Aufgabe 1: (30 Punkte)

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

Wollen Sie eine Multiple-Choice-Antwort korrigieren, kreisen sie bitte die falsche Antwort ein und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) Welche der folgenden Aussagen zu statischem bzw. dynamischem Binden ist **falsch**? 2 Punkte

- ❏ bei dynamischem Binden werden alle Adressen zum Bindezeitpunkt aufgelöst
- ❏ bei dynamischem Binden können auch zum Übersetzungszeitpunkt alle bekannten Adressbezüge bereits vollständig aufgelöst werden
- ❏ beim statischen Binden werden alle Adressen zum Bindezeitpunkt aufgelöst
- ❏ statisch gebundene Programme können zum Ladezeitpunkt an beliebige Speicheradressen platziert werden

b) Man unterscheidet zwischen privilegierten und nicht-privilegierten Maschinenbefehlen. Welche Aussage ist **richtig**? 2 Punkte

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

c) Welche Aussage zu Speicherzuteilungsverfahren ist **falsch**? 2 Punkte

- ❏ die worst-fit-Strategie kann einem mit der kürzesten Antwortzeit einen ausreichend großen Speicherbereich liefern
- ❏ best-fit arbeitet mit konstanter Komplexität und ist deshalb das beste Verfahren
- ❏ first-fit hat konstanten Aufwand bei der Verschmelzung von Lücken
- ❏ bei buddy-Verfahren gibt es keinen externen Verschnitt

d) Mit logischen Adressräumen kann man mehrere Zwecke erreichen. Was gehört **nicht** dazu? 3 Punkte

- ❏ Schutzmechanismus: man kann die Auswirkungen von Berechnungsfehlern oder technischen Fehlern (wie z. B. Bitkipper) begrenzen.
- ❏ Sicherheit: man kann unbefugten Zugriff auf Daten verhindern.
- ❏ Virtualisierung: man kann in einem Programmlauf mehr Speicher adressieren, als physiklaisch vorhanden ist.
- ❏ bessere Verwaltung: man kann Speicherbereiche mit unterschiedlicher Bedeutung voneinander abgrenzen.

e) Wodurch kann es zu Seitenflattern kommen? 2 Punkte

- ❏ durch Programme, die eine Defragmentierung auf der Platte durchführen
- ❏ wenn sich zu viele Prozesse im Zustand blockiert befinden
- ❏ wenn bei einer nicht-verdrängenden Scheduling-Strategie die Zahl der von den Prozessen aktiv genutzten Seiten die Zahl der verfügbaren Seitenrahmen übersteigt
- ❏ wenn zu viele Prozesse im Rahmen der mittelfristigen Einplaung ausgelagert (swap-out) wurden

f) Welche Aussage zum Thema Kooperatives Scheduling ist richtig? 2 Punkte

- ❏ Kooperatives Scheduling ist in Mehrbenutzersystemen unproblematisch, so lange sich die Benutzer des Systems kennen und bereit sind, sich kooperativ zu verhalten.
- ❏ Bei kooperativem Scheduling wird Prozessen die sich nicht kooperativ verhalten (z. B. in einer Endlosschleife rechnen) zwangsweise der Prozessor entzogen.
- ❏ Kooperatives Scheduling ist im Mehrprogrammbetrieb gut einsetzbar, weil das Betriebssystem im Rahmen der Systemaufrufe der beteiligten Prozesse Prozessumschaltungen vornehmen kann und damit eine Monopolisierung der CPU durch einen Prozess in jedem Fall automatisch verhindert wird.
- ❏ Programme, die unter kooperativem Scheduling zum Einsatz kommen, müssen auf diese Situation entsprechend vorbereitet sein (z. B. durch Einstreuen von regelmäßigen Aufrufen an das Betriebssystem), wenn eine Monopolisierung der CPU vermieden werden soll.

g) Man unterscheidet kurz- mittel- und langfristige Prozesseinplanung. Welche Aussage hierzu ist **richtig**? 3 Punkte

- ❏ Wenn ein Prozess auf Daten von der Platte wartet, wird er in den Zustand "blockiert" versetzt. Da die Einlagerung von Daten von der Platte sehr lange dauert (mehrere Millisekunden gegenüber einem CPU-Takt im GHz-Bereich), ist diese Situation der mittelfristigen Einplanung zuzuordnen.
- ❏ Wenn der Adressraum eines laufbereiten Prozesses aufgrund von Speichermangel ausgelagert wird ("swap-out") wird der Prozess im Rahmen der mittelfristigen Einplanung in den Zustand "blockiert" überführt bis die Daten wieder eingelagert werden.
- ❏ Wenn ein Prozess auf einen Seitenfehler (page fault) trifft, wird er im Rahmen der kurzfristigen Einplanung in den Zustand "blockiert" überführt bis die Seite eingelagert wurde.
- ❏ Wenn ein Prozess auf einen Seitenfehler (page fault) trifft, wird er im Rahmen der kurzfristigen Einplanung in den Zustand "schwebend bereit" überführt, weil er ja unmittelbar nach dem Einlagern der Seite wieder weiterlaufen kann.

