Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2021	**Praxisaufgabe 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	**12.07.2021 23:59 Uhr**

DOS OS

Sie arbeiten in dem kleinen Softwarestartup DOS (*Distributed Operating Systems*), dass sich auf die Entwicklung von Betriebssystemen spezialisiert hat. Vor kurzem hat Ihr Chef ein umfangreicheres Projekt gestartet, das zukünftig das Flaggschiff von DOS' Sortiement werden soll: Das *DOS OS*. Das *DOS OS* soll ein Nischenprodukt werden, das einen speziellen Anwendungsfall abdeckt: Es soll besondere Sicherheit gewähren, indem es ausschließlich auf dem RAM der ausführenden Maschine agiert. Das hat den Vorteil[1], dass keine Daten persistent gespeichert werden, was es deutlich schwieriger macht, sich unerlaubten Zugriff auf sensible Daten zu verschaffen.

Im Rahmen der Ihnen bevorstehenden Aufgabe hat sich ergeben, dass Sie sich um das Dateisystem, einen Allokator und um die Bedienung des Kaffeeautomaten kümmern.

UI

Zu einem vernünftigen Betriebssystem gehört natürlich auch ein angemessenes *user interface* (UI). Dieses wurde bereits in Form einer Konsole implementiert und kann von Ihnen benutzt werden. Es wird automatisch gestartet, sobald Sie das Projekt ausführen. Eine Übersicht über die Ihnen zur Verfügung stehenden Befehle finden sie oben in *main.c* oder mittels des Befehls *help*.

Aufgabe 2.1: Allokator (10P)

Ein Allokator ist eine Struktur, die Speicherbereiche auf Anfrage reservieren und wieder freigeben kann (wie z.B. *malloc*). In dieser Teilaufgabe sollen Sie Ihren eigenen Allokator basierend auf dem Prinzip einer verketteten Liste implementieren.

Das Konzept funktioniert folgendermaßen: Jeder reservierte Bereich wird durch ein Element in der verketteten Liste repräsentiert. Dabei besteht ein solches Element aus zwei Teilen: Dem tatsächlich reservierten Speicherbereich und einer voranstehenden Verwaltungsstruktur (siehe *mem_block* in *os_malloc.h* und Abb. 1).

Der *mem_block* enthält drei Parameter:

- **mem_block* next** ein Pointer auf des nächste Listenelement
- **unsigned int free** zeigt an, ob der betrachtete Block belegt (free = 0) oder frei (free $\neq$ 0) ist.

[1] ja, das ist tatsächlich ein Vorteil

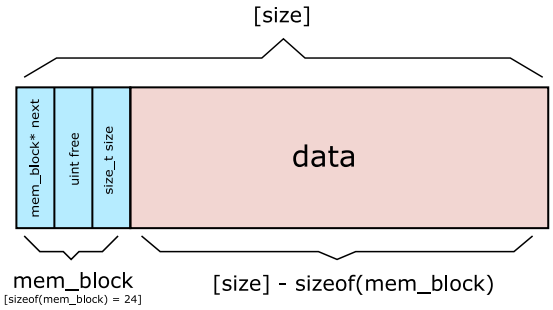

Abbildung 1: einzelnes Element der verketteten Liste

- **size_t size** gibt die Größe des Elements an. Der Parameter setzt sich somit aus der Größe des vom Benutzer angeforderten Speicherbereichs als auch aus der Größe des zugehörigen *mem_block* zusammen

Um ein neues Element (new) zu reservieren, wird zunächst ein passendes Element (old) gesucht. Ein Element gilt als passend, wenn es nicht bereits belegt ist und seine Speicherkapazität mindestens genau so groß wie die Summe aus angefragter Kapazität und Größe einer Verwaltungsstruktur ist (schließlich müssen diese beiden Dinge genügend Platz in dem ausgewählten Element finden). Wurde ein passendes Element gefunden, so kann das neue Element reserviert werden (siehe Abb. 2). Dazu sind folgende Schritte nötig:

- berechnen Sie, wie viel Speicherplatz von Element old nach dem Reservieren noch übrig ist
- erstellen Sie einen neuen *mem_block* hinter dem zu reservierenden Speicherbereich (er repräsetiert den übrig gebliebenen Speicherplatz das alten Elements nach der Reservierung des neuen Elements)
- übernehmen Sie den *next*-pointer von old in den neu erstellten *mem_block*, tragen Sie die Größe des verbliebenen Speicherbereiches ein und passen Sie den Belegungsstatus an
- passen Sie die Parameter des alten *mem_block* (der, der das neu reservierte Element repräsentiert) an: Berechnen Sie die Adresse des neuen *mem_block* und tragen Sie sie zusammen mit dem aktualisierten Belegungsstatus und der neuen *size* ein.

