Aufgabe 1: Ankreuzfragen (30 Punkte)

1) Einfachauswahlfragen (22 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) Welche Aussage zum Thema „Aktives Warten“ ist richtig? 2 Punkte

- ○ Aktives Warten auf andere Prozesse bei Scheduling-Strategien ohne Verdrängung auf einem Monoprozessorsystem kann zu Verklemmungen (*Deadlocks*) führen.
- ○ Aktives Warten vergeudet gegenüber passivem Warten immer CPU-Zeit.
- ○ Auf Mehrprozessorsystemen ist aktives Warten unproblematisch und deshalb dem passiven Warten immer vorzuziehen.
- ○ Bei verdrängenden Scheduling-Strategien verzögert aktives Warten nur den betroffenen Prozess, behindert oder verzögert aber nicht andere.

b) Welche Aussage zum Thema Synchronisation ist richtig? 2 Punkte

- ○ Durch den Einsatz von Semaphoren kann ein wechselseitiger Ausschluss erzielt werden.
- ○ Die V-Operation kann auf einem Semaphor nur von dem Thread aufgerufen werden, der zuvor auch die P-Operation aufgerufen hat.
- ○ Ein Semaphor kann ausschließlich für mehrseitige Synchronisation (*multilateral synchronisation*) verwendet werden.
- ○ Einseitige Synchronisation (*unilateral synchronisation*) erfordert immer Betriebssystem-Unterstützung.

c) Was versteht man unter Virtuellem Speicher? 2 Punkte

- ○ Adressierbarer Speicher in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.
- ○ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.
- ○ Unter einem virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.
- ○ Speicher, der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber möglicherweise größer als der verfügbare physikalische Hauptspeicher ist.

d) Gegeben seien die folgenden Präprozessor-Makros: 2 Punkte

```
#define SUB(a, b) a - b
#define MUL(a, b) a * b
```

Was ist das Ergebnis des folgenden Ausdrucks? `4 * MUL ( SUB(3,5), 2)`

- ○ 16
- ○ -16
- ○ -2
- ○ 2

e) Welche Seitennummer (*page number*) und welcher Versatz (*offset*) gehören bei einstufiger Seitennummerierung und einer Seitengröße von 2048 Bytes zu folgender logischer Adresse: 0xba1d 2 Punkte

- ○ Seitennummer 0x17, Versatz 0x21d
- ○ Seitennummer 0xba, Versatz 0x1d
- ○ Seitennummer 0x2e, Versatz 0x21d
- ○ Seitennummer 0xb, Versatz 0xa1d

f) Welche Aussage zu Interrupts ist richtig? 2 Punkte

- ○ Mit einer Signalleitung wird dem Prozessor eine Unterbrechung angezeigt. Der Prozessor sichert den aktuellen Zustand bestimmter Register, insbesondere des Programmzählers, und springt eine vordefinierte Behandlungsfunktion an.
- ○ Eine Signalleitung teilt dem Prozessor mit, dass er den aktuellen Prozess anhalten und auf das Ende der Unterbrechung warten soll.
- ○ Bei der mehrfachen Ausführung eines unveränderten Programms mit gleicher Eingabe treten Interrupts immer an den gleichen Stellen auf.
- ○ Durch eine Signalleitung wird der Prozessor veranlasst, die gerade bearbeitete Maschineninstruktion abzubrechen.

g) Welche Aussage zum Thema Systemaufrufe ist richtig? 2 Punkte

- ○ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.
- ○ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht selbst ausführen dürfte.
- ○ Nach der Bearbeitung eines beliebigen Systemaufrufes ist es für das Betriebssystem nicht mehr möglich, zu dem Programm zu wechseln, welches den Systemaufruf abgesetzt hat.
- ○ Durch einen Systemaufruf wechselt das Betriebssystem von der Systemebene auf die Benutzerebene, um unprivilegierte Operationen ausführen zu können.

h) Welche Aussage zu prioritätsbasierten Scheduling-Verfahren ist richtig?

2 Punkte

- ○ Kurze Prozesse werden bei Multilevel Feedback Queues generell bevorzugt.
- ○ Prioritätsumkehr (*priority inversion*) kann nur mit dynamischen Prioritäten auftreten.
- ○ Bei Multilevel Feedback Queues werden Prozesse, die zu lange laufen, abgebrochen (*anti-aging*), um sicherzustellen, dass andere Prozesse nicht verhungern (*starvation*).
- ○ Prioritätsumkehr (*priority inversion*) meint, dass ein Prozess mit niedriger Priorität abgebrochen wird (*umkehrt*), damit ein Prozess mit höherer Priorität laufen kann.

i) Welche der folgenden Aussagen über UNIX-Dateisysteme ist richtig?

2 Punkte

- ○ Auf das Wurzelverzeichnis (root directory, „/“) darf immer nur genau ein *hard link* verweisen.
- ○ Ein *hard link* kann auf eine Datei innerhalb eines anderen Dateisystems verweisen.
- ○ Ein *symbolic link* erhält die Inode Nummer der Datei auf die der Link verweist.
- ○ Wird der letzte *hard link* auf eine Datei entfernt, so wird auch die Datei selbst gelöscht.

j) Beim Einsatz von RAID-Systemen kann durch zusätzliche Festplatten Fehlertoleranz erzielt werden. Welche Aussage dazu ist richtig?

2 Punkte

- ○ Bei RAID 4 werden alle im Verbund beteiligten Festplatten gleichmäßig beansprucht.
- ○ Bei RAID 0 führt der Ausfall einer der beteiligten Festplatten nicht zu Datenverlust.
- ○ Bei RAID 1 werden die Datenblöcke über mehrere Festplatten verteilt und repliziert gespeichert.
- ○ Bei RAID 5 Systemen sind mindestens 5 Festplatten nötig.

k) Welche Aussage zu Prozessen und Threads ist richtig?

2 Punkte

- ○ 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.
- ○ Der Aufruf von `fork()` gibt im Elternprozess die Prozess-ID des Kindprozesses zurück, im Kindprozess hingegen den Wert 0.
- ○ Mittels `fork()` erzeugte Kindprozesse können in einem Multiprozessor-System nur auf dem Prozessor ausgeführt werden, auf dem auch der Elternprozess ausgeführt wird.

2) Mehrfachauswahlfragen (8 Punkte)

Bei den Mehrfachauswahlfragen in dieser Aufgabe sind jeweils m Aussagen angegeben, davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie alle richtigen Aussagen an.
Jede korrekte Antwort in einer Teilaufgabe gibt einen Punkt, jede falsche Antwort einen 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) Gegeben sei folgendes Programm:

