Aufgabe 1.1: Einfachauswahl-Fragen (22 Punkte)

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

Wollen Sie eine Multiple-Choice-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 über das aktuelle Arbeitsverzeichnis (Current Working Directory) trifft zu? 2 Punkte

- ❏ Besitzt ein UNIX-Prozess kein Current Working Directory, so beendet sich der Prozess mit einem Segmentation Fault.
- ❏ Pfadnamen, die nicht mit dem Zeichen '/' beginnen, werden relativ zum aktuellen Arbeitsverzeichnis interpretiert.
- ❏ Jedem UNIX-Benutzer ist zu jeder Zeit ein aktuelles Verzeichnis zugeordnet.
- ❏ Mit dem Systemaufruf `chdir()` kann das aktuelle Arbeitsverzeichnis aller Kindprozesse durch den Vaterprozess verändert werden.

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

- ❏ 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 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 deterministisch auftritt und damit eine Wiederaufnahme des Programms sofort wieder den Trap verursachen würde.
- ❏ 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.

c) Welche Aussage über `exec()` ist richtig? 2 Punkte

- ❏ Das im aktuellen Prozess laufende Programm wird durch das angegebene Programm ersetzt.
- ❏ Dem Vater-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.
- ❏ `exec()` erzeugt einen neuen Kind-Prozess und startet darin das angegebene Programm.
- ❏ Der an `exec()` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.

d) In einem Unix/Linux-Dateisystem gibt es symbolische Namen/Verweise (Symbolic Links) und feste Links (Hard Links) auf Dateien. Welche Aussage ist richtig. 2 Punkte

- ❏ Ein Symbolic Link kann nicht auf Dateien anderer Dateisysteme verweisen.
- ❏ Ein Hard Link kann nur auf Verzeichnisse, nicht jedoch auf Dateien verweisen.
- ❏ Für jede reguläre Datei existiert mindestens ein Hard-Link im selben Dateisystem.
- ❏ Wird der letzte Symbolic Link auf eine Datei gelöscht, so wird auch die Datei selbst gelöscht.

e) Namensräume dienen u. a. der Organisation von Dateisystemen. Welche Aussage ist richtig? 2 Punkte

- ❏ Hierarchische Namensräume werden erzeugt, indem man in einem Kontext symbolische Verweise auf Dateien einträgt.
- ❏ In einem hierarchisch organisierten Namensraum dürfen gleiche Namen in unterschiedlichen Kontexten enthalten sein.
- ❏ Flache Namensräume erlauben pro Benutzer nur einen Kontext.
- ❏ Flache Namensräume sind besonders einfach implementierbar und damit vor allem für Mehrbenutzersysteme gut geeignet.

f) Welche Seitennummer und welcher Versatz gehören bei einer Seitengröße von 1024 Bytes zu folgender logischer Adresse: 0xbeef 2 Punkte

- ❏ Seitennummer 0xb, Versatz 0xeef
- ❏ Seitennummer 0x2f, Versatz 0x2ef
- ❏ Seitennummer 0xb3, Versatz 0xbb3
- ❏ Seitennummer 0xbe, Versatz 0xef

g) Ein Programm will die drei Zeichenketten
`char a[] = "dire"; char b[] = "cto"; char c[] = "ry";`
mit der Funktion `sprintf(3)` wie folgt in einen Puffer `buffer` speichern:
`sprintf(buffer, "%s/%s/%s", a, b, c);`
Mit welcher Länge (in Bytes) muss der Puffer `buffer` mindestens angelegt werden, damit kein Überlauf entstehen kann?

2 Punkte

- ❏ 10
- ❏ 11
- ❏ 12
- ❏ 13

h) Für lokale Variablen, Aufrufparameter, etc. einer Funktion wird bei vielen Prozessoren ein Stack-Frame angelegt. Welche Aussage ist richtig?

2 Punkte

- ❏ Bei rekursiven Funktionsaufrufen kann der Speicher des Stack-Frames in jedem Fall wiederverwendet werden, weil die gleiche Funktion aufgerufen wird.
- ❏ Es ist nicht möglich auf lokale *automatic*-Variablen zuzugreifen, die sich außerhalb des eigenen Stack-Frames befinden.
- ❏ Der Compiler legt zur Übersetzungszeit fest, an welcher Position im Stack-Frame der main-Funktion die globalen Variablen angelegt werden.
- ❏ Ein Pufferüberlauf eines lokalen Feldes kann sicherheitskritisch sein, da er unter Umständen zur Ausführung beliebigen Codes führt.

