Aufgabe 1: (11 Punkte)

In dieser Aufgabe sind jeweils m Aussagen angegeben. Davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie jeweils an, ob die entsprechende Aussage richtig oder falsch ist.
Jede korrekte Antwort gibt 0.5 Punkte, jede falsche Antwort 0.5 Punkte Abzug. Nicht beantwortetete Aussagen gehen neutral in die Bewertung ein. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, 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 genannten Aufgaben gehören zwingend in den Betriebssystemkern?

Richtig	Falsch	
☐	☐	Scheduling (Prozessplanung)
☐	☐	Isolation zwischen Prozessen
☐	☐	Behandlung von Hardwareunterbrechungen
☐	☐	Multiplexing von virtuellen Ressourcen

b) Bereich: Dateisysteme

Richtig	Falsch	
☐	☐	In einem UNIX-Dateisystem sind Dateiobjekte stets in einer Baumstruktur angeordnet.
☐	☐	Ein Dateideskriptor repräsentiert eine prozesslokale Zugriffsbefähigung auf eine Datei.
☐	☐	Verzeichnisse definieren den Kontext für die (hierarchische) Namensauflösung.
☐	☐	Hardlinks beeinflussen den Linkzähler im Indexknoten der Zieldatei.

c) Bereich: Prozesszustände

Richtig	Falsch	
☐	☐	Es ist kein direkter Übergang von „blockiert“ in „bereit“ möglich.
☐	☐	Übergang von „blockiert“ in „bereit“ bedeutet: Ein anderer Prozess wurde vom Betriebssystem verdrängt und der aktuelle Prozess wird nun auf der CPU eingelastet.
☐	☐	Ein Prozess wird wegen eines ungültigen Speicherzugriffs (Segmentation Fault) beendet, wenn er sich selbst blockiert.
☐	☐	Ein Prozess im Zustand „erzeugt“ kann sich durch Aufrufen des Systemaufrufs `exec()` in „bereit“ versetzen.

d) Bereich: Traps und Interrupts

Richtig	Falsch	
☐	☐	Der Zugriff auf eine virtuelle Adresse kann zu einem Trap führen.
☐	☐	Normale Rechenoperationen können zu einem Trap führen.

e) Bereich: Scheduling

Richtig	Falsch	
☐	☐	Federgewichtige Prozesse (user-threads) können die Multiprozessorfähigkeit des Betriebssystems ausnutzen.
☐	☐	Scheduling-Ziele können in der Regel nicht alle gleichzeitig erreicht werden.
☐	☐	Kernel-Threads können Multiprozessoren nicht ausnutzen.
☐	☐	Aktives Warten kann auf einem Uniprozessorsystem vorteilhaft gegenüber passivem Warten sein.

f) Bitte beantworten Sie, wie häufig Sie die Lehrangebote jeweils wahrgenommen haben, egal ob online oder offline. Jede Antwort ist richtig. Es ist ein Kreuz pro Spalte zu setzen.
Optional: Gerne auch einige Worte zur Begründung.

Vorlesung	Tafelübung	Gruppenübung
○ 10-7	○ 5-4	○ 5-4
○ 6-4	○ 3-2	○ 3-2
○ 3-1	○ 1	○ 1
○ 0	○ 0	○ 0

Aufgabe 2: Programmieraufgabe – Filecheck (15 Punkte)

Sie haben ein altes Speichermedium von Ihnen gefunden, auf dem Sie einige kleine Skripte und Konfigurationsdateien finden wollen. Dieses enthält allerdings unzählige andere Dateien, Verzeichnisse und Links. Ihre Aufgabe ist es ein Program zu schreiben, welches reguläre Dateien mit einem Speicherbedarf von weniger als 1 KiB findet.

Das von Ihnen zu entwickelnde Programm `fc` soll das aktuelle Verzeichnis durchsuchen und für alle enthaltenen regulären Dateien den Namen und die Dateigröße in Byte auf der Standardausgabe ausgeben. Symbolischen Links soll gefolgt werden und versteckte Dateien werden ignoriert.

Sollte ein unerwarteter Fehler auftreten, so soll sich das Programm mit der Funktion `die(char *msg)` mit einer Fehlermeldung beenden.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

#define true 1
#define false 0

// Gegeben:

// Beenden mit Fehlerausgabe
void die(char *msg);

// Zu implementieren

// Eine Datei prüfen und behandeln
void handle_file(char *filename);

// Das aktuelle Verzeichnis durchlaufen
int main(int argc, char *argv[]);
```


closedir() closedir()

NAME
closedir – close a directory

SYNOPSIS
int closedir(DIR *_dirp_);

DESCRIPTION
The **closedir**() function closes the directory stream associated with *dirp*.

RETURN VALUE
The **closedir**() function returns 0 on success. On error, −1 is returned, and *errno* is set appropriately.

---

opendir(3) opendir(3)

NAME
opendir, fdopendir – open a directory

SYNOPSIS
DIR *opendir(const char *_name_);

DESCRIPTION
The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream.

RETURN VALUE
The **opendir**() function returns a pointer to the directory stream. On error, NULL is returned, and *errno* is set appropriately.

---

readdir(3) readdir(3)

NAME
readdir – read a directory

SYNOPSIS
#include <dirent.h>

struct dirent *readdir(DIR *_dirp_);

DESCRIPTION
The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory pointed to by *dirp*. It returns NULL on reaching the end of the directory or if an error occurred.

The *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* i-node number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file */
    char           d_name[256]; /* Null-terminated filename */ };
```

d_type This field indicates the file type:

DT_DIR	This is a directory.
DT_FIFO	This is a named pipe (FIFO).
DT_LNK	This is a symbolic link.
DT_REG	This is a regular file.
DT_SOCK	This is a UNIX domain socket.

RETURN VALUE
On success, **readdir**() returns a pointer to a *dirent* structure.

If the end of the directory is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set appropriately.

---

stat(2) stat(2)

NAME
stat, fstat – get file status

SYNOPSIS
int stat(const char *_path_, struct stat *_buf_);
int lstat(const char *_path_, struct stat *_buf_);

DESCRIPTION
These functions return information about a file.

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* i-node number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
};
```

RETURN VALUE
On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

```
void handle_file(char *filename) {
```

H:

M:


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

Aufgabe 3: Textfragen (19 Punkte)

a) Erklären Sie folgende Begriffe **stichwortartig** und setzen Sie diese in Beziehung zueinander: Prozess, Faden, Prozessor

b) Bei einem gegebenen Synchronisationsmuster mit mehreren wiederverwendbaren Betriebsmitteln sind alle Voraussetzungen für eine Verklemmung erfüllt. Wie könnte das Problem behoben werden. Skizzieren sie einen Lösungsvorschlag und begründen Sie stichwortartig welche Bedingung dadurch gebrochen wird.

c) Worin unterscheiden sich Online- und Offlinescheduling? Nennen Sie je einen Vorteil.

d) Klassifizieren Sie die folgenden Betriebsmittel

`Bildschirm`	
`Objekt in einer Queue`	
`Drucker`	

e) Ein Programm versucht eine illegale Instruktion durchzuführen. Beschreiben Sie in Stichpunkten den Ablauf. Ist der Ablauf unterdrückbar?