptspeicher belegt, wird immer die nächste freie Seite mit der niedrigsten Adresse verwendet. Um Speicherplatz zu sparen wird versucht, dass inhaltsgleiche Seiten, die nicht geändert werden können, nur einmal im Hauptspeicher liegen.



1) Initial wird ein Prozess (*PI*) ausgeführt, mit je 4096 Byte für das Text-, Daten- und Stacksegment. Die folgenden Seiten wurden dem Prozess zugeteilt: *PI-tl* = 0x5000, *PI-dl* = 0x7000, *PI-sl* = 0x9000. In obiger Skizze ist bereits die Zuordnung für das Stacksegment von Prozess *PI* (*PI-sl*) eingetragen. Targen Sie alle weiteren Belegungen und Zuordnung, die bis hier her aus der Angabe bekannt sind in diese Skizze ein. (5 Punkte)

2) *P1* führt die Systemaufrufe `fork(2)` und `exec(2)` aus um ein anderes Programm in *P2* auszuführen. Das gestartete Programm hat ein Textsegment der Größe 8000 Kilobyte sowie ein Datensegment mit 100 Byte. Erweitern Sie die Skizze aus Teilaufgabe 1 um den virtuellen Adressraum für *P2*. (6 Punkte)

3) Beide Prozesse laden die dynamische Bibliothek *Lib1* nach. *Lib1* hat je eine Seite im Text- sowie im Datensegment. In *P1* wird *Lib1* ab Adresse 0x9000 in den virtuellen Adressraum geladen. Für *P2* soll *Lib1* hingegen ab Adresse 0x7000 zugreifbar sein. Vervollständigen Sie obiges Bild entsprechend und markieren Sie die Änderungen mit einer (3). (6 Punkte)

Aufgabe 4: Einplanung von Prozessen (13 Punkte)

1) Bei der Einplanung von Prozessen werden Gütemerkmale bzw. Kriterien zur Aufstellung der konkreten Einlastungsreihenfolge von Prozessen unterschieden.

Die Kriterien lassen sich übergreifend in zwei Kategorien einteilen. Beschreiben Sie im folgenden kurz den jeweiligen Fokus der Kategorie. Nenne Sie darüber hinaus je einen Vertreter der entsprechenden Kategorie und beschreiben Sie, welches Verhalten durch dieses Merkmal erreicht werden soll. (6 Punkte)

a) Kategorie der **benutzerorientierten** Kriterien (3 Punkte)

b) Kategorie der **systemorientierten** Kriterien (3 Punkte)

2) Welche Kriterien sind beim **Echtzeitbetrieb** eines System besonders zu bevorzugen? Zu welchen Konflikten zwischen Kriterien kann es dabei kommen? (4 Punkte)

3) Einplanungsverfahren lassen sich ebenfalls verschiedenen Kategorien zuschreiben. Beschreiben Sie kurz die Eigenschaften von **kooperativen** Einplanungsstrategien. Geben Sie darüber hinaus ein Verfahren aus dieser Kategorie an. In welcher Anwendungssituation kann dieses sinnvoll eingesetzt werden? (3 Punkte)
