Aufgabe 1.1: Einfachauswahl-Fragen (22 Punkte)

Bei den Multiple-Choice-Fragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Multiple-Choice-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 zu Semaphoren ist richtig? 2 Punkte

- ❏ Die *P*-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- ❏ Die *V*-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- ❏ Ein Semaphor kann nur zur Signalisierung von Ereignissen, nicht jedoch zum Erreichen gegenseitigen Ausschlusses verwendet werden.
- ❏ Die *V*-Operation eines Semaphors kann ausschließlich von einem Thread aufgerufen werden, der zuvor mindestens eine *P*-Operation auf dem selben Semaphor aufgerufen hat.

b) Ein laufender Prozess wird in den Zustand *gestoppt* überführt. Welche Aussage passt zu diesem Vorgang? 2 Punkte

- ❏ Es ist kein direkter Übergang von *laufend* nach *gestoppt* möglich.
- ❏ Der Prozess terminiert und gibt seine Betriebsmittel frei.
- ❏ Der Prozess wurde im Rahmen der mittelfristigen Einplanung aus dem Arbeitsspeicher verdrängt und auf den Hintergrundspeicher ausgelagert.
- ❏ Der Prozess wurde angehalten, kann aber in der Zukunft fortgesetzt werden.

c) Welche Aussage über Variablen in C-Programmen ist richtig? 2 Punkte

- ❏ Lokale *automatic*-Variablen, die auf dem Stack angelegt werden, werden immer mit dem Wert 0 initialisiert.
- ❏ Wird dem Parameter einer Funktion innerhalb der Funktion ein neuer Wert zugewiesen, so ändert sich auch der Wert der Variablen, welche in der aufrufenden Funktion als Parameter angegeben wurde.
- ❏ Eine Funktion, die mit dem Schlüsselwort *static* definiert wird, kann nur innerhalb des Moduls aufgerufen werden, in dem sie definiert wurde, nicht jedoch aus einem anderen Modul heraus.
- ❏ Es ist nicht möglich, Zeiger als Parameter an Funktionen zu übergeben.

d) Welche Aussage zum Thema Speicherzuteilung ist richtig? 2 Punkte

- ❏ Beim Halbierungsverfahren (*buddy*-Verfahren) kann keine externe Fragmentierung auftreten.
- ❏ Beim Halbierungsverfahren (*buddy*-Verfahren) kann es vorkommen, dass zwei nebeneinander liegende freie Speicherbereiche nicht miteinander verschmolzen werden können.
- ❏ Beim *first-fit*-Verfahren ist die Liste der freien Speicherbereiche aufsteigend nach der Größe der jeweiligen Bereiche sortiert.
- ❏ Beim *next-fit*-Verfahren muss entstehender Verschnitt immer am Ende der Freispeicherliste einsortiert werden.

e) Welche Aussage zu Monitoren ist richtig? 2 Punkte

- ❏ Bei allen Monitorkonzepten (*Hansen*, *Hoare*, *Mesa*) verlässt der Prozess, der den Eintritt eines Ereignisses anzeigt (Signalgeber), den Monitor unmittelbar nach der Signalisierung.
- ❏ Wartet ein Prozess in einem Monitor auf ein Ereignis, so muss er den Monitor während der Wartezeit zwingend freigeben, um einer Verklemmung vorzubeugen.
- ❏ Ein Monitor nach *Hansen* befreit bei der Signalisierung höchstens einen der wartenden Prozesse.
- ❏ Wird einem Prozess durch einen Monitor nach *Hoare* die Aufhebung seiner Wartebedingung signalisiert, so wird die Bedingung erneut ausgewertet; falsche Signalisierungen können also toleriert werden.

f) Welche Aussage zum Thema Adressräume ist richtig? 2 Punkte

- ❏ Im realen Adressraum sind alle theoretisch möglichen Adressen auch gültig.
- ❏ Der Zugriff auf eine nicht abgebildete Adresse führt in virtuellen Adressräumen immer zur Beendigung des Programms, welches den Zugriff durchführen wollte.
- ❏ Der virtuelle Adressraum kann nie größer sein als der im Rechner vorhandene Hauptspeicher.
- ❏ In einem virtuellen Adressraum sind alle Adressen gültig, es müssen jedoch nicht zu jedem Zeitpunkt alle Daten im Hauptspeicher vorliegen.

g) Man unterscheidet bei Programmunterbrechungen zwischen Traps und Interrupts. Welche Aussage dazu ist richtig? 2 Punkte