4 Punkte

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define PI 3.1415
5
6  extern int x;
7
8  int main(int argc, char *argv[]) {
9      static int a;
10     int b = PI;
11
12     x = a + b;
13
14     printf("%f\n", b);
15
16     return EXIT_SUCCESS;
17 }
```

Welche der folgenden Aussagen bzgl. dieses Programms sind korrekt?

- ☐ Der Aufruf von `printf` in Zeile 14 gibt den Wert `3.1415` auf `stdout` aus.
- ☐ Beim Linken des Programms kann ein Fehler auftreten.
- ☐ `argv` ist ein Array aus Zeigern, die jeweils auf ein Array aus `chars` zeigen.
- ☐ Die Variable `a` ist uninitialisiert und enthält daher einen zufälligen Wert.
- ☐ Die globale Variable `PI` enthält den Wert `3.1415`.
- ☐ Beim Überschreiben der Variable `x` in Zeile 12 tritt ein Fehler auf, weil externe Variablen nicht überschrieben werden dürfen.
- ☐ An Index `0` des `argv`-Arrays liegt ein Zeiger auf den Programmnamen oder -pfad.
- ☐ Der Inhalt der Datei *stdlib.h* wird vor dem Übersetzen an die Stelle der entsprechenden `include`-Anweisung einkopiert.

b) Welche der folgenden Aussagen zu UNIX-Dateisystemen sind richtig?

4 Punkte

- ☐ Im Wurzelverzeichnis '/' existiert kein Eintrag '..'.
- ☐ Innerhalb eines Verzeichnisses können mehrere Verweise auf dieselbe Inode existieren, sofern diese unterschiedliche Namen haben.
- ☐ In den Attributen einer Inode wird ein Referenzzähler mit der Anzahl der *symbolic links*, die auf die Inode verweisen, gespeichert.
- ☐ In den Attributen einer Inode werden Dateityp, Eigentümer und Dateigröße gespeichert.
- ☐ Ein Pfadname, der nicht mit einem '/'-Zeichen beginnt, wird relativ zum Home-Verzeichnis des Benutzers interpretiert.
- ☐ Beim Anlegen einer Datei wird die maximale Größe festgelegt. Wird sie bei einer Schreiboperation überschritten, wird ein Fehler gemeldet.
- ☐ Im Wurzelverzeichnis '/' verweist der Eintrag '..' wieder auf das Wurzelverzeichnis.
- ☐ In jedem Verzeichnis gibt es einen Eintrag, der auf das Verzeichnis selbst verweist.

Aufgabe 2: `saver` (60 Punkte)

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

Schreiben Sie ein POSIX-1.2008 und C11-konformes Programm `saver`, welches zeilenweise Rechnernamen aus einer per Befehlszeilenargument übergebenen Datei einliest, überprüft ob die Rechner aktuell in Benutzung sind und inaktive Rechner herunterfährt um Energie zu sparen.

Beispielhafter Aufruf von `saver`:

```
hofmeier@tardis:~$ ./saver hostnames.txt
```

Das Programm soll folgendermaßen strukturiert sein:

- Funktion `main()`:
  Prüft zunächst die Befehlszeilenargumente und initialisiert ggf. benötigte Datenstrukturen. Zum Auslesen der per Befehlszeilenargument übergebenen Datei wird die Funktion `parseFile()` (siehe unten) aufgerufen. Im Anschluss wird für jeden Rechner durch Aufruf des Programms `check_idle` geprüft, ob dieser aktuell in Benutzung ist. Die Überprüfung per `check_idle` soll durch Aufruf der Funktion `run()` (siehe unten) parallel ausgeführt werden.
  Sobald die Überprüfung **aller** Rechner abgeschlossen ist, werden **inaktive** Rechner durch das Programm `shutdown_remote` parallel heruntergefahren. Zur parallelen Ausführung von `shutdown_remote` soll ebenfalls die Funktion `run()` genutzt werden. Ein Rechner gilt als inaktiv, wenn der entsprechende `check_idle` Prozess mit `EXIT_SUCCESS` terminiert ist. Sollte ein Prozess ohne Exitcode terminieren, wird der entsprechende Rechner nicht heruntergefahren.
  Das Programm wartet abschließend darauf, dass alle gestarteten Prozesse beendet wurden und gibt dann alle angeforderten Ressourcen (inkl. der in `parseFile()` angelegten Liste) frei.
- Funktion `void parseFile(char* filename)`:
  Die Funktion liest die als Parameter übergebene Datei zeilenweise ein. Die Datei enthält pro Zeile einen Rechnernamen. Leere Zeilen und Zeilen, die länger als `MAX_LINE` sind, sollen ignoriert werden. Zur weiteren Verwaltung werden alle eingelesenen Rechnernamen in eine modulglobale, einfach verkettete Liste bestehend aus `struct host`-Einträge eingetragen. Jeder Listeneintrag soll die folgenden Informationen enthalten können:
  - Rechnername
  - PID des bearbeitenden Prozesses
  - Statusinformationen von `wait()`
  - ggf. benötigte Datenstruktur(en) für die Listenimplementierung
- Funktion `void waitProcess(void)`:
  Wartet per `wait()` passiv auf **einen beliebigen** der per `run()` gestarteten Prozesse. Die Funktion speichert die von `wait()` gelieferten Statusinformationen des terminierten Prozesses im entsprechenden `struct host`-Eintrag.
- Funktion `void run(char *bin, struct host *arg)`:
  Erzeugt einen neuen Kindprozess und führt die Anwendung `bin` mithilfe einer Funktion der `exec()`-Familie aus. `bin` erhält als Befehlszeilenargument den Rechnernamen aus `arg`. Der Elternprozess speichert die Prozess-ID des erzeugten Kindes in `arg` und kehrt ohne zu warten zurück. Achten Sie auch im Kindprozess auf korrekte und vollständige Fehlerbehandlung.

Hinweise:

- `check_idle` und `shutdown_remote` bekommen jeweils **einen** Rechnernamen als Befehlszeilenargument. Sie dürfen davon ausgehen, dass beide Programme in `PATH` enthalten sind.
- Achten Sie auf korrekte und vollständige Fehlerbehandlung.

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define MAX_LINE 4096

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

static void usage(void) {
  fprintf(stderr, "Usage: saver <hostname file>\n");
  exit(EXIT_FAILURE);
}
```

```
struct host {
  pid_t pid; // PID des bearbeitenden Prozesses
  // Eigene Mitglieder
};
// Makros, Funktionsdeklarationen, globale Variablen
```



```
// Funktion main
  // Befehlszeilenargumente prüfen
  // Datei parsen
  // Rechner auf Inaktivität prüfen
```

```
  // Inaktive Rechner herunterfahren
```

```
  // Aufräumen und Beenden
```

```
// Ende Funktion main
```

