Aufgabe 1: (18 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 Aussage zum Thema Betriebsarten ist **richtig**? 2 Punkte

- ❏ Beim Stapelbetrieb können keine globalen Variablen existieren, weil alle Daten im Stapel-Segment (Stack) abgelegt sind.
- ❏ Echtzeitsysteme findet man hauptsächlich auf großen Serversystemen, die eine enorme Menge an Anfragen zu bearbeiten haben.
- ❏ Mehrzugangsbetrieb ist nur in Verbindung mit CPU- und Speicherschutz sinnvoll realisierbar.
- ❏ Mehrprogrammbetrieb ermöglicht die simultane Ausführung mehrerer Programme innerhalb desselben Prozesses.

b) Welche Aussage zum Thema Adressraumschutz ist **richtig**? 2 Punkte

- ❏ Bei allen Verfahren des Adressraumschutzes führt jeder Zugriff auf eine ungültige Speicheradresse zu einem Trap.
- ❏ Beim Adressraumschutz durch Abteilung wird der logische Adressraum in mehrere Segmente mit unterschiedlicher Semantik unterteilt.
- ❏ Beim Adressraumschutz durch Eingrenzung ist es prinzipbedingt nicht möglich, dass mehrere Prozesse auf ein Stück gemeinsamen Speichers zugreifen.
- ❏ In einem segmentierten Adressraum kann zur Laufzeit kein weiterer Speicher mehr dynamisch nachgefordert werden.

c) Welche Aussage zu Zeigern ist **richtig**? 2 Punkte

- ❏ Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden.
- ❏ Die Übergabesemantik für Zeiger als Funktionsparameter ist call-by-reference.
- ❏ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.
- ❏ Zeiger vom Typ `void*` existieren in C nicht, da solche "Zeiger auf Nichts" keinen sinnvollen Einsatzzweck hätten.

d) Was ist ein Stack-Frame? 2 Punkte

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

e) Welche Aussage zum Thema Prozesszustände ist **falsch**? 2 Punkte

- ❏ Nach seiner Beendigung geht ein Prozess in den Zustand *beendet* über. In diesem Zustand wird ein Prozess auch als Zombie-Prozess bezeichnet.
- ❏ Es können sich maximal soviele Prozesse im Zustand *laufend* befinden, wie physikalische Prozessorkerne im System existieren.
- ❏ Nach Wegfall der Blockadebedingung geht ein Prozess direkt in den Zustand *laufend* über.
- ❏ Ein Prozess kann nur durch eigene Aktivität in den Zustand *blockiert* gelangen.

f) Welche Aussage zum Thema Prozesse/Threads ist **richtig**? 2 Punkte

- ❏ Beim federgewichtigen Prozess bilden Adressraum und Prozess eine Einheit.
- ❏ Leichtgewichtige Prozesse müssen vom Betriebssystem speziell unterstützt werden.
- ❏ Eine gleichzeitige Ausführung mehrere schwergewichtiger Prozesse auf verschiedenen Prozessoren ist nicht möglich.
- ❏ Die Einlastung eines federgewichtigen Prozesses ist eine privilegierte Operation und erfordert Unterstützung der Betriebssystem.

g) Welche Information wird **nicht** im Dateikopf (Inode) eines typischen UNIX-Dateisystems gespeichert? 2 Punkte

- ❏ Datum und Zeit der letzten Änderung
- ❏ Datum und Zeit des letzten Zugriffs
- ❏ Nummer des Inodes
- ❏ Zugriffsrechte für den Dateibesitzer

h) Welche Aussage zum Thema Hard-Links ist **falsch**? 2 Punkte

- ❏ Auf jede Datei existiert mindestens ein Hard-Link
- ❏ Auf jedes Verzeichnis existieren mindestens zwei Hard-Links
- ❏ Jeder Hard-Link besitzt in seinem Kontext einen eindeutigen Namen
- ❏ Hard-Links dürfen nur vom Systemadministrator angelegt werden

i) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist **richtig**? 2 Punkte

- ❏ Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzähler, Register, Stack).
- ❏ Wenn ein Programm nur einen aktiven Ablauf enthält, nennt man diesen Prozess, enthält das Programm mehrere Abläufe, nennt man diese Threads.
- ❏ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann aber auch mehrere verschiedene Programme ausführen
- ❏ Ein Programm kann immer nur von einem Prozess ausgeführt werden

Aufgabe 2: mbump (47 Punkte)

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

Schreiben Sie ein Programm `mbump` (BackUp for Multi-Processors), das alle regulären Dateien eines Verzeichnisses rekursiv auf ein Bandlaufwerk sichert.

Das Programm arbeitet mit mehreren POSIX-Threads, wobei jeder Thread die Sicherung der Dateien genau eines Verzeichnisses übernimmt.

Der Inhalt einer zu sichernden Datei wird jeweils zusammen mit dem Dateinamen und der tatsächlichen Größe der Datei (filesize) über einen Ringpuffer (Auftragspuffer) als Backup-Auftrag an den Haupt-Thread übergeben. Dieser sichert alle Backup-Aufträge nacheinander auf das Bandgerät. Sie können vereinfachend davon ausgehen, dass keine Datei größer ist, als der im Auftragspuffer vorgesehene Platz.