- ❏ Die Behandlung einer Unterbrechung muss nebeneffektfrei sein - der Prozessorstatus des unterbrochenen Programms muss somit vor der Behandlung gesichert und vor dem Rücksprung wiederhergestellt werden.
- ❏ Interrupts werden immer vom unterbrochenen Programm behandelt, Traps hingegen vom Betriebssystem.
- ❏ Traps können nicht durch Speicherzugriffe ausgelöst werden.
- ❏ Bei der mehrfachen Ausführung eines unveränderten Programms mit gleicher Eingabe treten Interrupts immer an den gleichen Stellen auf.

h) Welche der folgenden Aussagen über UNIX-Dateisysteme ist richtig? 2 Punkte

- ❏ Der Name einer Datei wird in ihrem Dateikopf (*inode*) gespeichert.
- ❏ In einem Verzeichnis darf es keinen Eintrag geben, der auf das Verzeichnis selbst verweist.
- ❏ Auf eine Datei in einem Dateisystem verweisen immer mindestens zwei *hard-links*.
- ❏ Innerhalb eines Verzeichnisses können mehrere Verweise auf den selben *inode* existieren, sofern diese unterschiedliche Namen haben.

i) Welche Aussage zu Seitenersetzungsstrategien ist richtig? 2 Punkte

- ❏ Für die Implementierung der Ersetzungsstrategie *LRU* werden keine Zeitgeber benötigt.
- ❏ Bei der Ersetzungsstrategie *FIFO* kann es nicht zu Seitenflattern kommen.
- ❏ Bei der Ersetzungsstrategie *LFU* wird die am seltensten referenzierte Seite aus dem Speicher verdrängt..
- ❏ Bei der Verwendung globaler Seitenersetzungsstrategien sind Seitenfehler vorhersagbar bzw. reproduzierbar.

j) Welche Aussage zu Prozessen und Threads ist richtig? 2 Punkte

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

k) Welche Seitennummer und welcher Offset gehören bei einstufiger Seitennummerierung und einer Seitengröße von 1024 Bytes zu folgender logischer Adresse: `0xc01a`? 2 Punkte

- ❏ Seitennummer `0xc`, Offset `0x1a`
- ❏ Seitennummer `0x30`, Offset `0x1a`
- ❏ Seitennummer `0xc0`, Offset `0x1a`
- ❏ Seitennummer `0xc01`, Offset `0xa`

Aufgabe 1.2: Mehrfachauswahl-Fragen (8 Punkte)

Bei den Multiple-Choice-Fragen in dieser Aufgabe sind jeweils m Aussagen angegeben, n ($0 \leq n \leq m$) Aussagen davon sind richtig. Kreuzen Sie **alle richtigen** Aussagen an. Jede korrekte Antwort in einer Teilaufgabe gibt einen halben Punkt, jede falsche Antwort einen halben 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 zum Thema persistenter Datenspeicherung sind richtig? 4 Punkte

- ❍ Bei kontinuierlicher Speicherung ist es immer problemlos möglich, bestehende Dateien zu vergrößern.
- ❍ Bei indizierter Speicherung kann es prinzipbedingt nicht zu Verschnitt kommen.
- ❍ Bei verketteter Speicherung mittels FAT-Ansatz kann die Verkettungsinformation redundant gespeichert werden, um die Fehleranfälligkeit zu reduzieren.
- ❍ Im Vergleich zu den anderen Verfahren ist bei indizierter Speicherung die Positionierzeit des Festplatten-Armes beim Zugriff auf alle Datenblöcke einer Datei minimal.
- ❍ Beim Einsatz von RAID 1 kann eine der beteiligten Platten ausfallen, ohne dass das Gesamtsystem ausfällt.
- ❍ Beim Einsatz von RAID 0 kann eine der beteiligten Platten ausfallen, ohne dass das Gesamtsystem ausfällt.
- ❍ Journaling-Dateisysteme garantieren, dass auch nach einem Systemausfall alle Metadaten wieder in einen konsistenten Zustand gebracht werden können.
- ❍ Festplatten eignen sich besser für sequentielle als für wahlfreie Zugriffsmuster.

b) Welche der folgenden Aussagen zur Einplanung von Prozessen sind richtig? 4 Punkte