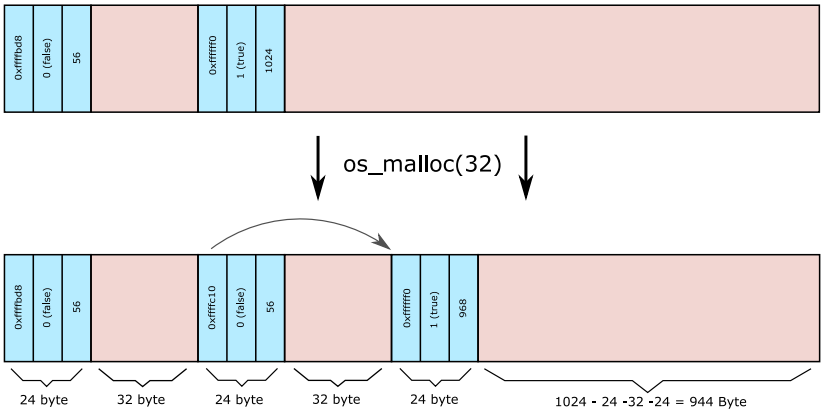

Abbildung 2: Reservieren eines neuen Elements

Neben der Allokation eines neuen Speicherbereiches muss der Allokator natürlich auch in der Lage sein, bereits reservierte Bereiche wieder freizugeben (siehe Abb. 3).

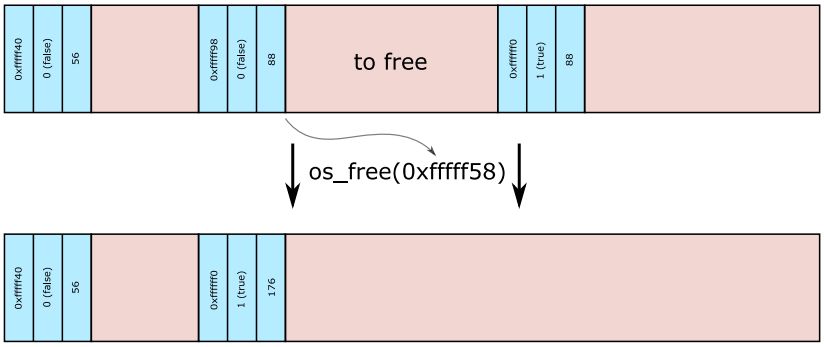

Abbildung 3: Freigabe eines Elements

Um einen Speicherbereich freizugeben, reicht es grundsätzlich aus, den Belegungsstatus des zugehörigen *mem_block* zu aktualisieren. Allerdings muss beachtet werden, dass zwei (oder mehr) nebeneinanderliegende, freie Blocke zu einem größeren Block zusammengefasst werden können[2]. Folgende Fälle sind möglich:

- **Sowohl der Block davor als auch der Block dahinter sind belegt.** In diesem Fall reicht die Aktualisierung des Belegungstatus aus.
- **Entweder der Block davor oder der Block dahinter ist belegt.** Hier müssen die beiden betroffenen Blöcke miteinander verschmolzen werden. Das heißt, der jeweils vordere *mem_block* übernimmt den *next*-pointer des hinteren und erhält seine neue Größe und der jeweils hintere *mem_block* wird gelöscht (bzw. er wird einfach „vergessen“)
- **Sowohl der Block davor als auch der Block dahinter sind frei.** Gleiche Vorgehensweise wie bei Fall 2, allerdings müssen diesmal drei Blöcke miteinander verschmolzen werden.

Um Ihnen den Einstieg in die Aufgabe zu erleichtern, wird Ihnen die Funktion *os_init()* vorgegeben. Die Funktion *os_init()* wird bei der Initialisierung des Systems aufgerufen. Dabei bekommt der Allokator den Speicherbereich gegeben, auf dem er arbeiten darf. Bevor er aber Reservierungen und Freigaben bearbeiten kann, muss er den gegebenen Speicherbereich in ein großes Listenelement umwandeln. Dies geschieht in *os_init()* (siehe Abb. 4).

