Aufgabe 1: (14 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) Gegeben ist folgendes Makro: 2 Punkte

```
#define SUM(a,b) a+b
```

Wie ist das Ergebnis des folgenden Ausdrucks?

```
SUM(2, 3) * SUM(2 - 1, 4)
```

- ☐ 9
- ☐ 11
- ☐ 15
- ☐ 25

b) Welche Aussagen zum Schlüsselwort **`volatile`** sind richtig? 2 Punkte

- ☐ Das Schlüsselwort **`volatile`** erlaubt dem Compiler bessere Optimierungen durchzuführen.
- ☐ Das Schlüsselwort **`volatile`** unterbindet alle Nebenläufigkeitsprobleme in der entsprechenden Datei.
- ☐ Das Schlüsselwort **`volatile`** sorgt dafür, dass die damit definierte Variable nur so kurz wie möglich in einem Register gehalten wird.
- ☐ Das Schlüsselwort **`volatile`** beschleunigt den Zugriff auf die damit definierte Variable.

c) Welche der folgenden Aussagen bzgl. der Interruptsteuerung ist richtig? 2 Punkte

- ☐ Während der Bearbeitung eines Interrupts nimmt der Prozessor keine weiteren Interrupts an.
- ☐ Wurde gerade ein Pegel-gesteuerter Interrupt ausgelöst, so muss erst ein Pegelwechsel der Interruptleitung stattfinden, bevor erneut ein Interrupt ausgelöst wird.
- ☐ Flankengesteuerten Interrupts können nicht blockiert werden, da sie völlig unvorhersehbar auftreten.
- ☐ Pegel-gesteuerte Interrupts werden beim Wechsel des Pegels ausgelöst.

d) Gegeben ist folgender Ausdruck: 2 Punkte

```
if ( (a = 5) || (b != 3) ) { c = 24; } else { c = 42; }
```

Welche Aussage ist richtig?

- ☐ Falls `a` den Wert `7` und `b` den Wert `5` enthält, wird `c` auf `42` gesetzt.
- ☐ Der Compiler meldet einen Fehler, weil dieser Ausdruck nicht zulässig ist.
- ☐ Falls `a` den Wert `5` und `b` den Wert `7` enthält, wird `c` auf `42` gesetzt.
- ☐ Der Wert von `a` hat keinen Einfluss auf das Ergebnis in `c`.

e) Was versteht man unter Polling? 2 Punkte

- ☐ Wenn ein Gerät so lange Interrupts auslöst, bis die Daten durch den Mikrocontroller abgeholt wurden.
- ☐ Wenn ein Programm regelmäßig eine Peripherie-Schnittstelle abfragt, ob Daten oder Zustandsänderungen vorliegen.
- ☐ Wenn ein Gerät durch Auslösen eines Interrupts Daten von einem Mikrocontroller anfordert.
- ☐ Wenn ein Programm zum Zugriff auf kritische Daten Interrupts sperrt.

f) Welche Aussage zur Speicherallokation ist richtig? 2 Punkte

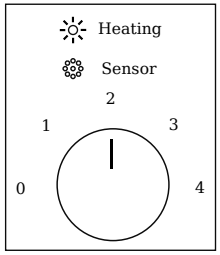
- ☐ Die dynamische Allokation von Speicher ist auf einem Mikrokontroller zu bevorzugen, da erst zur Laufzeit geprüft wird, ob der Speicher wirklich zur Verfügung steht.
- ☐ Die Speicheradresse von statisch allokierten Variablen kann sich zur Laufzeit ändern.
- ☐ `automatic`-Variablen werden im Heap allokiert.
- ☐ Die Verwendung von statisch allokierten Variablen erlaubt den Speicherbedarf bereits nach dem Binden abzuschätzen.

g) In Betriebssystemen wie UNIX oder Linux unterscheidet man die Begriffe Programm und Prozess. Welche Aussage ist richtig? 2 Punkte

