Übung 2

Abgabe: 17.5.2018 8:30

Aufgabe 2.1: C-Programmierung: Lineare Listen (4 + 3 = 7 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.

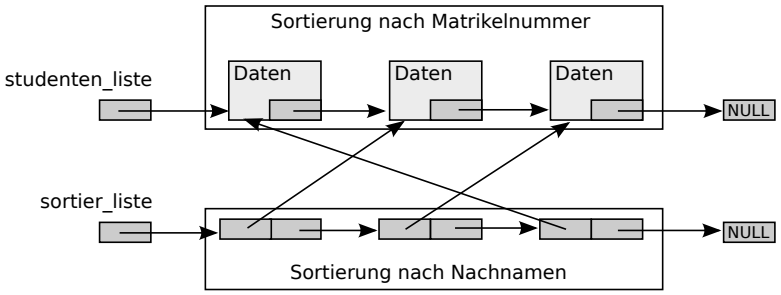

Abbildung 1: Aufbau von verketteten Listen.

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) Als Anhang zum Übungsblatt finden Sie 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 -O OUTFILE listen.c`
ohne Warnungen kompilieren. Der Anhang enthält auch ein Makefile, das ihnen hilft, das Programm mittels `make` zu kompilieren. Dokumentieren Sie in der Abgabe die Ausgabe des Programms.

b) Es soll ermöglicht werden, die Liste statt nach der Matrikelnummer, nach einem beliebigen Merkmal zu sortieren. Schreiben Sie hierzu eine Funktion, die eine weitere lineare Liste `sortier_liste` 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. Schreiben Sie zwei Vergleichsfunktionen, die als Parameter übergeben werden können, um die Liste entweder nach Vor- oder Nachname sortieren zu können.

Hinweis: Ihre Vergleichsmethoden müssen alle die selbe Methodensignatur aufweisen; halten Sie es also allgemein und verwenden sie stets diese Signatur: `int compare(stud_type* a, stud_type* b)`. Typischerweise nutzt man einen `int` als Rückgabewert, um $a < b$, $a == b$ und $a > b$ durch -1, 0 und 1 darzustellen.

Aufgabe 2.2: Prozesse unter Linux (2 + 1 + 1 + 1 + 3 = 8 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 `root`-Prozess steht, der zusammen mit dem Linux-System gestartet wird. Jeder Prozess kann mehrere Kindprozesse erzeugen, jeder Kindprozess hat hingegen genau einen Vaterprozess.

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 mit dem Kommando `fork()`. Durch diesen Aufruf wird aus einem Prozess heraus ein zweiter Prozess (Kindprozess) gestartet. Dieser erhält eine Kopie der Systemumgebung des Vaterprozesses, d.h. er arbeitet auch auf demselben Programmcode. Die Unterscheidung zwischen Vater- und Kindprozess erfolgt über den Rückgabewert von `fork()`: der Vaterprozess erhält die PID des Kindprozesses, der Kindprozess erhält als Rückgabewert stets 0. Wenn also Vater- und Kindprozess unterschiedliche Aufgaben ausführen sollen, kann im Quelltext anhand des Rückgabewertes verzweigt werden. Benötigt ein Prozess seine eigene PID, kann er sie mittels `getpid()` erfragen.

Mit der Funktion `wait()` kann der Vaterprozess so lange schlafen gelegt werden, bis der Kindprozess terminiert. Ebenfalls in Zusammenhang mit `fork()` wird häufig eine Variante von `exec()` verwendet, mit dem der Kindprozess eine neue Programmdatei lädt und ausführt.

a) Als Anhang zu diesem Übungsblatt finden Sie im L2P das Programm `prozesse.c`. Compilieren und starten Sie es. Bitte warten Sie nicht darauf, dass das Programm terminiert; es beinhaltet eine Endlosschleife.

   Modifizieren Sie das Programm derart, dass zwei Kindprozesse gestartet werden. Der zweite Kindprozess soll eine eigene Funktion erhalten, die (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 Character schreiben) 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 Vaterprozesses abgespeichert. Geben Sie einen Grund an, warum dies gemacht wird.

e) Eine interessante Frage ist nun noch: werden Vater- und Kindprozess gleich behandelt? Geben alle Prozesse aus Aufgabenteil a) ungefähr gleich häufig ihr Zeichen aus? Um dies zu ermitteln, schreiben Sie ein Programm, welches Zeichen von der Standardeingabe (`stdin`) liest und in

regelmäßigen Abständen die Häufigkeiten der einzelnen Buchstaben auf der Standardausgabe ausgibt. Wenden Sie es auf den von Ihnen modifizierten Buchstabengenerator (aus Aufgabenteil a)) an und überprüfen Sie, ob alle Prozesse etwa dieselbe Prozessorzeit erhalten. Geben Sie als Lösung ihren Programmcode ab.

Aufgabe 2.3: Fork (4 + 1 = 5 Punkte)

a) Wir betrachten folgendes C-Programm:

```
1  #include <unistd.h>
2  #include <sys/wait.h>
3
4  int main() {
5      if (fork() == 0) {
6          write(1, "A", 1);
7          if (fork() == 0) {
8              write(1, "B", 1);
9          }else if(wait(NULL) > 0) {
10             write(1, "C", 1);
11         }
12     }else {
13         fork();
14         write(1, "D", 1);
15         if (fork() == 0) {
16             write(1, "E", 1);
17         }else  {
18             write(1, "F", 1);
19         }
20     }
21     return 0;
22 }
```

Zeichne einen Baum, der die Prozessstruktur darstellt. Die *Knoten* des Baums sollen jeweils die `fork`-*Verzweigstellen des Programms* repräsentieren. Verzweige den Baum an diesen Stellen, indem du mit dem *Kind im linken Teilbaum* und mit dem *Vater im rechten Teilbaum* fortfährst. Füge Blätter zum Baum hinzu, jedes Blatt beschreibt so genau einen Prozess im Lebenszyklus des Programms. Schreibe in die *Blätter* des Baums alle *Ausgaben, die* `write` *für diesen Prozess erzeugt hat*.

b) Welche Aussagen kannst Du über die *Reihenfolge der Bildschirmausgaben* treffen?