h) Welche der folgenden Aussagen bzgl. Threads ist **falsch**? 2 Punkte

- ❏ Die Einlastung (das Dispatching) eines Threads ist eine privilegierte Operation und kann deshalb grundsätzlich immer nur durch das Betriebssystem vorgenommen werden.
- ❏ Das Betriebssystem ist nicht in der Lage, einen einzelnen User-Level Thread bei einer fehlerhaften Operation (z. B. Segmentation fault) gezielt abzubrechen.
- ❏ Ein Anwendungsprogrammierer kann die Schedulingstrategie für seine User-Level Threads selbst programmieren.
- ❏ Die Erzeugung eines Kernel-Level Threads ist teurer als die Erzeugung eines User-Level Threads.

i) Es gibt verschiedene Ursachen, wie Nebenläufigkeit in einem System entstehen kann (gewollt oder auch ungewollt). Was gehört **nicht** dazu? 2 Punkte

- ❏ durch Interrupts
- ❏ durch preemptives Scheduling
- ❏ durch Traps
- ❏ durch Threads auf einem Multiprozessorsystem

j) Bei der Behandlung von Ausnahmen (Traps oder Interrupts) unterscheidet man zwei Bearbeitungsmodelle. Welche Aussage hierzu ist richtig? 2 Punkte

- ❏ Das Wiederaufnahmemodell dient zur Behandlung von Interrupts (Fortführung des Programms nach einer zufällig eingetretenen Unterbrechung). Bei einem Trap ist das Modell nicht sinnvoll anwendbar, da ein Trap ja deterministisch auftritt und damit eine Wiederaufnahme des Programms sofort wieder den Trap verursachen würde.
- ❏ Nach dem Beendigungmodell 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.
- ❏ Interrupts dürfen auf keinen Fall nach dem Beendigungsmodell behandelt werden, weil überhaupt kein Zusammenhang zwischen dem unterbrochenen Prozess und dem Grund des Interrupts besteht.
- ❏ Das Betriebssystem kann Interrupts, die in ursächlichem Zusammenhang mit dem gerade laufenden Prozess stehen, nach dem Beendigungsmodell behandeln, wenn eine sinnvolle Fortführung des Prozesses nicht mehr möglich ist.

k) In einem kritischen Abschnitt soll ein Element aus einer doppelt verketteten Liste entnommen und in eine andere, einfach verkettete Liste eingehängt werden. Welches Synchronisationsverfahren ist hierbei für den Einsatz in einem Multiprozessorsystem **am besten geeignet**? 2 Punkte

- ❏ Einseitige Synchronisation (wie Unterbrechungssperren), weil ja nur kurzzeitig (wenige Maschinenbefehle lang) der Zugriff auf die Listen durch andere Prozesse verhindert werden muss.
- ❏ Nicht-blockierende Synchronisationsverfahren, weil sie am besten mit den parallelen Abläufen in einem Multiprozessorsystem zurechtkommen.
- ❏ Eine Lock-Variable zur Absicherung des kritischen Abschnitts, verbunden mit aktivem Warten falls der kritische Abschnitt gerade belegt ist.
- ❏ Ein Monitor, verbunden mit einer Abgabe des Prozessors, falls der Monitor gerade belegt ist.

l) Welche Aussage zum Thema Monitor ist falsch? 2 Punkte

- ❏ Bei einem Hoare’schen Monitor kann die Neuauswertung der Wartebedingung entfallen, weil kein anderer Faden nach der Signalisierung den Monitor betreten konnte.
- ❏ pthread_wait und pthread_broadcast realisieren die wait- und signal-Primitiven Hansen’scher Monitore
- ❏ pthread_wait und pthread_signal realisieren Hoare’sche Monitore
- ❏ bei Hansen’schen Monitoren entscheidet ausschließlich der Scheduler wer nach einer Monitorfreigabe den Monitor als nächstes betritt.

m) Betrachten Sie folgende Aussagen zur Speicherung der Daten einer Datei auf Festplatte. Welche Aussage ist **falsch**? 2 Punkte

- ❏ Eine kontinuierliche Speicherung der Daten in aufeinanderfolgenden Blöcken erhöht die Lese- und Schreibleistung gegenüber verstreuter Speicherung.
- ❏ Bei kontinuierlicher Speicherung kann das Auffinden eines genügend großen zusammenhängenden Speicherbereiches schwierig sein.
- ❏ Kontinuierliche Speicherung ist gänzlich unmöglich, falls Dateien dynamisch erweiterbar sein sollen.
- ❏ Bei kontinuierlicher Speicherung kann die Ortsinformation lediglich aus der Nummer des ersten Plattenblocks und der Dateilänge bestehen.