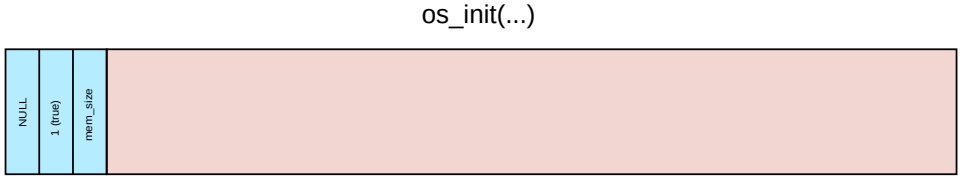

Abbildung 4: So sollte der gesamte Speicherbereich des Allokators nach *os_init()* aussehen

[2]dies ist hier Teil der Aufgabenstellung

Aufgabenstellung:

Da Ihr Chef sich für spätere Phasen des Projekts verschiedene Möglichkeiten offenhalten möchte, trägt er Ihnen zusätzlich die Aufgabe auf, verschiedene Belegungsstrategien für Ihren Allokator zu implementieren. Diese sind: *First Fit*, *Best Fit* und *Worst Fit*. Dabei beziehen sich diese Strategien ausschließlich auf den Schritt „finden Sie ein passendes Element", bei dem Sie ein bereits bestehendes, freies Element mit ausreichender Größe lokalisieren sollen, um darin das neue Element zu reservieren. Innerhalb des gewählten Elements sollen Sie das neue Element aber ganz an den Anfang legen.

a) **os_malloc_first_fit() (2P)**
Implementieren Sie in der Funktion *os_malloc_first_fit()* einen Allokator nach dem oben beschriebenen Prinzip, der die *first fit*-Belegungsstrategie verwendet!

b) **os_malloc_best_fit() (2P)**
Implementieren Sie in der Funktion *os_malloc_best_fit()* einen Allokator nach dem oben beschriebenen Prinzip, der die *best fit*-Belegungsstrategie verwendet!

c) **os_malloc_worst_fit() (2P)**
Implementieren Sie in der Funktion *os_malloc_worst_fit()* einen Allokator nach dem oben beschriebenen Prinzip, der die *worst fit*-Belegungsstrategie verwendet!

d) **os_free() (4P)**
Implementieren Sie die Funktion *os_free()* so, wie oben beschrieben!

Beachten Sie:

- da der für den Allokator nutzbare Speicherplatz nicht unbegrenzt ist, hat das letzte Element der List keinen Nachfolger, sein *next*-pointer hat den Wert NULL. Beachten Sie dies beim Traversieren Ihrer Liste
- beachten Sie außerdem, dass sich der *size*-Parameter eines jeden *mem_block* auf die Summe aus der Größe des reservierten Speicherplatzes und seiner eigenen bezieht
- in den oben gegebenen Beispielen wird davon ausgegangen, dass *sizeof(mem_block) = 24* gilt. Gehen Sie **nicht** davon aus, dass dies auch auf Ihrere Maschine der Fall ist. Benutzen Sie stattdessen immer den *sizeof()*-Operator
- *os_malloc()*-Aufrufe müssen selbstverständlich den Pointer auf den reservierten Speicherplatz, nicht auf sich selbst, zurückgeben.
- analog zum oberen Punkt bekommen *os_free()*-Aufrufe nur den Pointer auf den freizugebenden Speicherplatz anstelle des Pointers auf seinen *mem_block*

- der Allokator muss ein 8-Byte-Alignment einhalten. Das heißt, dass er nur Speicherbereiche reservieren darf, deren Größe ein Vielfaches von 8 sind. Beispiel: *os_malloc(19)* würde einen Pointer auf ein Speicherbereich mit einer Größe von 24 Byte zurückgeben.
- es dürfen keine nicht-nutzbaren Speicherbererich entstehen. Damit ist folgendes gemeint: Dadurch, dass jedes Element einen Verwaltungsblock braucht, ergibt sich eine Mindestgröße für jedes Element (24 Byte Verwaltungsblock + 8 Byte Daten = 32 Byte). Sollte eine Allokation dafür sorgen, dass ein Speicherbereich entsteht, der kleiner als besagte Mindestgröße ist, so muss dieser kleine Speicherbereich dem reservierten Bereich hinzugefügt werden.
- sollte für eine *os_malloc()*-Anfrage nicht genügend Speicherplatz vorhanden sein, muss NULL zurückgegben werden
- der Aufruf *os_free(NULL)* soll - wie *free(NULL)* auch - erlaubt sein, hat allerdings keine Wirkung
- double-frees (zweimal *os_free()* auf den gleichen Pointer) müssen nicht extra abgefangen oder behandelt werden

Aufgabe 2.2: Dateisystem (20P)

Die Struktur des Dateisystems ist bereits vorgegeben (siehe *filetree.h*). Es basiert auf vier Datentypen:

```
1  /* Node
2   * abstract tree node, is 'derived' by File or Directory
3   * NOTE: This struct should ALWAYS be instantiated as part of
        either File or Directory struct.
4   */
5  struct Node{
6      uint32_t flags;
7      Directory * parent;
8      Node * next;
9      Node * prev;
10     char * name;
11 };
```