- ☐ Ein Programm kann zu einem Zeitpunkt nur einmal gleichzeitig auf einem Rechner ausgeführt werden.
- ☐ Ein Programm kann mehrfach gestartet werden. Jede Programmausführung erfolgt in einem Prozess. Tritt bei einer dieser Programmausführungen ein Fehler auf (z.B. Segmentation Fault), so werden alle Prozesse, die das Programm gerade ausführen abgebrochen.
- ☐ Die Begriffe Programm und Prozess bedeuten eigentlich das Gleiche. Der Begriff Programm stammt aus dem Windows-Umfeld, während man in Linux-Systemen stattdessen meistens von Prozessen spricht.
- ☐ Ein Prozess ist ein Programm, das sich in Ausführung befindet. Es ist möglich, dass verschiedene Prozesse das gleiche Programm ausführen, jedoch dabei unterschiedliche Zugriffsrechte (z.B. auf Dateien) haben.

Aufgabe 2: Heizung (30 Punkte)

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



Implementieren Sie die Steuerung einer klimafreundlichen Heinzungsanlage. Zur Schonung des eigenen Geldbeutels und der Umwelt wird hierzu die Raumtemperatur mittels eines Sensors kontinuierlich gemessen, um so sicherzustellen, dass eine zuvor definierte Maximaltemperatur nie dauerhaft überschritten werden kann. Sofern die gewünschte Maximaltemperatur nicht erreicht wurde, kann die Heizleistung mit Hilfe eines Drehreglers feingesteuert werden. Eine leuchtende Kontrollleuchte soll dabei eine laufende Heizphase signalisieren. Falls die gemessene Maximaltemperatur überschritten wird oder der Regler den linken Anschlag erreicht, soll die Heizleistung auf null reduziert und die Kontrollleuchte ausgeschaltet werden.

Im Detail soll Ihr Programm wie folgt funktionieren:

- Initialisieren Sie die Hardware in der Funktion **`void`** `init(`**`void`**`)`. Treffen Sie hierbei keine Annahmen über den initialen Zustand der Hardware-Register.
- Der Eingang `PD2` (Interrupt 0) ist mit dem Raumtemperatursensor verbunden. Die externe Beschaltung stellt sicher, dass genau dann eine steigende Flanke auftritt, wenn die Maximaltemperatur erreicht wird und eine fallende Flanke dann, wenn diese den Wohlfühlbereich (Bereich unterhalb der Maximaltemperatur) wieder betritt. Stellen Sie sicher, dass dieses Event unbedingt vor dem Timer-Event abgearbeitet wird.
- Für die Zeittaktung soll ein 8-Bit Timer verwendet werden. Konfigurieren Sie diesen so, dass er alle $T = 1ms$ einen Interrupt auslöst.
- Der Regler soll alle T Zeiteinheiten ausgelesen werden. Messen Sie dafür zunächst durch Aufruf von **`int16_t`** `sb_adc_read(ADCDEV)` für `POTI` den Wert r des Drehreglers als vorzeichenlose 10 Bit Ganzzahl. Während des Aufrufs müssen die Interrupts gesperrt sein.
- Sofern die Maximaltemperatur nicht überschritten wurde, soll der gelesene Wert r mittels der Funktion **`uint16_t`** `convert(`**`int16_t`** `r)` in eine vorzeichenlose 16 Bit Ganzzahl umgewandelt werden. Benutzen Sie für die Umwandlung folgende Gleichung:

$$\texttt{convert(r)} = \begin{cases} 0 \quad\quad + \texttt{HEAT\_SCALE} \times (\texttt{r} \bmod 256) & \text{für } \texttt{r} \in \{0, \ldots, 255\} \\ 2^{12} - 1 + \texttt{HEAT\_SCALE} \times (\texttt{r} \bmod 256) & \text{für } \texttt{r} \in \{256, \ldots, 511\} \\ 2^{13} - 1 + \texttt{HEAT\_SCALE} \times (\texttt{r} \bmod 256) & \text{für } \texttt{r} \in \{512, \ldots, 767\} \\ 2^{14} - 1 + \texttt{HEAT\_SCALE} \times (\texttt{r} \bmod 256) & \text{für } \texttt{r} \in \{768, \ldots, 1022\} \\ 2^{16} - 1 & \text{für } \texttt{r} \in \{1023\} \end{cases} \quad (1)$$

- Um eine unübersichtliche **`if-else`**-Kaskade zu vermeiden, sollen die Grundwerte (0, $2^{12} - 1$, $\ldots$) innerhalb des Arrays **`uint16_t`** `heating_interval[5]` gespeichert werden. Die Heizstufe (0, 1, 2, 3, 4), welche sich aus dem Intervall von r ergibt, soll als Index dienen.
- Der konvertierte Wert soll schließlich an Funktion **`void`** `heat(`**`uint16_t`** `value)` übergeben werden. Diese kapselt die eigentliche Steuerung der Heizleistung. Beachten sie zusätzlich, dass die Kontrollleuchte immer nur genau dann aktiv sein soll, wenn die Maximaltemperatur noch nicht erreicht wurde und der konvertierte Wert größer null ist.
- Stellen Sie sicher, dass sich der Mikrocontroller möglichst oft im Schlafmodus befindet.

Information über die Hardware

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

Temperatursensor: Interruptleitung an **`PORTD`**, Pin 2

- Steigende Flanke: Zulässige Maximaltemperatur wurde überschritten, Heizung wird ausgeschaltet.
- Fallende Flanke: Raumtemperatur hat Wohlfühlbereich erreicht, Feinsteuerung mittels Temperaturregler beginnt.
- Pin als Eingang konfigurieren: Entsprechendes Bit im **`DDRD`**-Register auf 0
- Internen Pull-Up-Widerstand deaktivieren: Entsprechendes Bit im **`PORTD`**-Register auf 0
- Externe Interruptquelle **`INT0`**, ISR-Vektor-Makro: **`INT0_vect`**
- Aktivieren/Deaktivieren der Interruptquelle erfolgt durch Setzen/Löschen des **`INT0`**-Bits im Register **`EIMSK`**

Konfiguration der externen Interruptquelle **`INT0`** (Bits im Register **`EICRA`**)

Interrupt 0		Beschreibung
ISC01	ISC00	
0	0	Interrupt bei low Pegel
0	1	Interrupt bei beliebiger Flanke
1	0	Interrupt bei fallender Flanke
1	1	Interrupt bei steigender Flanke

Kontrollleuchte für Aufheizen: Ausgang an **`PORTB`**, Pin 0

- Gibt an, ob gerade aktiv geheizt wird. Für ausgeschaltene Heizung auf LOW setzen, sonst auf HIGH
- Pin als Ausgang konfigurieren: Entsprechendes Bit im **`DDRB`**-Register auf 1
- Initialer Zustant zunächst 0, entsprechendes Bit im **`PORTB`**-Register auf 0

Zeitgeber (8-bit): **`TIMER0`**

- Es soll die Überlaufunterbrechung verwendet werden (ISR-Vektor-Makro: **`TIMER0_OVF_vect`**)
- Der ressourcenschonendste Vorteiler (*prescaler*) ist 64, wodurch es bei dem 16 MHz CPU-Takt (hinreichend genau) alle $1ms$ zum Überlauf des 8-bit-Zählers **`TCNT0`** kommt.
- Aktivieren/Deaktivieren der Interruptquelle erfolgt durch Setzen/Löschen des **`TOIE0`**-Bits im Register **`TIMSK0`**

Konfiguration der Frequenz des Zeitgebers **`TIMER0`** (Bits im Register **`TCCR0B`**)

CS02	CS01	CS00	Beschreibung
0	0	0	Timer aus
0	0	1	CPU-Takt
0	1	0	CPU-Takt / 8
0	1	1	CPU-Takt / 64
1	0	0	CPU-Takt / 256
1	0	1	CPU-Takt / 1024
1	1	0	Ext. Takt (fallende Flanke)
1	1	1	Ext. Takt (steigende Flanke)

Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.

```
#include <avr/interrupt.h>
#include <avr/io.h>
#include <avr/sleep.h>
#include <stdint.h>
#include <adc.h>

extern int16_t sb_adc_read(ADCDEV dev);
extern void heat(uint16_t value);
#define HEAT_SCALE 8

// Funktionsdeklarationen, globale Variablen, etc.
```

```
// Unterbrechungsbehandlungsfunktionen
```

```
// Ende Unterbrechungsbehandlungsfunktionen
```

D:


```
// Funktion main

  // Initialisierung und lokale Variablen
```

```

  // Hauptschleife
```

```

    // Ereignisse verarbeiten
```

```
  // Ende main
```

M:


```
// Konvertierungsfunktion
```

```
// Ende Konvertierungsfunktion
```

C:

```
// Initialisierungsfunktion
```

```
// Ende Initialisierungsfunktion
```

I:

Aufgabe 3: strip (21.5 Punkte)

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

Schreiben Sie das Programm `strip`, welches die Eingabe über `stdin` gefiltert auf `stdout` ausgibt. Als Filter können Zeichenketten, sogenannte Masken, als Parameter beim Programmaufruf übergeben werden. Beginnt eine Zeile mit einer der Masken, wird die Zeile verworfen und nicht ausgeben. So lassen sich beispielsweise Kommentare aus Datensätzen entfernen.

Inhalt der Datei test.txt ausgeben:

```
$> cat test.txt
no comment
//comment
comment sign // in line
```

`strip` starten und test.txt nach `stdin` weiterleiten:

```
$> ./strip -m "//" < test.txt
no comment
comment sign // in line
```

```
$> ./strip -m "//" -i < test.txt
//comment
```

Das Programm soll im Detail wie folgt funktionieren:

- Implementieren Sie zuerst die Hilfsfunktion `int filter(const char **masks, unsigned num_masks, const char *line)`, welche wie folgt funktionieren soll:
  - Für jeden Aufruf von `filter()` soll überprüft werden, ob die übergebene Zeile `line` mit einer der Masken in `masks` beginnt (siehe `strncmp(2)`).
  - `num_masks` gibt an, wie viele Masken im Array `masks` enthalten sind.
  - Gab es mit keiner Maske eine Übereinstimmung, soll der Wert `0` zurückgegeben werden, andernfalls der Wert `1`.
- In der Funktion `main()` soll `strip` zunächst die Kommandozeilenparameter auswerten:

  `-m` Das nächste übergebene Argument stellt eine Filtermaske dar.

  `-i` Die Ausgabe soll invertiert werden.
- Im Anschluss soll das Programm zeilenweise von `stdin` lesen.
- Die eingelesene Zeile soll auf `stdout` ausgegeben werden, sofern die Hilfsfunktion `filter()` den Wert `0` zurückgibt. Andernfalls wird die Zeile verworfen.
- Wurde die Option `-i` als Kommandozeilenparameter übergeben, soll eine Zeile stattdessen ausgegeben werden, wenn die Hilfsfunktion `filter()` den Wert `1` zurückgibt.
- Können keine Zeichen mehr gelesen werden, soll sich `strip` erfolgreich beenden.

Hinweise:

- Sie dürfen davon ausgehen, dass die einzelnen Zeilen nicht länger als `1024` Zeichen (siehe `MAX_LINE_LENGTH`) sind.
- Sie dürfen davon ausgehen, dass maximal `32` Masken (siehe `MAX_MASKS`) als Kommandozeilenparameter übergeben werden.

Achten Sie auf eine korrekte Fehlerbehandlung der verwendeten Funktionen. Fehlermeldungen sollen generell auf `stderr` erfolgen. Zur kompakten Fehlerbehandlung können die vorgegebenen Funktionen `die()` (`errno` gesetzt) und `err()` (`errno` nicht gesetzt) genutzt werden.

Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Maximale Zeilenlänge
#define MAX_LINE_LENGTH 1024
// Maximale Anzahl per "-m" uebergebene Masken
#define MAX_MASKS       32

// Ausgabe von Systemfehlermeldungen
static void die(const char *str) {
  perror(str);
  exit(EXIT_FAILURE);
}

// Ausgabe von sonstigen Fehlermeldungen
static void err(const char *msg) {
  fprintf(stderr, "%s\n", msg);
  exit(EXIT_FAILURE);
}
```

```
// Funktion filter
```

F:


```
// Funktion main


  //Argumente parsen
```

```
// Zeilen einlesen, filtern, ausgeben
```

M:

Aufgabe 4: Bitoperationen (10 Punkte)

a) Kreuzen Sie an welche LEDs nach Aufruf von `func(0xcc)` leuchten.

2 Punkte