Die Speicherbereiche für die Backup-Aufträge werden wiederverwendet. Hierfür wird beim Programmstart eine feste Zahl von `BJOB`-Strukturen allokiert und in einen zweiten Ringpuffer (Vorhaltepuffer) eingetragen und somit den Arbeiterthreads zur Verfügung gestellt. Der Haupt-Thread trägt nach dem Sichern eines Auftrags den zugehörigen Speicherbereich wieder in den Vorhaltepuffer ein.

Das Programm soll folgendermaßen arbeiten:

- Das Programm bekommt als Argumente den Gerätenamen des Bandlaufwerks und das zu sichernde Verzeichnis übergeben. Der Haupt-Thread ruft zur Erzeugung des ersten Arbeiterthreads die Funktion `spawnThread` auf und füllt den Vorhaltepuffer. Anschließend entnimmt der Haupt-Thread die abzuarbeitenden Aufträge aus dem Auftragspuffer und schreibt sie mit allen Informationen (Dateiname,filesize und Daten) auf das Bandlaufwerk. Das Schreiben auf das Bandlaufwerk funktioniert wie das Schreiben in eine gewöhnliche reguläre Datei. Das Programm beendet sich, wenn keine Arbeiterthreads mehr existieren und alle Aufträge auf das Band gesichert wurden.
- Funktion `void spawnThread(char *dirname);` Erzeugt einen neuen Arbeiterthread zum Durchsuchen des Verzeichnisses `dirname`.
- Thread-Startfunktion `void *tstart(char *dirname);` `tstart` durchsucht das übergebene Verzeichnis und ignoriert hierbei die Einträge `.` und `..` sowie alle Einträge, bei denen es sich nicht um eine reguläre Datei oder ein Verzeichnis handelt. Für jede gefundene reguläre Datei wird die Funktion `backupFile` aufgerufen. Für jedes Unterverzeichnis wird die Funktion `spawnThread` zur Erzeugung eines neuen Arbeiterthreads für dieses Unterverzeichnis aufgerufen.
- Funktion `void backupFile(char *filename);` Liest die übergebene Datei ein und trägt Aufträge in den Auftragspool ein.
- Für die Ringpuffer steht Ihnen die aus den Übungen bekannte Schnittstelle (siehe auch Manual-Seiten) zur Verfügung. Sie müssen diese nicht selbst implementieren.
  Zusätzlich liefert die `bb_get`-Funktion als Ergebnis `NULL`, wenn der Puffer leer ist und keine Arbeiterthreads mehr existieren, die ihn noch füllen könnten.

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 Programmanweisungen 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.

```
/* includes */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <pthread.h>
#include <limits.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include "bbuffer.h"


#define BUFSIZE  32
#define DATASIZE 1048576

typedef struct BJOB {
        char   filename[PATH_MAX];
        size_t filesize;
        char   data[DATASIZE];
} BJOB;
```

/* Funktionsdeklarationen, globale Variablen */

/* Funktion main */

/* Argumente pruefen und weitere Initialisierungen */

/* Auftragspuffer und Vorhaltepuffer allokieren */

/* Bandgeraet oeffnen */

/* ersten Arbeiterthread erzeugen */

```
/*  Vorhaltepuffer fuellen */

/*  Backup-Auftraege auf Bandgeraet schreiben */

/*  alle Auftraege geschrieben, abschliessen */

/* Ende Funktion main */
```

A:


```
/* Funktion spawnThread */
```

S:

```
/* Funktion tstart */
```

```
    /*  Directory oeffnen und Eintraege lesen */
```



```
    /* Directory schliessen */
```

```
    /* Ressourcen freigeben, Thread beenden */
```

T:

```
/* Funktion backupFile */
```

B:

Aufgabe 3: (18 Punkte)

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

a) Beschreiben Sie diese beiden Arten:
   i) wodurch entstehen diese Ausnahmesituationen?
   ii) wodurch unterscheiden sie sich?
   iii) geben Sie jeweils zwei Beispiele an.
   (8 Punkte)

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

c) Wie wird eine Ausnahmebehandlung von der CPU abgewickelt? (4 Punkte)

Aufgabe 4: (7 Punkte)

Gegegeben sei das nachfolgende Programmfragment.

Skizzieren Sie den Aufbau des logischen Adressraums eines Prozesses, der ein Programm ausführt, das dieses Fragment enthält.

- Wozu werden welche Segmente angelegt?
- Zeichnen Sie für jede Variable und den formalen Parameter par ein, wo diese ungefähr im logischen Adressraum zu finden sein werden.
- Wohin zeigt der Zeiger p (ungefähre Position einzeichnen!)?

```
static int ga;

int f1(int par) {
  static int lb=5;
  int *p;

  p = malloc(100 * sizeof(int));

}
```