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

- ❏ Bei RAID 4 Systemen wird die Paritätsinformation gleichmäßig über alle beteiligten Platten verteilt.
- ❏ Der Lesezugriff auf ein gestreiftes Plattensystem insbesondere auch auf ein RAID 5 System ist 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 RAID 5 Systemen sind mindestens 5 Festplatten nötig.

Aufgabe 2: (60 Punkte)

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

a) Implementieren Sie ein Server-Programm **`fdserver`**, das auf Anfrage den Inhalt von Dateien oder Directories über eine TCP/IP-Verbindung ausliefert. Wir gehen davon aus, dass dieser Server mit einer sehr hohen Anfragerate belastet wird, deshalb sollen die Anfragen nicht in separaten Sohnprozessen, sondern durch bereitstehende Threads behandelt werden.

Das **`fdserver`**-Programm erhält als Aufrufparameter die Nummer des Ports, auf dem Verbindungen angenommen werden.

Das Programm soll folgendermaßen arbeiten:

- Zunächst werden vom Haupt-Thread die bereitzuhaltenden Threads gestartet. Sie erledigen ihre Aufgabe in der Funktion **`serve`**.
  Bedenken Sie, dass jeder Thread seinen eigenen Stack hat (die dort liegenden Variablen sind also Thread-lokal), aber alle Threads ein gemeinsames Datensegment haben.
- Die Zahl der Threads wird im Quellcode über ein Makro **`NTHREAD`** mit Wert 25 festgelegt.
- Anschließend nimmt das Programm Verbindungen über beliebige IP-Adressen auf dem angegebenen Port an.
  Bis zu 20 Verbindungswünsche sollen anstehen können, bevor Verbindungswünsche abgewiesen werden.
- Wurde eine Verbindung angenommen, wird einer der bereitstehenden Threads mit der Bearbeitung beauftragt. Die Beauftragung erfolgt, indem der Socket-Deskriptor der angenommenen Verbindung in einen Ringpuffer (der Größe **`NTHREAD`**) eingetragen wird.
- Alle Threads warten in der Funktion **`serve`** an diesem Ringuffer. Nachdem ein Socket-Deskriptor eingetragen wurde, wird ein Thread deblockiert, der dann die Verbindung bearbeitet.
- Sollte der Ringpuffer voll sein, werden so lange keine Verbindungen angenommen.
- Die Koordinierung zwischen Haupt-Thread und den **`serve`**-Threads erfolgt durch zählende Semaphore. Implementieren Sie hierzu ein Semaphor-Modul **`sem.c`** mit den Funktionen **`sem_init`**, **`P`** und **`V`** (Schnittstellen siehe Datei **`sem.h`** auf der nächsten Seite).
- Beachten Sie auch, dass zwar nur der Haupt-Thread Aufträge in den Ringpuffer einträgt, aber evtl. mehrere Threads gleichzeitig Aufträge entnehmen wollen.
- Ein Client sendet über die Verbindung die Namen von Dateien oder Directories (komplette Pfandnamen, String mit max. 1024 Byte), deren Inhalt er geliefert bekommen möchte.
- Der **`serve`**-Thread entnimmt den Socketdeskriptor aus dem Ringpuffer, erzeugt einen **`FILE`**-Zeiger daraus, liest den Datei-/Directory-Namen von der Verbindung und ruft die Funktion **`void handle(FILE *socket, char path[])`** zur weiteren Bearbeitung auf. Anschließend wird der Socket geschlossen und der nächste Auftrag erwartet.
- In der **`handle`**-Funktion wird ermittelt, ob **`path`** eine reguläre Datei oder ein Directory ist. Zur weiteren Bearbeitung wird entweder **`fhandle(FILE *socket, char path[])`** oder **`dhandle(FILE *socket, char path[])`** aufgerufen (bei anderen Dateitypen oder Fehlern bei der Auftragsbearbeitung wird eine aussagekräftige Fehlermeldung auf die Socketverbindung ausgegeben und die Bearbeitung dieses Auftrags abgebrochen).
- Die **`fhandle`**-Funktion gibt einfach den Inhalt der regulären Datei auf den Socket aus, die **`dhandle`**-Funktion gibt die Namen aller in dem Directory verzeichneten Dateien (jeweils auf einer Zeile) auf den Socket aus.

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 Leitlinie zu geben. Es ist überall 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 ggf. auch weitere Funktionen benötigen.