Listing 1: struct node

Eine Node ist ein Wrapper für alle möglichen Elemente des Dateisystems. Jedes Element des Dateisystems lässt sich also über eine solche generische Struktur beschreiben. Eine Node besteht aus den folgenden Variablen:

- **flags** gibt Aussagen über den tatsächlichen Typ des daran gebundenen Elements (File oder Directory)
- **parent** jedes Element hat ein Elternverzeichnis (bis auf root)
- **next** innerhalb eines Verzeichnisses werden alle darin enthaltenen Elemente mittels einer doppelt verketteten Liste gespeichert. Dieser pointer zeigt auf das nächste Element
- **prev** zeigt auf das vorherige Element im gleichen Verzeichnis
- **name** Name der Datei oder des Verzeichnisses. Der Name der root ist der leere string "".

```c
/* Tree
 * file tree holding a reference to the root folder
 */
struct Tree{
    Directory * root;
};
```

Listing 2: struct tree

Ein Tree ist eine minimale struktur um einen Dateibaum einfach halten zu können. Er besteht lediglich aus einem Directory-pointer, der das root-Verzeichnis dieses Baumes referenziert.

```c
/* Directory
 * a directory in the file tree, derives from Node
 */
struct Directory{
    Node node; /* Directory 'derives' from Node */
    Node * first_child;
};
```

Listing 3: struct directory

Directories repräsentieren die Klasse der Verzeichnisse und können somit eine Menge an Kindern (Dateien und Verzeichnisse) beinhalten. Sie bestehen aus einer Node und einem pointer, der ihr erstes Kind referenziert. Beachten Sie, dass es sich hier bei der node-Variable nicht um einen Node-pointer, sondern tatsächlich um ein struct vom Typ Node handelt. Dies sorgt dafür, dass aus Sicht des Speichers ein Directory exakt so wie eine Node aufgebaut ist, mit der Ausnahme, dass ein Directory nach dem `char * name` noch die Variable `Node * first_child` beinhaltet. Das ermöglicht es, ein Directory zu einer Node zu casten und wie eine solche zu benutzen (siehe Beispiel unten).

```c
/* File
 * a file in the file tree, derives from Node
 */
struct File{
    Node node; /* File 'derives' from Node */
    void * data;
    size_t data_len;
};
```

Listing 4: struct file

Im Gegensatz zu Directories können Files zwar keine Kinder enthalten, dafür aber Daten abspeichern. Dazu enthält das struct neben der obligatorischen Node einen void-pointer (also einen pointer, der nicht auf einen ganz bestimmten Datentyp zeigen muss) und eine Variable, um die Größe der Datei speichern zu können.

Beispiel

Es folgen Beispiele, die die Verwendung der structs genauer erläutert.

Bei Erzeugung eines neuen Nodes muss klar sein, um welchen Typ es sich handelt: File oder Directory. Ein eigenständiger Node sollte niemals erzeugt werden.

```
1 /* Directory erzeugen */
2 Directory * dir = ALLOCATE(sizeof(Directory));
3
4 /* File erzeugen */
5 File * file = ALLOCATE(sizeof(File));
6
7 /* schlechte Idee, siehe unten */
8 Node * n = ALLOCATE(sizeof(Node));
```

Da das Directory- bzw. File-struct einen Node enthält, können wir auf dieses Node-'Objekt' zugreifen.

```
1 /* So kann z.B. das richtige flag beim directory gesetzt
     werden */
2 dir->node.flags = FILE_TREE_FLAG_DIRECTORY;
3
4 /* ebenso bei File */
5 file->node.flags = 0;
```

Und nun zur Pointer-Magie: Da der Node als erstes in der Directory-Struct steht (siehe *filetree.h*) sind die Adressen `dir` und `&dir->node` identisch. Dies bedeutet, dass ein Directory-Pointer als ein Node-Pointer interpretiert werden kann. Analoges gilt für das File-struct.

```
1 Node * node = (Node*)dir;
2 node->parent; /* identisch zu dir->node.parent */
```

Umgekehrt kann ein Node-Pointer als Directory- oder File-Pointer interpretiert werden. Hierbei muss das entsprechende Flag überprüft werden, wie in der folgenden Beispiel-Funktion gezeigt:

```
1 void print_type(Node * node){
2         if(node->flags & FILE_TREE_FLAG_DIRECTORY){
3                 Directory * d = (Directory*)node;
4                 puts("Node is a Directory!");
5         }else{
6                 File * f = (File*)node;
7                 puts("Node is a File!");
8         }
9 }
```

Dementsprechend sollte an dieser Stelle auch klar sein, warum ein Node niemals einzeln erstellt werden sollte. Funktionen wie die obige erwarten, dass ein Node entweder ein Directory oder ein File ist.

Zusatz-Info: Im Kontext von Objekt-Orientierten Programmiersprachen würde *Node* als eine abstrakte Klasse bezeichnet werden. Mit dem Schlüsselwort `abstract` kann z.B. in

der Programmiersprache Java eine Klasse deklariert werden, die nicht direkt instanziiert werden kann. Ein *Node*-Objekt kann somit nur als Teil eines erbenden Objektes (*Directory* oder *File*) existieren.

Aufgabenstellung:

Implementieren Sie in der Datei *filetree.c* alle mittels */* TODO: ... */* gekennzeichneten Funktionen. Berücksichtigen Sie dabei die Kommentare in *filetree.c* und *filetree.h*. Verwenden Sie für die Allozierung und Freigabe dynamischer Daten die respektiven Makros *ALLOCATE(SIZE)* und *FREE(PTR)*. Dies hat den Vorteil, dass Sie somit zum Testen zwischen ihrer eigenen und der Standard-Implementierung von *malloc()* bzw. *free()* wechseln können (siehe *filetree.h*).

a) **filetree_new() (1P)**
Diese Funktion soll einen neuen Tree erzeugen und mit einem leeren *root*-Directory initialisieren. Das *root*-Directory hat als Namen einen leeren string `""` (**Achtung:** Damit ist nicht *NULL* gemeint!). Der parent von *root* ist NULL.

b) **filetree_destroy() (2P)**
Geben Sie allen Speicher frei, der vom Tree alloziert wurde. Also auch alle Files/Directories die erzeugt wurden. Dies ist am einfachsten rekursiv zu lösen.

c) **filetree_name_valid() (1P)**
Überprüfen Sie ob der gegebene string eine valide Bezeichnung für ein File/Directory darstellt. Gültige Namen sind dabei **nicht-leere** strings, die ausschließlich Groß- und Kleinbuchstaben (`a-z`, `A-Z`), Ziffern (`0-9`), sowie die Sonderzeichen `._-` (Punkt, Unterstrich, Bindestrich) enthalten. Außerdem sind die Namen "`.`" (current directory), sowie "`..`" (parent directory) reserviert und werden deshalb als invalide File/Directory -Namen behandelt. Geben Sie den der Error-Code `FILE_TREE_ERROR_ILLEGAL_NAME` zurück, falls der Name invalide ist, ansonsten `FILE_TREE_SUCCESS`.

d) **filetree_mkdir() (1P)**
Erzeugen Sie einen neuen, leeren Directory im gegebenen Directory `parent` mit dem gegebenen Namen. Prüfen Sie dafür zunächst, ob der Name valide ist und geben Sie `FILE_TREE_ERROR_ILLEGAL_NAME` zurück falls dies nicht der Fall ist. Danach überprüfen Sie, ob ein File oder Directory mit dem gegebenen Namen bereits in dem Ordner existiert. Ist dies der Fall, geben Sie den Fehler-Code `FILE_TREE_ERROR_DUPLICATE_NAME` zurück. Nun erstellen Sie ein neues Directory-Struct (dynamischer Speicher!) und fügen dieses **am Anfang** der verketteten Liste

im parent-Directory hinzu. Hierzu müssen Sie auch den `first_child`-Pointer im parent-Directory entsprechend anpassen. Achten Sie auf die korrekte Initialisierung aller Variablen des Struct (`flags`, `parent`, etc.). Achten Sie zudem darauf, dass eine Kopie des an die Funktion übergebenen `const char * name` in der `name`-Variable des Node-Structs im Directory erstellt werden muss (neuen Speicher reservieren!). Kann das Directory erfolgreich erstellt werden so geben Sie `FILE_TREE_SUCCESS` zurück. Sie können davon ausgehen, dass für alle Funktions-Parameter eine gültige Adresse übergeben wird.

e) **filetree_mkfile() (1P)**
Erzeugen Sie ein neues File im gegebenen Directory `parent` mit dem gegebenen Namen. Entsprechend der Funktion `filetree_mkdir()` nehmen Sie die gleichen Überprüfungen vor (Name valide?, existiert File/Directory mit gleichem Namen bereits?), erstellen anschließend das File und fügen dieses **am Anfang** in die verkettet Liste im parent-Directory ein. Auch hier sollten Sie auf die entsprechende Initialisierung aller Variablen des File-Struct achten. Kopieren Sie außerdem die gegebenen Daten `const void * data` in das File (neuen Speicher reservieren!) und setzen Sie die Variable `data_len` im File-Struct. Sie können davon ausgehen, dass für `Directory * parent` und `const char * name` eine gültige Adresse übergeben wird. Für den Parameter `const void * data` jedoch kann `NULL` übergeben werden, dann wird die `data`-Variable im File-Struct auch auf `NULL` gesetzt und damit signalisiert, dass das File leer ist. Kann das File erfolgreich erstellt werden so geben Sie `FILE_TREE_SUCCESS` zurück.

