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: