Übungsblatt 2

Abgabe: 20. Mai 2021

Aufgabe 2.1: C-Programmierung: Lineare Listen (5 + 5 = 10 Punkte)

Zeiger sind gut zum Aufbau komplexer Datenstrukturen geeignet. Ein Beispiel dafür sind **lineare Listen**. Eine lineare Liste ist eine Menge von Objekten gleichen Typs, die über Zeiger miteinander verkettet sind. Ein Beispiel ist in Abbildung 1 gegeben. Hier wird eine Liste verwaltet, in der Studierende mit Name und Matrikelnummer erfasst sind. Die Liste ist nach Matrikelnummern aufsteigend sortiert.

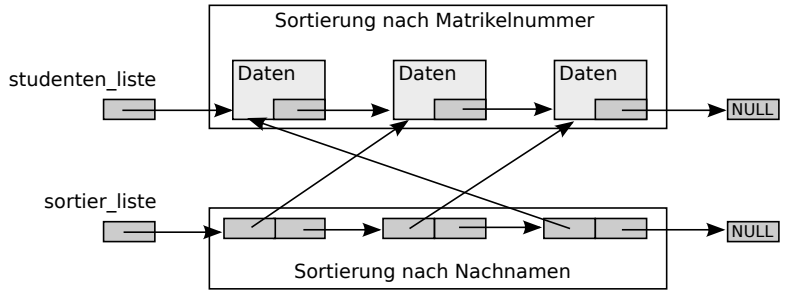

Abbildung 1: Aufbau von verketteten Listen.

Folgende Zugriffsfunktionen stehen für die lineare Liste zur Verfügung:

`is_empty()` testet, ob die Liste leer ist

`enqueue()` fügt einen neuen Studierenden in die Liste ein

`dequeue()` löscht einen Studierenden aus der Liste

`get_student()` liest die Daten zu einem Studierenden aus der Liste aus

a) Zusammen mit dem Übungsblatt finden Sie im Lernraum die Quellcodedatei `listen.c`. Studieren Sie den Quellcode, compilieren Sie das Programm und testen Sie es. Sie werden feststellen, dass die Funktionen `enqueue` und `dequeue` noch nicht implementiert sind. Implementieren Sie beide Funktionen mit Hilfe der im Codegerüst angegeben Kommentare. Ihr Programm muss mit
`gcc -std=c11 -Wall -Wextra -pedantic -fsanitize=address,undefined -o OUTFILE listen.c`
ohne Warnungen kompilieren. Dokumentieren Sie in der Abgabe Ihrer Lösung die Ausgabe des Programms.

b) Es soll ermöglicht werden, die Liste statt nach Matrikelnummer nach einem beliebigen Merkmal zu sortieren. Schreiben Sie hierzu eine Funktion, die eine weitere lineare Liste `sort_students` erzeugt. Die Elemente dieser Liste sollen keine eigenen Einträge des jeweiligen Datums enthalten, sondern lediglich auf die ursprünglichen Elemente der Datenbank verweisen (vgl. Abbildung 1). Nutzen Sie zur Implementierung Funktionspointer, um die jeweilige Vergleichsmethode zur Sortierung einfach anzupassen. Übergeben Sie den Funktionspointer als Parameter zu Ihrer Funktion, die die neue `sortier_liste` erzeugt. Elemente die als gleich eingestuft werden sollen hierbei in derselben Sortierung vorkommen, wie in der Ursprungsliste. Schreiben Sie zwei Vergleichsfunktionen, die als Parameter übergeben werden können, um die Liste entweder nach Vor- oder Nachname sortieren zu können.

   Fügen Sie danach am Ende der `main` Methode noch ein Nutzungsbeispiel ein: Erzeugen Sie eine nach Vornamen sortierte Liste und geben Sie diese aus. Erzeugen Sie dann eine nach Nachnamen sortierte Liste und geben diese ebenfalls aus. Benutzen sie dasselbe Ausgabeformat wie `test_dump`.

Hinweis: Ihre Vergleichsmethoden müssen alle die selbe Methodensignatur aufweisen; halten Sie es also allgemein und verwenden sie stets diese Signatur: `int compare(stud_type const* a, stud_type const* b)`. Typischerweise nutzt man einen `int` als Rückgabewert, um $a < b$, $a == b$ und $a > b$ durch eine negative Zahl, 0 und eine positive Zahl darzustellen.

Aufgabe 2.2: Praktisches Prozessmanagment (0,5 + 0,5 + 1 = 2 Punkte)

Betrachten Sie folgendes kleines Beispielprogramm:

```
#include <stdio.h>

int main(void){
        for(unsigned char i = 0; i < 500; ++i) {
                printf("%i ",i);
        }
        return 0;
}
```

Sie führen das Programm im Hintergrund aus und merken, dass es in einer Endlosschleife hängt.

a) Wie können Sie das Programm wieder abbrechen?

b) Wie lassen sich die aktuell auf dem Rechner laufenden Prozesse anzeigen? Geben Sie den Befehl mit allen notwendigen Argumenten an!

c) Warum hängt das Programm in einer Endlosschleife? Welche Modifikation müssen Sie vornehmen, damit es nach der vorgesehenen Zahl an Schleifendurchläufen terminiert?

Aufgabe 2.3: Fork (3 + 1 = 4 Punkte)

a) Wir betrachten folgendes C-Programm:

```
#include <unistd.h>
#include <sys/wait.h>

int main() {
    if (fork() == 0) {
        write(1, "B", 1);
        if (fork() == 0) {
            write(1, "u", 1);
        } else {
            write(1, "S", 1);
        }
    } else {
        fork();
        write(1, "20", 1);
        if (fork() == 0) {
            write(1, "1", 1);
        } else {
            write(1, "9", 1);
        }
    }
    return 0;
}
```

Zeichnen Sie einen Baum, der die Prozessstruktur darstellt. Die *Knoten* des Baums sollen jeweils die `fork`*-Verzweigstellen des Programms* repräsentieren. Verzweigen Sie den Baum an diesen Stellen, indem Sie mit dem *Kindprozess im linken* und mit dem *Elternprozess im rechten Teilbaum* fortfahren. Die Blätter des Baums beschreiben so jeweils genau einen Prozess im Lebenszyklus des Programms. Schreiben Sie in die *Blätter* des Baums alle *Ausgaben, die* `write` *für diesen Prozess erzeugt hat*.

b) Betrachten Sie die Ausgabe von ‘u’ und ‘S’. Welches der beiden Zeichen wird eher ausgegeben? Begründen Sie Ihre Antwort.

Aufgabe 2.4: Prozesse unter Linux (1 + 1 + 1 + 1 = 4 Punkte)

Der zentrale Begriff in Betriebssystemen ist der Prozess. Das Prozesssystem unter Linux gleicht in gewissem Sinne dem Dateisystem: Prozesse bilden eine hierarchische Baumstruktur, an deren Wurzel ein `init`-Prozess steht, der zusammen mit dem Linux-System gestartet wird. Ein Prozess kann mehrere Kindprozesse erzeugen, jeder Kindprozess hat hingegen genau einen Elternprozess.

Die Identifikation der Prozesse erfolgt über eine Nummer, die Prozess-ID (`PID`). Welche Prozesse gerade laufen und welche PID sie haben, können Sie mit dem Kommando `ps` (process status) ermitteln.

Die Erzeugung eines neuen Prozesses geschieht unter C z.B. mit der Funktion `posix_spawn()`. Durch diesen Aufruf wird aus einem Prozess heraus ein zweiter Prozess (Kindprozess) gestartet.

Mit der Funktion `waitpid()` kann der Elternprozess so lange schlafen gelegt werden, bis der Kindprozess terminiert.

a) Zusammen mit diesem Übungsblatt finden Sie im Lernraum zwei Programme: `parent.c` und `child1.c`. Kompilieren Sie die Programme zu `parent` und `child1` respektive und starten Sie `parent` in demselben Ordner in dem beide Programme liegen. Bitte warten Sie nicht darauf, dass die Programme terminieren; sie beinhalten Endlosschleifen.

   Modifizieren Sie `parent.c` derart, dass zwei Kindprozesse gestartet werden. Der zweite Kindprozess soll ein weiteres Programm `child2.c` bzw. `child2` ausführen, welches (analog zu den bestehenden) das Zeichen ‘C’ ausgibt.

b) Ihnen ist sicher aufgefallen, dass die Zeichen auf zwei verschiedene Arten auf die Standardausgabe geschrieben werden (welches man im Allgemeinen nicht tun sollte). Der Kindprozess nutzt `fprintf`, welches zur Standard-C-Bibliothek (libc) gehört. Der Vater-Prozess verwendet `write`, welches auf den gleichnamigen Syscall zurückfällt. Was bewirkt das Kommando `fflush` beim `fprintf` und wieso wird es beim `write` nicht verwendet? Welche der beiden Varianten halten Sie ganz allgemein (sprich: wenn Sie mehr als nur einen Schreibaufruf zwischen `fflush` aufrufen durchführen) für effizienter?

c) Nehmen Sie nun an, dass einer der beiden Kindprozesse nur eine begrenzte Zahl an Schleifendurchläufen durchführen würde. Wie Sie in diesem Fall z.B. durch `top` feststellen können, existiert dieser Prozess auch nach seiner Beendigung noch als ‘Zombie-Prozess’ weiter. Was ist ein Zombie-Prozess und wofür ist dieser Zustand nötig?

d) Wird ein Kindprozess erzeugt, wird in seinem PCB die PID des Elternprozesses abgespeichert. Geben Sie einen Grund an, warum dies gemacht wird.