f) **filetree_ls() (2P)**
In dieser Funktion sollen der Inhalt eines gegebenen Directory ausgegeben werden. Iterieren Sie dazu über die Liste der Nodes in dem übergebenen Directory und geben Sie den Name und einen Typ-Bezeichner, `FILE` oder `DIRECTORY`, für jedes File/Directory in einer separaten Zeile auf der Konsole (z.B. mittels `printf()`) aus.
Es sind genau 5 Leerzeichen zwischen Name und Typ-Bezeichner des Nodes mit dem längsten Namen auszugeben. Zusätzliche Leerzeichen sind bei Nodes mit kürzeren Namen einzufügen, damit die Ausgaben der Typ-Bezeichner genau übereinander stehen.
Beispiel: Im root-Directory liegt ein Directory mit dem Namen `my_dir` und eine Datei `my_file.txt`, die Ausgabe von `filetree_ls(root)` sieht dann wie folgt aus:

```
my_dir          DIRECTORY
my_file.txt     FILE
```

Es muss also zunächst der längste Name ermittelt werden, bevor mit der Ausgabe der ersten Zeile gestartet werden kann. Achten Sie bitte darauf, exakt diese Vorgaben einzuhalten. Nutzen sie für die Überprüfung Ihrere Ausgabe auch die in der Vorgabe enthaltenen automatischen Tests. Sie können davon ausgehen, dass für alle Funktions-Parameter eine gültige Adresse übergeben wird.

g) **filetree_find() (3P)**
Diese Funktion soll den gesamten Baum rekursiv, von einem Start-Directory ausgehend, ausgeben. Dazu wird absolute Pfad vom Start-Directory und von jedem File/-Directory die sich in dem Start-Directory oder einem Unterverzeichnis davon befinden auf einer separaten Zeile ausgegeben. Der Baum wird in *preorder*-Reihenfolge ausgegeben (also Files/Directories die näher an der root liegen zuerst), stets angefangen beim Anfang der verketteten Liste (`first_child`). Zusätzlich wird der Parameter `const char * name` übergeben. Ist dieser Parameter nicht NULL und ist der string nicht-leer (d.h. nicht der string `""`), so werden nur die absoluten Pfade von Files/Directories ausgegeben, die exakt diesen Namen haben (case-sensitive).
Beispiel: Angenommen der File-Tree beinhaltet im root-Directory ein Directory `a` und in diesem Directory liegt ein File `b` (`first_child`) und ein File `c` (`first_child->next`).
Der Aufruf `filetree_find(root, NULL)` ergibt folgende Ausgabe:

```
/
/a
/a/b
/a/c
```

Der Aufruf `filetree_find(root->first_child, "")` ergibt:

```
/a
/a/b
/a/c
```

Der Aufruf `filetree_find(root, "a")` ergibt:

```
/a
```

Der Aufruf `filetree_find(root, "B")` gibt nichts aus. Sie können davon ausgehen, dass für den Parameter `Directory * start` eine gültige Adresse übergeben wird.

h) **filetree_mv() (2P)**
In dieser Funktion verschieben Sie ein File/Directory an einen neuen Ort. Ihre Aufgabe ist es also, den übergebenen `Node * source` aus der verketteten Liste seines parent zu entfernen und an den Anfang der List im `Directory * destination` wieder einzufügen. Dies soll auch dann geschehen, wenn `destination` bereits der direkte parent von `source` ist (dies setzt den Node an den Anfang der Liste, ohne ihn tatsächlich zu verschieben). Ist jedoch das `destination`-Directory ein Unterverzeichnis vom `source`-Node (oder `source == destination`), soll nichts verschoben werden und der Error-Code `FILE_TREE_ERROR_SUBDIR_OF_ITSELF` zurückgegeben werden. Auch wenn bereits ein File/Directory im `destination`-Directory mit dem gleichen Namen wie vom `source`-Node existiert, soll nichts verschoben werden und der Error-Code `FILE_TREE_ERROR_DUPLICATE_NAME` zurückgegeben werden. Wurde die `source`-Node erfolgreich verschoben, wird `FILE_TREE_SUCCESS` zurückgegeben. Sie können davon ausgehen, dass für alle Funktions-Parameter eine gültige Adresse übergeben wird.