- ❍ Bei kooperativer Einplanung kann es zur Monopolisierung der CPU kommen.
- ❍ Bei der Verwendung des *Round-Robin*-Verfahrens kann der Konvoi-Effekt nicht auftreten.
- ❍ Der Einsatz des *FCFS*-Verfahrens setzt kooperative Prozesse voraus.
- ❍ Die Verwendung probabilistischer Einplanungsverfahren ist nur möglich, wenn dem Planer alle Prozesse und ihre CPU-Stoßlängen im Voraus bekannt sind.
- ❍ In einem asymmetrischen Multiprozessorsystem ist der Einsatz asymmetrischer Einplanungsverfahren obligatorisch.
- ❍ Statische (*off-line*) Einplanungsverfahren sind besonders für den Einsatz in interaktiven Systemen geeignet.
- ❍ *Virtual-Round-Robin* benachteiligt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.
- ❍ Beim Einsatz des *multilevel-queue*-Verfahrens (*MLQ*) werden die Prozesse nach ihrem Typ in separate Bereitlisten aufgeteilt, die jeweils eine eigene lokale Einplanungsstrategie verwenden.

Aufgabe 2: zwitscher (60 Punkte)

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

Schreiben Sie ein Programm `zwitscher`, das auf dem TCP/IPv6-Port 2017 (`LISTEN_PORT`) einen Dienst anbietet, über den Benutzer Kurznachrichten verbreiten und abrufen können. Die Abarbeitung mehrerer paralleler Verbindungen soll von Arbeiter-Threads übernommen werden, die vom Haupt-Thread nach der Annahme einer Verbindung gestartet werden - die maximale Anzahl gleichzeitig erlaubter Threads wird dem Server als einziges Kommandozeilen-Argument übergeben. Jede Nachricht soll vom Dienst außerdem mit einer global eindeutigen, von 0 an aufsteigenden ID vom Typ `unsigned int` versehen werden (**Hinweis**: die maximale Länge der String-Repräsentation ist als Makro `UINT_LEN` gegeben).

Alle Nachrichten werden unterhalb des Verzeichnisses `"/etc/zwitscher/"` (`STORAGE_DIR`) in Unterverzeichnissen, die den Namen des jeweiligen Benutzers tragen, als eigene Dateien unter dem Namen `t<ID der Nachricht>` gespeichert. (zum Beispiel: `/etc/zwitscher/realDonaldDuck/t45`).

Ein Client sendet nach erfolgreicher Verbindung zunächst eine Zeile, die die auszuführende Aktion spezifiziert; der Dienst unterstützt dabei zwei Befehle:

- `PUT` *username* zeigt an, dass eine Kurznachricht an den Server gesendet werden soll. Der Benutzername hat dabei eine maximale Länge von 15 Zeichen (`MAX_NAME_LEN`). Die Nachricht selbst, die eine Maximallänge von 140 Zeichen (`MAX_MSG_LEN`) aufweisen darf, wird anschließend in einer eigenen Zeile an den Server geschickt.
- `GET` *username* liefert alle bisher zu einem Nutzernamen gespeicherten Kurznachrichten, jeweils durch Zeilenumbrüche getrennt, an den Client zurück.

Das Programm soll folgendermaßen strukturiert sein:

- Das Hauptprogramm initialisiert zunächst alle benötigten Datenstrukturen und nimmt auf einem Socket Verbindungen an. Wurde die maximale Anzahl laufender Threads erreicht, soll vor der Verbindungsannahme passiv gewartet werden. Eine erfolgreich angenommene Verbindung wird an einen neuen Thread, welcher in der Funktion `handleConnection` startet, weitergegeben.
- Funktion `void* handleConnection(void *argument)`: Die von den Threads ausgeführte Funktion. Zur passend übergebenen Verbindung werden zunächst durch einen Aufruf an die **von uns vorgegebene** Funktion `sock2fds` ein Array aus zwei `FILE *` zur einfacheren Kommunikation mit dem Client erzeugt - an der ersten Stelle des Arrays wird hierbei der Lesekanal zurückgegeben, an der zweiten Stelle der Schreibkanal. Zur Bearbeitung der Anfragezeile wird dann die Funktion `parseCommand` aufgerufen. Bei einem Fehler während der Bearbeitung soll dem Client eine kurze Fehlermeldung gesendet werden.
- Funktion `int parseCommand(FILE *rx, FILE *tx)`: Liest die Anfragezeile vom Client, wertet sie aus und ruft zur Abarbeitung die passende Funktion (`storeMessage` bzw. `getMessages`) auf. Im Falle des `PUT`-Kommandos muss außerdem die Nachricht vom Client eingelesen werden. Tritt ein Fehler bei der Auswertung oder in einer der aufgerufenen Funktionen auf, gibt `parseCommand` -1 zurück, im Erfolgsfall den Wert 0.
- Funktion `int storeMessage(const char *username, const char *message)`: Nach der Zuweisung der aktuellen ID wird zunächst eine temporäre Datei im Arbeitsverzeichnis des Servers angelegt, welche nur die ID als Namen trägt. Nachdem die übergebene Nachricht in der Datei gespeichert wurde, soll sie mit Hilfe der Funktion `rename` (siehe Manual-Seiten) an die passende Stelle (siehe oben) verschoben werden. Im Erfolgsfall gibt die Funktion den Wert 0 zurück, ansonsten wird -1 zurückgegeben.
- Funktion `int getMessages(const char *username, FILE *tx)`: Liefert alle Nachrichten, die zu einem Benutzer vorliegen, an den Client aus. Dazu wird das passende Unterverzeichnis nach regulären Dateien durchsucht, deren Name mit dem Zeichen `'t'` beginnt. Der Inhalt passender Dateien wird dann über den übergebenen Kommunikationskanal ausgeliefert. Im Erfolgsfall gibt dieFunktion den Wert 0 zurück, ansonsten wird -1 zurückgegeben.