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).
- ❏ Der Binder erzeugt aus mehreren Programmteilen (Module) einen Prozess.
- ❏ Ein Programm kann immer nur von einem Prozess ausgeführt werden.
- ❏ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann aber auch mehrere verschiedene Programme ausführen.

j) Beim Einsatz von RAID-Systemen kann durch zusätzliche Festplatten ein fehlertolerierendes Verhalten erzielt werden. 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.
- ❏ RAID-0 erzielt Fehlertoleranz durch das Verteilen der Daten auf mehrere Festplatten.
- ❏ Bei allen RAID-Systemen ist ein höherer Schreib-Durchsatz als bei einer einzelnen Platte möglich, da alle Platten gleichzeitig beauftragt werden können.

k) Wozu dient der Maschinenbefehl *cas* (compare-and-swap)?

2 Punkte

- ❏ Um in einem System mit Seitennummerierung (Paging) Speicherseiten in die Auslagerungspartition (*swap*) schreiben zu können.
- ❏ Um bei Monoprozessorsystemen Interrupts zu sperren.
- ❏ Um bei der Implementierung von Schlossvariablen (Locks) aktives Warten zu vermeiden.
- ❏ Um auf einem Multiprozessorsystem einfache Modifikationen an Variablen ohne Sperren implementieren zu können.

Aufgabe 1.2: Mehrfachauswahl-Fragen (8 Punkte)

Bei den Multiple-Choice-Fragen in dieser Aufgabe sind jeweils *m* Aussagen angegeben, *n* ($0 \leq n \leq m$) Aussagen davon sind richtig. Kreuzen Sie **alle richtigen** Aussagen an. Jede korrekte Antwort in einer Teilaufgabe gibt einen halben Punkt, jede falsche Antwort einen halben 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 Programmfragment: 4 Punkte

```
static char a[] = "20150721";
int foo(int x) {
    static int b = 0;
    int c = 0;
    int (*d)(const int *) = foo;
    int *e = malloc(800);
    a[0] = '3';
    ++x;
    ++b;
    ++c;
    ...
}
```

Welche der folgenden Aussagen zu den Variablen im Programm sind richtig?

- ❍ Die Funktion `foo` kann von anderen Modulen aus aufgerufen werden.
- ❍ `b` hat nach mehrfacher Ausführung der Funktion `foo` immer den Wert `1`, da die Variable jedes Mal neu mit `0` initialisiert wird.
- ❍ `c` hat nach mehrfacher Ausführung der Funktion `foo` immer den Wert `1`, da die Variable jedes Mal neu mit `0` initialisiert wird.
- ❍ Das Ergebnis des Aufrufs der Funktion `foo` wird in `d` gespeichert.
- ❍ Die Zuweisung `a[0] = '3';` in Zeile 7 des Programmfragments verändert die Zeichenkette `a` und läuft ohne Fehler (z. B. *Segmentation fault*) durch.
- ❍ `e` liegt auf dem Stack.
- ❍ `e` zeigt auf ein Array, in dem Platz für 800 Ganzzahlen vom Typ `int` ist.
- ❍ Die Anweisung `++x` ändert den Wert von `x` und beeinflusst somit den Wert einer Variablen im Aufrufer.

b) Welche der folgenden Aussagen zum Thema Speicherverwaltung sind richtig? 4 Punkte

- ❍ Der Systemaufruf `free(2)` sorgt dafür, dass die angegebene Seite im Freiseitenpuffer landet.
- ❍ Das Buddy-Verfahren verhindert internen Verschnitt.
- ❍ Bei allen drei listenbasierten Verfahren (First-Fit, Best-Fit und Worst-Fit) ist externer Verschnitt möglich.
- ❍ Beim Buddy-Verfahren können zwei aneinandergrenzende Blöcke gleicher Größe immer verschmolzen werden.
- ❍ Beim Buddy-Verfahren können zwei Blöcke der Größe 2^i genau dann verschmolzen werden, wenn sich ihre Adressen in Bitposition i unterscheiden.
- ❍ Bei einer Speicheranforderung muss bei Worst-Fit u. U. die gesamte Freispeicherliste durchlaufen werden.
- ❍ Der Verschmelzungsaufwand bei Best-Fit ist verglichen mit Worst-Fit erhöht.
- ❍ Bei der Platzierungsstrategie *First-Fit* ist die Verschmelzung von freien Speicherbereichen einfacher als bei *Worst-Fit*.

Aufgabe 2: rexec (60 Punkte)

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

Schreiben Sie ein Programm `rexec`, das auf dem TCP/IPv6-Port 2015 (`LISTEN_PORT`) eine Schnittstelle anbietet, um Binärdateien entgegenzunehmen und auszuführen. Nutzer können zunächst verschiedene Binärdateien hochladen, um diese zu einem späteren Zeitpunkt gesammelt ausführen zu können. Die hochgeladenen Dateien werden im Zielverzeichnis `"/var/lib/.rexec"` (Makro `TARGET_DIR`) unter einem vom Client vorgegebenen Namen abgelegt.

Ein Client sendet pro Verbindung zunächst eine Befehlszeile mit einer Maximallänge von 100 (`MAX_LINE_LEN`) Zeichen an den Server. Der angebotene Dienst unterstützt genau zwei Befehle:

- `PUT` *`name`* zeigt dem Server an, dass eine Binärdatei unter dem Namen *`name`* angelegt werden soll. Den Inhalt der Datei (bis EOF) sendet der Client im Anschluss an die Befehlszeile über das Netzwerk an den Server.
- `RUN` führt die zuvor hochgeladenen Dateien parallel aus. Hierbei wird nicht auf das Ende der Ausführungen gewartet.

Das Programm soll folgendermaßen strukturiert sein:

- Das Hauptprogramm nimmt auf einem Socket Verbindungen an. Für jede eingehende Verbindung wird die Funktion `handle_connection()` ausgeführt. Da der Dienst auf dem Zielsystem möglichst unerkannt bleiben soll, sollen im Hauptprogramm keine Fehlermeldungen auf der Serverseite ausgegeben werden - Fehlerüberprüfung soll (und muss) dennoch stattfinden und ggf. zum Abbruch des Serverprogramms führen.
- Funktion `handle_connection(FILE *rx, FILE *tx)`: Liest die Anfragezeile vom Client ein, wertet sie aus und ruft zur Abarbeitung die passende Funktion (`put_binary()` bzw. `run_binaries()`) auf. Sollte der Client einen unbekannten Befehl gesendet haben oder die Bearbeitung des Befehls fehlgeschlagen sein, antwortet ihm der Server mit der Zeile `"Failed."` Nach einer erfolgreich bearbeiteten Anfrage wird am Ende die Zeile `"Done."` gesendet.
- Funktion `put_binary(const char *name, FILE *rx, FILE *tx)`: Öffnet zunächst die Datei mit dem Namen `name` im Zielverzeichnis. (**Tipp**: Verwenden Sie die Funktion `open(2)` mit den vorgegebenen `flags` `FILE_FLAGS` und dem `mode` `EXEC_MODE`, um einen passenden Dateideskriptor zu erhalten). Anschließend liest die Funktion die Binärdaten vom Client und speichert sie in der geöffneten Datei. Tritt ein Fehler auf, soll eine (kurze) Fehlermeldung (bei Fehler in `open(2)` z.B. `"open failed!"`) an den Client gesendet und `-1` zurückgegeben werden. Im Erfolgsfall gibt die Funktion `0` zurück.
- Funktion `run_binaries(FILE *tx)`: Durchsucht das Zielverzeichnis nach ausführbaren regulären Dateien (**Tipp**: Benutzen Sie das vorgegebene Makro `S_ISEXEC(mode)` zur Überprüfung auf Ausführbarkeit. Dieses erwartet als Parameter den Wert des `st_mode`-Feldes der `stat`-Struktur). Jede passende Datei soll in einem eigenen Prozess ohne weitere Parameter ausgeführt werden. Um nicht zu viele Dateien gleichzeitig auszuführen, soll die Anzahl der laufenden Prozesse auf 4 (`MAX_PROCS`) limitiert sein. Ist die Maximalzahl bereits erreicht, soll passiv gewartet werden, bis sich ein Kindprozess beendet. Fehler bei der Verarbeitung des Verzeichnisses und dem Erzeugen der Kindprozesse sollen ebenfalls mittels einer kurzen Nachricht an den Client gemeldet werden und `-1` zurückgegeben werden. Im Erfolgsfall liefert `run_binaries()` `0` zurück.

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 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 gegebenenfalls auch weitere Funktionen benötigen.