M:


```
// Funktion parseFile
```

```
  // Zeilenweises Auslesen der Datei
```

```
  // Fehlerbehandlung + Aufräumen
```

```
// Ende Funktion parseFile
```

P:


```
// Funktion waitProcess
```

```
// Ende Funktion waitProcess
```

W:

```
// Funktion run
```

```
// Ende Funktion run
```

R:

Schreiben Sie ein Makefile, welches die Targets `all` und `clean` unterstützt. Ebenfalls soll ein Target `saver` unterstützt werden, welches das Programm `saver` baut. Greifen Sie dabei stets auf Zwischenprodukte (z.B. `saver.o`) zurück.

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `saver` löschen.

Definieren und nutzen Sie dabei die Variablen `CC` und `CFLAGS` konventionskonform. Achten Sie darauf, dass das Makefile ohne eingebaute Variablen und Regeln (Aufruf von `make -Rr`) funktioniert!

Mk:

Aufgabe 3: Synchronisation (12 Punkte)

1) Erläutern Sie das Konzept Semaphor. Welche Operationen sind auf Semaphoren definiert und was tun diese Operationen? (5 Punkte)

2) Was versteht man unter einer Verklemmung (*Deadlock*)? (2 Punkte)

3) Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie das folgende Szenario mit Hilfe eines Semaphors korrekt synchronisiert werden kann: Zehn Threads führen in der Funktion `calcValue()` parallele Berechnungen durch und addieren die berechneten Werte auf den Wert einer globalen Variable `accu` auf. Zu jedem Zeitpunkt müssen so viele Threads wie möglich die Funktion `calcValue()` parallel ausführen können. Ihnen stehen dabei folgende Semaphor-Funktionen zur Verfügung: (5 Punkte)

- `SEM * semCreate(int);`
- `void P(SEM *);`
- `void V(SEM *);`

Beachten Sie, dass für eine korrekte Lösung nicht unbedingt in allen Zeilen eine Operation vorgenommen werden muss. **Kennzeichnen Sie durch '/', wenn Ihre Implementierung in einer freien Zeile keine Operation benötigt.**