i) **filetree_rm() (1P)**
Diese Funktion entfernt den gegebenen Node aus der verketteten Liste im parent-

Directory und gibt allen vom Node allozierten Speicher frei. Ist die übergebene Node ein Directory, so müssen rekursiv auch alle Unterverzeichnisse freigegeben werden. Sollte die root übergeben werden (`node->parent == NULL`), so wird nichts gelöscht und als Fehler-Code `FILE_TREE_ERROR_RM_ROOT` zurückgegeben. Bei erfolgreichem Entfernen des Node wird `FILE_TREE_SUCCESS` zurückgegeben. Sie können davon ausgehen, dass stets eine gültige Adresse für `Node * node` übergeben wird.

j) **filetree_print_file() (1P)**
In dieser Funktion soll der Inhalt einer Datei (als ASCII-Zeichen interpretiert) auf der Konsole ausgegeben werden. Am Ende der Daten soll noch ein Zeilenumbruch (`\n`) ausgegeben werden. Beachten Sie, dass die Daten in einem File nicht unbedingt Null-Terminiert sind. Der folgende Aufruf wäre also **fehlerhaft**:
`printf("%s\n", (char*)file->data);`
Gehen Sie davon aus, dass stets eine gültige Adresse für `File * file` übergeben wird.

k) **filetree_resolve_path() (4P)**
Diese Funktion soll einen absoluten oder relativen Pfad erhalten und den dazugehörigen Node in dem Parameter `n` zurückgeben. Ein absoluter Pfad fängt immer mit `/`, von der root ausgehend, an. Ein relativer Pfad fängt direkt mit einem Namen, ausgehend vom gegebenen `Directory * current_dir`, an. Um in ein Verzeichnis "hineinzugehen" wird eine beliebige Anzahl von `/` (min. 1) verwendet. Ein Pfad endet zudem mit einer beliebigen Anzahl an `/` oder direkt mit dem Namen des zu ermittelnden Node. Ein Punkt `.` bedeutet "dieses Vezeichnis", es wird hierbei also nicht tiefer in den Baum gegangen. Zwei Punkte `..` bedeuten "vorheriges Verzeichnis", es wird hierbei also zum parent des aktuellen Verzeichnis gegangen. Der Pfad `""` (leerer string) oder `"/"` steht für das root Verzeichnis.
Wird ein Node an dem gegebenen Pfad gefunden so wird `*n` auf den Pointer des gefundenen Nodes gesetzt und `FILE_TREE_SUCCESS` zurückgegeben. Wird kein entsprechender Node an dem gegebenen Pfad gefunden wird `FILE_TREE_ERROR_NOT_FOUND` zurückgegeben. Beachten Sie, dass die root keinen parent hat, dementsprechend existiert z.B. `/..` nicht (`FILE_TREE_ERROR_NOT_FOUND` zurückgeben).
Beispiel: Angenommen der File-Tree `t` beinhaltet im root-Directory ein Directory `a` und in diesem Directory liegt ein File `b`. Der Parameter `Node * node` liefert das Ergebnis zurück.
`filetree_resolve_path(t, "", t->root, &node)` gibt in `node` einen pointer auf die root zurück.
`filetree_resolve_path(t, "/a/", t->root, &node)` gibt in `node` einen pointer auf den Node `a` zurück.
`filetree_resolve_path(t, ".//b", a, &node)` gibt in `node` einen pointer auf den Node `b` zurück.
`filetree_resolve_path(t, "b", a, &node)` gibt in `node` einen pointer auf den Node `b` zurück.
`filetree_resolve_path(t, "/a/..", a, &node)` gibt in `node` einen poin-

ter auf die root zurück.
`filetree_resolve_path(t, "./c", a, &node)` schreibt nichts in `node` und gibt `FILE_TREE_ERROR_NOT_FOUND` zurück.
Sie können davon ausgehen, dass für alle Funktions-Parameter eine gültige Adresse übergeben wird.

l) **filetree_get_path() (1P)**
Diese Funktion liefert den absolution Pfad eines gegebenen Nodes als string. Der zurückgegebene Pfad hat **kein** / am Ende. Die Funktion soll dynamisch Speicher für den Pfad reservieren und zurückgeben. Wird die root übergeben soll ein leerer string `""` zurückgegeben werden (**Achtung**: Auch für diesen muss dynamisch Speicher reserviert werden). Der Aufrufer ist dafür verantwortlich, diesen Speicher mittels `FREE()` wieder freizugeben. **Beispiel:** Angenommen der File-Tree beinhaltet im root-Directory ein Directory `a` und in diesem Directory liegt ein File `b`.
`filetree_get_path(root)` gibt `""` zurück.
`filetree_get_path(a)` gibt `"/a"` zurück.
`filetree_get_path(b)` gibt `"/a/b"` zurück.
Gehen Sie davon aus, dass stets eine gültige Adresse für `Node * n` übergeben wird.