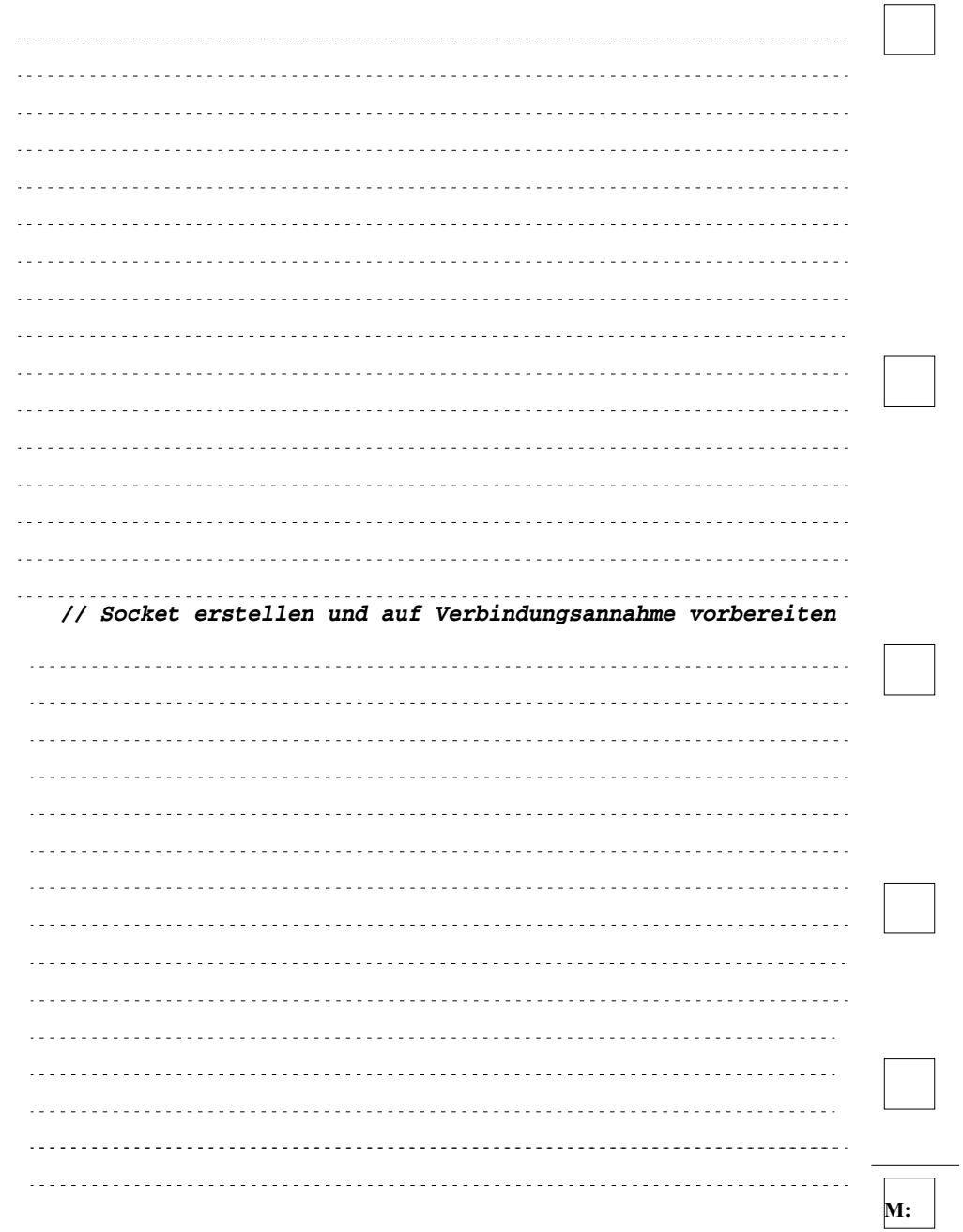

```
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

// Konstanten, Hilfsfunktionen
#define LISTEN_PORT 2015
#define MAX_LINE_LEN 100
#define TARGET_DIR "/var/lib/.rexec"
#define MAX_PROCS 4
#define FILE_FLAGS (O_CREAT | O_TRUNC | O_WRONLY)
#define EXEC_MODE (S_IRWXU | S_IXGRP | S_IXOTH)
#define S_ISEXEC(mode) (((mode) & EXEC_MODE) != 0)

static void die() {
    exit(EXIT_FAILURE);
}

// Funktionsdeklarationen, globale Variablen usw.
```

```
// Funktion main()
```



```
   // Socket erstellen und auf Verbindungsannahme vorbereiten
```


M:


```
   // Verbindungen annehmen und bearbeiten
```

```
// Ende Funktion main

// Signalbehandlung für SIGCHLD
```

```
// Ende Signalbehandlung
```

C:

```
// Funktion handle_connection()
```



H:

```
// Ende Funktion handle_connection()
```



```
// Funktion put_binary()
```




P:

```
// Ende Funktion put_binary()
```

```
// Funktion run_binaries()

   // Auftraege suchen ...
```