```
static void func(uint8_t leds) {
    sb_led_setMask(leds & 0x4c);
}
```

☐ `RED0`	(Bit 0)	*Notizen:*
☐ `YELLOW0`	(Bit 1)	
☐ `GREEN0`	(Bit 2)	
☐ `BLUE0`	(Bit 3)	
☐ `RED1`	(Bit 4)	
☐ `YELLOW1`	(Bit 5)	
☐ `GREEN1`	(Bit 6)	
☐ `BLUE1`	(Bit 7)	

b) Kreuzen Sie an welche LEDs nach Aufruf von `func(4)` leuchten.

2 Punkte

```
static void func(uint8_t i) {
    if(i > 0) {
        sb_led_setMask((1 << (i-1) | (1 << i));
    }
}
```

☐ `RED0`	(Bit 0)	*Notizen:*
☐ `YELLOW0`	(Bit 1)	
☐ `GREEN0`	(Bit 2)	
☐ `BLUE0`	(Bit 3)	
☐ `RED1`	(Bit 4)	
☐ `YELLOW1`	(Bit 5)	
☐ `GREEN1`	(Bit 6)	
☐ `BLUE1`	(Bit 7)	

c) Kreuzen Sie an welche LEDs nach Aufruf von `func(0xfe)` leuchten.

2 Punkte

```
static void func(uint8_t leds) {
    leds |= ~((1 << 6) | 1);
    sb_led_setMask(leds);
}
```

☐ `RED0`	(Bit 0)	*Notizen:*
☐ `YELLOW0`	(Bit 1)	
☐ `GREEN0`	(Bit 2)	
☐ `BLUE0`	(Bit 3)	
☐ `RED1`	(Bit 4)	
☐ `YELLOW1`	(Bit 5)	
☐ `GREEN1`	(Bit 6)	
☐ `BLUE1`	(Bit 7)	

d) Beschreiben Sie welche LEDs nach Aufruf von `func()` leuchten. Sie müssen keine konkreten LEDs nennen. Beispielantworten: *die obersten vier LEDs*, *alle LEDs für die eine 1 in* `leds` *gesetzt ist*, *die unterste LED*. (2 Punkte)

```
static void func(uint8_t leds) {
    sb_led_setMask(leds ^ 0xff);
}
```

e) Beschreiben Sie welche LEDs beim Aufruf `func()` je Iteration leuchten. Sie müssen keine konkreten LEDs nennen. Beispielantworten: *die obersten vier LEDs*, *alle LEDs für die eine 1 in* `leds` *gesetzt ist*, *die unterste LED*. (2 Punkte)

```
static void func(void) {
    for(uint8_t i = 0; i < 4; i++) {
        sb_led_setMask(0xfe + i);
    }
}
```

Iteration 1 (i=0):

Iteration 2 (i=1):

Iteration 3 (i=2):

Iteration 4 (i=3):

Aufgabe 5: Nebenläufigkeit (8 Punkte)

Sie dürfen diese Seite zur besseren Übersicht heraustrennen!

Das nachfolgende Codebeispiel für einen 8-Bit-AVR-Mikrocontroller überprüft, ob die modulglobale Variable `counter` ein anliegendes Event signalisiert. Wenn dies der Fall ist, wird darauf reagiert, ansonsten wird mit der Hauptschleife fortgefahren. Der Wert von `counter` wird asynchron durch die Unterbrechungsbehandlung von `INT0` erhöht.

Die Implementierung dieser Funktionalität beinhaltet ein Nebenläufigkeitsproblem.

```
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdint.h>

static volatile uint8_t counter = 0;

ISR(INT0_vect) {
    counter++;
}

void main(void) {
    while(1) {
        if(counter > 0) {
            counter--;
            // Verarbeite Event...
        }
    }
}
```

Die beiden folgenden Assemblerausschnitte zeigen Abschnitte der `main()`-Funktion und der Unterbrechungsbehandlungsfunktion für `INT0`. In den Assemblerausschnitten können Sie den Kommentaren entnehmen, welche C-Anweisung die folgenden Assemblerinstruktionen repräsentieren.

Hauptprogramm

```
; counter--;
H1: lds   r24, counter
H2: subi  r24, 0x01
H3: sts   counter, r24
```

Interruptbehandlung `INT0`

```
; counter++;
I1: lds   r25, counter
I2: addi  r25, 0x01
I3: sts   counter, r25
```

a) Benennen Sie das Nebenläufigkeitsproblem. (1 Punkt)

b) Demonstrieren Sie einen konkreten Programmablauf, bei dem das Nebenläufigkeitsproblem auftritt. (4 Punkte)

Tragen Sie dafür die relevanten Speicher- und Registerinhalte **nach** der Ausführung jeder Assemblerinstruktion in die nachfolgende Tabelle ein. Gehen Sie davon aus, dass die Variable `counter` initial den Wert `0x0f` hat und treffen Sie keine Annahmen über andere Variablen- oder Speicherinhalte.
Kennzeichnen Sie auch in der Tabelle, wann gegebenfalls ein Interrupt auftritt.

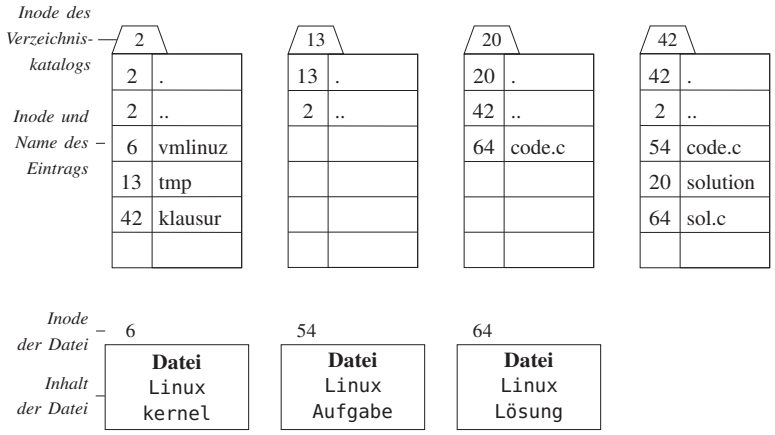
Zeile	**counter**	**r24**	**r25**
–	`0x0f`	–	–

c) Mit welchem Mechanismus ließe sich das Nebenläufigkeitsproblem lösen? (1 Punkt)

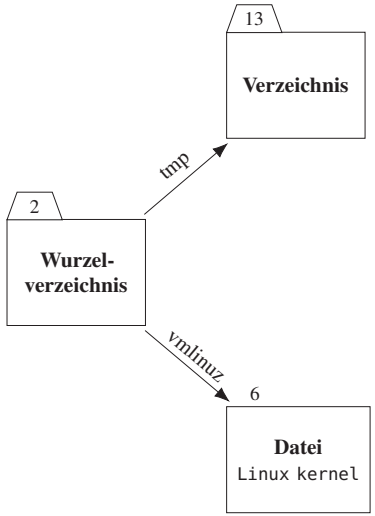
d) Ist die Verwendung des Schlüsselworts `volatile` für die Deklaration der Variablen `value` nötig? Begründen Sie kurz. (2 Punkte)

Aufgabe 6: Dateisystem (6.5 Punkte)

Ein Dateisystem ermöglicht das strukturierte Ablegen von Daten. Nachfolgend ist ein beispielhafter und vereinfachter Ausschnitt der hierfür in Linux benötigten Verwaltungsinformationen abgebildet.



a) Vervollständigen Sie den dazugehörigen Verzeichnisbaum. Orientieren Sie sich dabei an dem bereits vorgegebenen Schema: Verzeichnisse und Dateien werden mit einem beschrifteten Rechteck gekennzeichnet, ein Verweis mit einem beschrifteten Pfeil. Das Einzeichnen der `.` und `..` Verweise ist allerdings **nicht** erforderlich. (4.5 Punkte)



b) Erklären Sie den Unterschied zwischen **Verweisen** (Hard Links) und **symbolischen Verweisen** (Symbolic Links) im Hinblick auf nichtexistente Dateien. (2 Punkte)