Hauptthread:

```
static int accu;
static SEM *s;
int main(void){
-------------------------------------
  for(int i = 0; i < 9; ++i) {
    startWorkerThread(threadFunc);
  }
-------------------------------------
  while(1) {
-------------------------------------
    int x = calcValue();
-------------------------------------
    accu += x;
-------------------------------------
  }
}
```

Arbeiterthread:

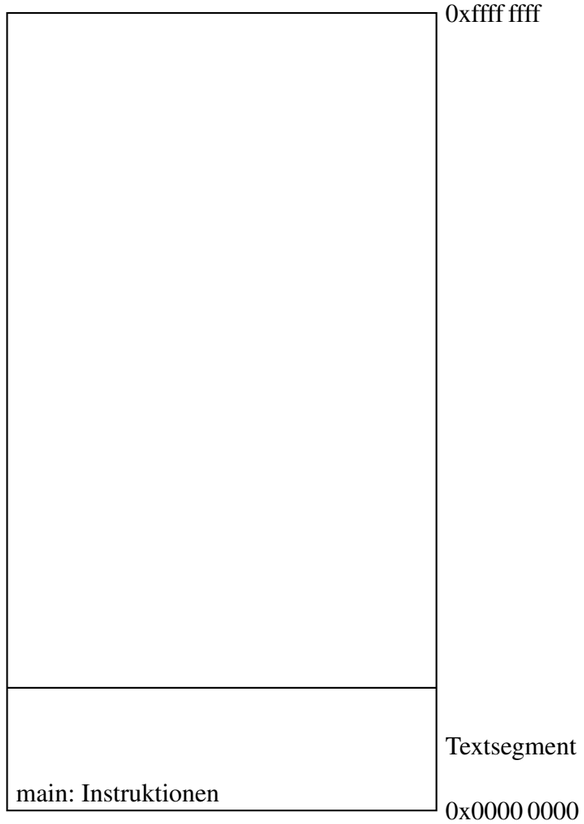
```
void threadFunc(void) {
-------------------------------------
  while(1) {
-------------------------------------
    int x = calcValue();
-------------------------------------
    accu += x;
-------------------------------------
  }
-------------------------------------
}
```

Aufgabe 4: Adressräume & Speicherverwaltung (18 Punkte)

1) Gegeben sei das nachfolgende Programm. Skizzieren Sie den Aufbau des logischen Adressraums eines Prozesses, der dieses Programm ausführt. Tragen Sie die Segmente und deren Namen (analog zum schon vorgegebenen Textsegment) in unten stehender Zeichnung ein. Unterscheiden Sie hierbei die Bereiche zur Speicherung von initialisierten und nicht initialisierten Variablen. Zeichnen Sie für jede Variable ein, wo diese ungefähr im logischen Adressraum zu finden sein wird und welchen Wert sie enthält. Illustrieren Sie im Falle von Zeigervariablen mittels Pfeil, auf welches Datum die Variable jeweils zeigt.

Vermerken Sie zudem, in welche Richtung Segmente variabler Größe wachsen. Gehen Sie hierbei von einem x86-System aus. (8 Punkte)

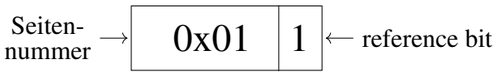
```
static int a = 2;
static int b = 0;

const char *s = "Hello World\n";
int t = 0;

int main(void) {
    int g = 7;
    static int h = 42;
    void *arr = malloc(7);
    int (*func)(void) = main;
}
```



2) Eine in der Praxis gut einsetzbare Strategie ist Second Chance: CLOCK. In dieser Aufgabe soll CLOCK als (Prozess-)lokale Seitenersetzungsstrategie eingesetzt werden. (10 Punkte)