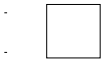
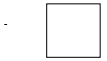
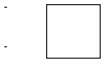
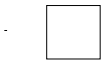




```
   // ... und jeden gefundenen Auftrag ausfuehren

   // Ende der Schleife "Auftragsbearbeitung"

// Ende Funktion run_binaries()
```

R:

Aufgabe 3: (15 Punkte)

a) Was versteht man unter einem *Inode* in einem Unix/Linux-Dateisystem? (2 Punkte)

b) Warum ist der Wert des im *Inode* eines Verzeichnisses gespeicherten Referenz-Zählers in einem Unix/Linux-Dateisystem immer mindestens 2? Welche Besonderheit liegt dabei beim Wurzelknoten "/" vor? (2 Punkte)

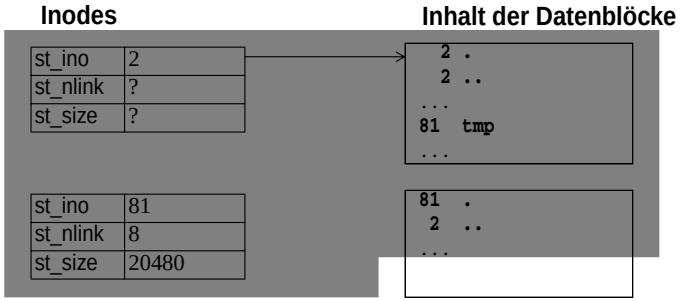
c) Gegeben ist die folgende Ausgabe des Kommandos **`ls -aoRi /tmp/sp/`** (rekursiv absteigende Ausgabe aller Dateien und Verzeichnisse unter `/tmp/sp/` mit Angabe der Inode-Nummer, des Referenzzählers und der Dateigröße) auf einem Linux-System.

```
ruprecht@fauiXX:/$ ls -aoRi /tmp/sp/
/tmp/sp/:
total 36
39 drwxr-xr-x 3 ruprecht  4096 Jul 20 12:55 .
81 drwxrwxrwt 8 root     20480 Jul 20 12:55 ..
71 -rw-rw-r-- 1 ruprecht   287 Jul 20 12:48 2015
67 lrwxrwxrwx 1 ruprecht    12 Jul 20 12:46 important -> ordner/fileX
45 drwxr-xr-x 3 ruprecht  4096 Jul 20 12:54 ordner

/tmp/sp/ordner:
total 80
45 drwxr-xr-x 3 ruprecht  4096 Jul 20 12:54 .
39 drwxr-xr-x 3 ruprecht  4096 Jul 20 12:55 ..
75 drwxr-xr-x 2 ruprecht  4096 Jul 20 12:55 deeper
49 -rw-r--r-- 2 ruprecht    70 Jul 20 12:44 file1
53 -rwxr-xr-- 1 ruprecht 58368 Jul 20 12:45 file2
49 -rw-r--r-- 2 ruprecht    70 Jul 20 12:44 fileX
73 lrwxrwxrwx 1 ruprecht     5 Jul 20 12:53 thefile -> file2

/tmp/sp/ordner/deeper:
total 8
75 drwxr-xr-x 2 ruprecht 4096 Jul 20 12:55 .
45 drwxr-xr-x 3 ruprecht 4096 Jul 20 12:54 ..
```

Ergänzen Sie im weißen Bereich die auf der folgenden Seite im grauen Bereich bereits angefangene Skizze der Inodes und Datenblöcke des Linux-Dateisystems um alle entsprechenden Informationen, die aus obiger Ausgabe entnommen werden können. (11 Punkte)



st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

st_ino	
st_nlink	
st_size	

Aufgabe 4: (15 Punkte)

Zur Koordinierung von nebenläufigen Vorgängen, die auf gemeinsame Betriebsmittel zugreifen, unterscheidet man zwischen einseitiger und mehrseitiger Synchronisation.

a) Beschreiben Sie die Unterschiede zwischen einseitiger und mehrseitiger Synchronisation. (2 Punkte).

b) Betriebsmittel lassen sich in zwei Kategorien einteilen. Nennen und beschreiben Sie diese und geben Sie je zwei Beispiele. (6 Punkte)

c) Zur Koordination des Zugriffs auf Betriebsmittel können Semaphore eingesetzt werden. Erläutern Sie das Konzept, nennen Sie die Operationen, die auf einem Semaphor definiert sind, und was diese tun. Nennen Sie außerdem ein Einsatzszenario, in dem Semaphore verwendet werden können. (7 Punkte)