/** Datei sem.h *****/***

```
#include <pthread.h>

/* Semaphor-Datentyp */
typedef struct {
        int s;             /* Semaphor-Zustand */
        pthread_mutex_t m;
        pthread_cond_t c;
} SEM;

/* Schnittstellen */

SEM *sem_init(int n); /* lege neuen Semaphor an, initialisiere
                         ihn mit Wert n und liefere Zeiger
                         darauf zurück */
void P(SEM *s);      /* P und V dekrementieren bzw.  */
void V(SEM *s);      /* inkrementieren jeweils um 1 */
```


/** Modul sem.c *****/***

```
#include <malloc.h>
#include "s.h"

/* Funktion sem_init */
SEM *sem_init(int n) {
```

```
/* Funktion P */
void P(SEM *s) {
```

```
/* Funktion V */
void V(SEM *s) {
```

K:

```
/***** Modul fdserver.c *****/
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <dirent.h>
#include <sys/stat.h>
#include <pthread.h>
#include <string.h>

#include "s.h"

/* Funktionsdeklarationen, globale Variablen */

void *serve(void *arg);
void handle(FILE* socket, char path[]);
void fhandle(FILE *socket, char path[]);
void dhandle(FILE *socket, char path[]);
```

A:


```
/* Funktion main */     {

    /* lokale Variablen und was man sonst am
       Anfang so vorbereitet */

    /* Semaphore erzeugen und initialisieren */
```

A: K:

```
   /* serve-Threads erzeugen */
```

```
   /* Socket oeffnen und vorbereiten */
```

A:
S:


```
   /* Schleife: Verbindungen annehmen und Auftraege erteilen */
```

```
   /* Ende von main - wird nie erreicht */
   return 0;
}
```

S:
K:

/* Funktion serve */

/* socket aus Puffer entnehmen */

/* Datei-/Directorynamen lesen und bearbeiten */

K:
B:

/* Funktion handle */

```
void handle(FILE *s, char path[]) {
```

/* Funktion fhandle */

```
void fhandle(FILE *s, char path[]) {
```

B:

```
/* Funktion dhandle */
void dhandle(FILE *s, char path[]) {
```

b) Schreiben Sie ein Makefile zum Erzeugen des fdserver-Programms.
Ein Aufruf von

```
make fdserver
```

soll das Programm erzeugen, ein Aufruf von

```
make clean
```

soll das fdserver-Programm und evtl. erzeugte **.o**-Dateien entfernen.

B:
M:

Aufgabe 3: (30 Punkte)

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

Die erste Seite des virtuellen Adressraums wird nicht genutzt. Das Textsegment des Prozesses umfasst 2 Seiten, das Datensegment umfasst ebenfalls 2 Seiten, direkt im Anschluss daran. Das Stacksegment umfasst eine Seite ganz am Ende des virtuellen Adressraums.

Die Seiten des Textsegments liegen im Hauptspeicher in den Seitenrahmen (=Kacheln) auf Adresse 0x4000 und 0x7000, die erste Seite des Datensegments liegt in Seitenrahmen 0x6000, die zweite Seite ist ausgelagert und liegt auf der Platte im Block 0x37000, die Seite des Stacks liegt in Seitenrahmen 0x9000. Das (sehr kleine) Betriebssystem belegt die ersten 4 Seitenrahmen.

Die Seitenrahmen von 0xa000 bis 0xf000 sind durch einen anderen Prozess belegt.

a) Skizzieren Sie den Aufbau des virtuellen Adressraums des Prozesses und den Aufbau und die Belegung des Hauptspeichers sowie die Abbildungen dazwischen.

b) Die Seiten des Textsegments seien read-only in den Adressraum abgebildet, alle anderen Seiten zum Lesen und Schreiben. Die Ausführung von Maschinenbefehlen aus dem Daten- oder Stacksegment ist nicht zulässig.

Skizzieren Sie die Datenstrukturen, die die MMU für die Umsetzung des beschriebenen virtuellen Adressraums benötigt und wie dabei die logische Adresse 0x3240 umgesetzt wird.

c) In dem Programm stehen folgende Anweisungen:

```
void f1() {
  static int *p = (int *)0x4018;
  int i;
  ...
  *p = 41;
  i = 5;
  ...
```

Beschreiben Sie, was bei der Ausführung des Prozesses passiert, wenn er die beiden obigen Zuweisungen ausführt:

c1) Welche Aktivitäten laufen der Reihe nach in der Anwendung und im Betriebssystem ab? (nummerieren Sie diese Schritte).

c2) Welche Wechsel des Prozesszustands finden hierbei bei welchen Schritten statt?

c3) Tragen Sie in Ihre Skizzen auf Seite 18 und 19 die Änderungen ein, die im Laufe der Ausführung der Anweisungen erfolgen und markieren Sie die entsprechenden Stellen mit (c3). Tragen Sie dabei auch die Orte ein (ungefähr), an denen die Werte 41 und 5 im Hauptspeicher hingeschrieben werden.