Beachten Sie:

- Die Verwendung der eigens implementierten Funktionen *os_malloc() os_free()* ist nicht verpflichtend und wird daher in diesem Aufgabenteil nicht bewertet.
- Ein Allokator (sowohl malloc, als auch ihre eigene Version) interessiert sich nicht für den ihm übergebenen pointer-Typ (der sowieso nur für den Compiler relevant ist), wenn es z.B. um Freigaben geht. Er gibt dann lediglich den mit dem übergebenen pointer assoziierten Speicherbereich frei. Das heißt: Sollten sie zuerst dynamisch ein Directory anlegen, den Zeiger darauf zu einem Node-pointer casten und anschließend diesen Node-pointer freigeben, so wird der gesamte (für das Directory reservierte) Speicherbereich freigegeben und nicht nur die Menge an Speicher, die eine einzelne Node einnehmen würde.

Aufgabe 2.3: Kaffeeautomat (0P)

Bereiten Sie sich in der Küche eine Tasse Kaffee oder ein alternatives Getränk Ihrer Wahl zu.

Hinweise:

- **Make** Um ein Projekt in C zu kompilieren verwendet man C-Compiler wie GCC. Diese werden klassisch über die Konsole ausgeführt (z.B. `gcc foo.c bar.c -o foobar` um aus den beiden Programmtexten `foo.c` und `bar.c` die ausführbare Datei `foobar` zu erzeugen). Gerade bei größeren Projekten mit vielen Programmtexten

ist es umständlich, solche langen Befehle immer wieder erneut eingeben zu müssen. Das Programm *make* schafft hier Abhilfe.

Make nutzt die im Makefile der Vorgabe bereits enthaltenen targets um das Projekt entsprechend zu kompilieren. `make all` kompiliert dabei das ganze Projekt. Die folgenden targets stehen Ihnen zur Verfügung:

  - **all** kompiliert das gesamte Projekt
  - **run** wie **all**, nur dass die ausführbare Datei auch direkt ausgeführt wird
  - **clean** entfernt alle durch *make* generierten Dateien

- **Tests** Zusätzlich zu diesen Standardtargets werden Ihnen auch Testtargets angeboten, mit denen Sie unkompliziert kleine Tests über die von Ihnen implementierten Funktionen laufen lassen können. Die zugehörigen Tests befinden sich im *tests/*-Ordner und dürfen von Ihnen nach Belieben angepasst werden (Sie dürfen sich bei Bedarf sogar neue Tests schreiben). Das Ergebnis eines jeden Tests wird in *tests/output/* geschrieben.

  Um einen Test auszuführen reicht ein einfacher Aufruf von *make* mit dem Testnahmen ohne Dateiendung. So würde der Aufruf für den Test *test_malloc_best_fit.c* wie folgt aussehen: *make test_malloc_best_fit*

  In dem *tests/*-Ordner befinden sich auch drei besondere Tests, welche auf das Stichwort *all* enden. Diese Tests führen alle anderen Tests aus, die zum entsprechenden Aufgabenteil gehören. So führt der Test *tests_malloc_all* all diejenigen Tests aus, die mit dem Allokator in Verbindung stehen (Achtung: diese Auswahl geschieht über Stringvergleiche).

  Beachten Sie, dass ein bestandener Test nicht zwingend bedeutet, dass Ihre Implementierung vollständig und korrekt ist.

- **Abgabe** Die Abgabe erfolgt allein mit Hilfe des Befehls `make submission`. Dieser verschnürt die Code-Dateien `filetree.c` und `os_malloc.c` zu einem ZIP-Archiv, welches bei ISIS hochzuladen ist. Es kann beliebig oft abgegeben werden, bewertet wird nur die letzte Abgabe. Die Abgabe muss mit dem Makefile der Vorgabe kompilierbar sein. Abzug gibt es für Compilerwarnings und Speicherfehler. Außerdem führen wir Plagiatsprüfungen durch!

- **Referenzsystem** Da der Zugang zu den Universitätsservern über ssh bzw. vpn immer noch eingeschränkt ist, verwenden wir als Referenzsystem Ubuntu 20.04. Abgaben, die unter Windows bzw. Mac funktionieren, tun dies u.U. nicht auf Linux Systemen!