Zur Koordination stehen Ihnen Semaphore mit den Funktionen `semCreate`, `P` und `V` zur Verfügung. Die Semaphor-Funktionen müssen Sie **nicht** selbst implementieren. Die Schnittstellen entsprechen dabei dem Modul, das sie aus den Übungen kennen; sie sind auf der folgenden Seite am Anfang des Programms deklariert.

Auf den folgenden Seiten finden Sie ein Gerüst für das beschriebene Programm. In den Kommentaren sind nur die wesentlichen Aufgaben der einzelnen zu ergänzenden Programmteile beschrieben, um Ihnen eine gewisse Leitlinie zu geben. Es ist überall sehr großzügig Platz gelassen, damit Sie auch weitere notwendige Anweisungen entsprechend Ihrer Programmierung einfügen können.

Einige wichtige Manual-Seiten liegen bei - es kann aber durchaus sein, dass Sie bei Ihrer Lösung nicht alle diese Funktionen oder gegebenenfalls auch weitere Funktionen benötigen.


```
#include <dirent.h>
#include <errno.h>
#include <netinet/in.h>
#include <pthread.h>
#include <signal.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>
#include "sem.h"

#define LISTEN_PORT  2017
#define MAX_MSG_LEN  140
#define MAX_NAME_LEN 15
#define STORAGE_DIR  "/etc/zwitscher/"
#define UINT_LEN     (sizeof(unsigned int) * 8)

SEM *semCreate(int initVal);
void P(SEM *sem);
void V(SEM *sem);

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

// Erzeugt zum uebergegebenen Socket ein dynamisch allokiertes,
// zwei-elementiges FILE*-Array (Index 0: Lesen, Index 1: Schreiben).
// Im Fehlerfall wird NULL zurueckgegeben.
static FILE** sock2fds(int sock) {
    FILE *rx = fdopen(sock, "r");
    if (!rx) {
        close(sock);
        return NULL;
    }
    int sock2 = dup(sock);
    if (sock2 < 0) {
        close(sock);
        return NULL;
    }
    FILE *tx = fdopen(sock2, "w");
    if(!tx) {
        fclose(rx);
        close(sock2);
        return NULL;
    }
    FILE **farray = calloc(2, sizeof(FILE *));
    if (farray) {
        farray[0] = rx;
        farray[1] = tx;
    }
    return farray;
}
```

```
// Funktions- und Strukturdeklarationen, globale Variablen, etc.
```

```
// Funktion main()
```

```
  // Argumente pruefen, Initialisierungen, etc.
```



```
  // Socket erstellen und fuer Verbindungsannahme vorbereiten
```

```
// Verbindungen annehmen, Threads starten
```

M:

```
// Ende Funktion main
```



```
// Funktion handleConnection
```

H:

```
// Ende Funktion handleConnection
```

```
// Funktion parseCommand
```


```
// Ende Funktion parseCommand
```



```
// Funktion storeMessage
```


```
// Ende Funktion storeMessage
```

```
// Funktion getMessages
```

```
// Ende Funktion getMessages
```

G:

Aufgabe 3: (20 Punkte)

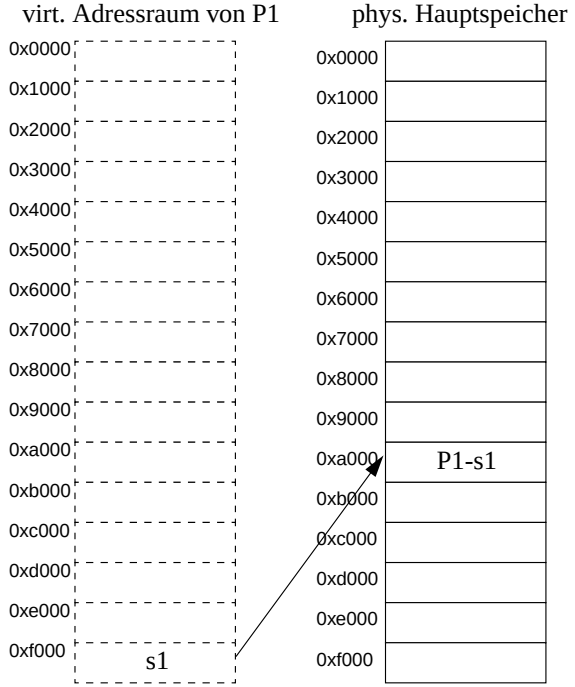
In einem System mit Seitenadressierung (paged address space), Adresslänge = 16 Bit, Seitengröße = 4 KiBi Bytes, Hauptspeichergröße = 64 KiBi Bytes wird ein Programm durch einen Prozess P1 ausgeführt. *(zur Erinnerung :-) : $4096_{10} = 1000_{16}$)*

Die erste Seite eines virtuellen Adressraums wird grundsätzlich nicht genutzt. Das darauf folgende Textsegment des Programms umfasst eine Seite (t1), das Datensegment umfasst ebenfalls eine Seite (d1) direkt im Anschluss daran. Das Stacksegment umfasst eine Seite (s1) ganz am Ende des virtuellen Adressraums.

Den Seiten sind folgende Seitenrahmen im Hauptspeicher zugewiesen:
P1-t1: 0x5000, P1-d1: 0xe000, P1-s1: 0xa000

Das (sehr kleine) Betriebssystem (BS) belegt die ersten 4 Seitenrahmen im physikalischen Hauptspeicher.

a) Tragen Sie die den Aufbau des virtuellen Adressraums (wo liegen welche Seiten?), die bekannte Belegung der Seitenrahmen des physikalischen Hauptspeichers und die Abbildung (Pfeile) analog zu dem Beispiel von Seite s1 in der nachfolgenden Skizze ein. (3 Punkte)



b) Skizzieren Sie die Datenstrukturen, die die MMU für die Umsetzung des virtuellen Adressraums von P1 benötigt, tragen Sie sinnvolle Zugriffsrechte für die einzelnen Seiten ein und veranschaulichen Sie in Ihrer Skizze, wie die logische Adresse 0x1338 umgesetzt wird. (10 Punkte)

c) In Prozess P1 wird folgende Anweisung ausgeführt:

```
char *p = malloc(8000);
```

Erläutern Sie, was das Betriebssystem an dieser Stelle tun muss, und tragen Sie in den Skizzen der Teilaufgaben a) und b) ein, was sich ändert. Wenn erforderlich, wählen Sie selbst geeignete Adressen. Markieren Sie die Eintragungen jeweils mit einem ⓓ . (7 Punkte)

Aufgabe 4: (10 Punkte)

a) Was versteht man unter einer Verklemmung und was sind die Bedingungen, damit eine Verklemmung auftreten kann? (6 Punkte)

b) Die im folgenden skizzierten Programmstücke werden von 2 Threads parallel ausgeführt. Hierbei wird jeweils ein Datum aus einem Puffer gelesen, in einen anderen Puffer eingefügt und - falls dies erfolgreich war - das Datum anschließend aus dem ersten Puffer entfernt. Der Zugriff auf die Puffer darf nicht gleichzeitig von mehreren Threads erfolgen, weshalb die Zugriffe durch Semaphore synchronisiert sind.

```
thread1(){
  P(&sem_buf1);
  item = get_item(&buf1);
  P(&sem_buf2);
  if (put_item(&buf2, item) >  0 ) {
    remove_item(&buf1);
    return_val = 1;
  }
  else
    return_val = -1;

  V(&sem_buf2);
  V(&sem_buf1);
  return return_val;
}
```

```
thread2(){
  P(&sem_buf2);
  item = get_item(&buf2);
  P(&sem_buf1);
  if (put_item(&buf1, item) >  0 ) {
    remove_item(&buf2);
    return_val = 1;
  }
  else
    return_val = -1;

  V(&sem_buf1);
  V(&sem_buf2);
  return return_val;
}
```


Welches grundlegende Problem in Bezug auf Verklemmungen besteht in der angegebenen Implementierung? Beziehen Sie sich bei Ihrer Antwort auf die in a) beschriebenen Voraussetzungen für Verklemmungen. Geben Sie zwei mögliche Lösungen für dieses Problem an. (4 Punkte)