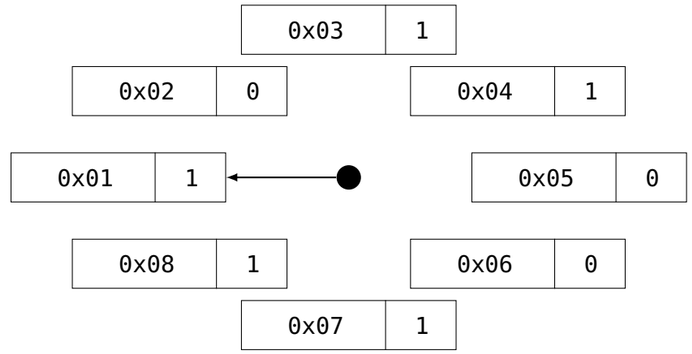
Im Folgenden sind die für die Seitenverwaltung erforderlichen Daten der aktuell anwesenden Seiten eines Prozesses dargestellt. Nehmen Sie eine Seitengröße von 4096 Bytes an (ergibt 12 Bit Offset). Gehen Sie davon aus, dass alle Seiten les- und schreibbar sind und alle zugegriffenen Adressen gültig sind.

Hinweise: Der CLOCK-Zeiger zeigt jeweils auf den Eintrag, der bei der nächsten Suche nach einem freien Seitenrahmen (*page frame*) als erstes überprüft wird. Für neu eingelagerte Seiten wird das *reference bit* mit `1` initialisiert.



Beachten Sie, dass **Zustand B ausgehend von Zustand A** bestimmt werden soll und alle Zugriffe in der angegebenen Reihenfolge geschehen (erst `1)` dann `2)` usw.).

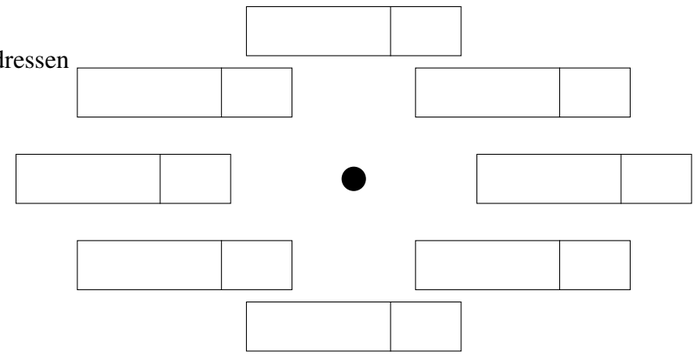
Ausgangszustand:



Zustand A:
<u>nach Zugriff</u> auf folgende Adressen

1) Lesen von `0x06` `120`

2) Schreiben nach `0x0b` `42c`

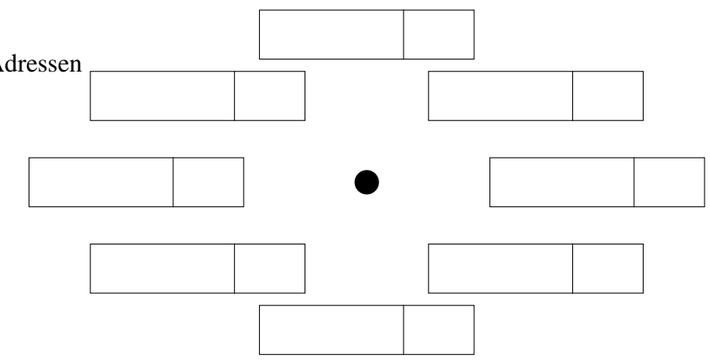


Zustand B:
<u>nach Zugriff</u> auf folgende Adressen

3) Schreiben nach `0x02` `120`

4) Lesen von `0x01` `f0c`

5) Schreiben nach `0x0a` `00a`



Ersatzgrafik für Teilaufgabe 4.2.
Sie dürfen diese Grafik nutzen, falls Sie sich beim Ausfüllen der Grafik in 4.2. verzeichnet haben.
Markieren Sie eindeutig, welche der Grafiken zur Bewertung herangezogen werden soll!