3.1 Einfache Shell-Simulation (3 Punkte)

Implementieren Sie eine rudimentäre Simulation einer Unix-Shell. Schreiben Sie hierfür eine Funktion `main()`, in der Sie in einer Endlosschleife Benutzereingaben einlesen. Gibt der Benutzer `"exit"` ein, wird die Schleife verlassen und das Programm erfolgreich beendet. Ansonsten wird der vom Benutzer eingegebene Text direkt an das Betriebssystem weitergeleitet.

- Sie benötigen die folgenden Funktionen: `fgets()`, `strncmp()` und `system()`.
- Sollten Sie dynamisch Speicher allokieren, geben Sie diesen am Ende mittels `free()` frei! Bei statischer und automatischer Allokation (z.B. `char userstring[256];`) entfällt dies.
- Beachten Sie, dass Zeichenketten (Strings) in C mit \0 terminiert werden. Eine Zeichenkette benötigt also ein Zeichen mehr Platz, als sie eigentlich beinhaltet. `userstring[256]` kann damit 255 Zeichen aufnehmen.

Quellen: `shellsim.c`

Executable: `shellsim`

3.2 Linked-List (7 Punkte)

Einfach verkettete Listen (Singly Linked Lists) vereinfachen die Verwaltung von unterschiedlichen Datenstrukturen in Betriebssystemkerneln und eignen sich sehr gut zum Einstieg in die C-Programmierung.

Im Folgenden finden Sie eine Beispiel-Verwaltungsstruktur einer Singly Linked List:

```
struct list_elem {
        struct list_elem *next;  // Zeiger auf das naechste Element
        char             *data;  // Zeiger auf ein Datenobjekt
};

typedef struct list {
        struct list_elem *first; // erstes Element in der Liste
        struct list_elem *last;  // letztes Element in der Liste
} list_t;
```

Schreiben Sie ein Modul list.[c|h], eine einfach verkettete Liste erzeugt. Dieses soll die im folgenden aufgelisteten Funktionen bereitstellen.

Implementieren Sie die Datenstruktur der **Singly Linked List**, indem Sie die folgenden Funktionen bereitstellen. Allokieren Sie jedes einzelne Listenelement mit der `malloc`-Funktion, die Sie während der Tutorübung kennen gelernt haben. Sie können die oben gegebenen Funktionssignaturen als C-Datei (Rahmenprogramm) auf Moodle herunterladen.

- `list_t *list_init ();`

  Initialisiert eine neue Liste und liefert einen Zeiger auf die Basisstruktur zurück. Liefert einen Zeiger auf die Liste zurück oder NULL, falls ein Fehler aufgetreten ist.

- `struct list_elem *list_insert (list_t *list, char *data);`

  Erzeugt ein neues Listenelement und fügt es am Beginn der Liste ein. Liefert einen Zeiger auf das neue Listenelement zurück oder NULL, falls ein Fehler aufgetreten ist.

- `struct list_elem *list_append (list_t *list, char *data);`

  Erzeugt ein neues Listenelement und fügt es am Ende der Liste ein. Liefert einen Zeiger auf das neue Listenelement zurück oder NULL, falls ein Fehler aufgetreten ist.

- `int list_remove (list_t *list, struct list_elem *elem);`

  Durchsucht die Liste list von vorne und entfernt das Element list_elem, sofern es gefunden wird. Die Datenstruktur von elem wird ebenfalls freigegeben, die eigentlichen Daten hingegen nicht, denn es ist nicht bekannt, ob diese noch gebraucht werden oder überhaupt dynamisch alloziert wurden. Liefert 0 zurück, wenn das Element gefunden wurde, andernfalls -1.

- `void list_finit (list_t *list);`

  Entfernt alle Element aus der Liste und gibt die Basisdatenstruktur list frei.

- `struct list_elem *list_find (list_t *list, char *data,`
  `int (*cmp_elem) (const char *, const char *));`

  Sucht in der Liste ein Element, das den Daten des angegebenen Elements entsprechend einer anzugebenden Vergleichsfunktion entspricht. Die Funktion soll die Liste von vorne nach hinten durchgehen und das erste Element zurückliefern, dessen Daten dem übergebenen `data` entsprechen. In unserem Fall sind ist der `data`-Teil jedes Elements jeweils eine Zeichenkette (String), so dass als Vergleichfunktion einfach `strcmp()` verwendet werden kann. Die Funktion muss also auf jedes Element der Liste und den übergebenen Parameter `data` jeweils die Vergleichsfunktion `cmp_elem` anwenden. Eine Übereinstimmung liegt vor, wenn die Vergleichsfunktion 0 zurückliefert.

- `void list_print (list_t *list, void (*print_elem) (char *));`

  Geht die Elemente der Liste von vorne bis hinten durch und ruft für jedes Element die als zweiten Parameter übergebene Funktion mit dem Argument data aus der Struktur `struct list_elem` auf. Für die Ausgabe werden die Elemente in aufsteigender Reihefolge (beginnend bei 1) durchnumeriert.

  Für diese Funktion setzen wir eine Besonderheit von C ein: Funktionspointer. Da Ihre Listenfunktionen nicht wissen (können), was für Daten Sie in der Liste ablegen werden, definieren Sie eine eigene Funktion, die die Daten interpretiert und ausgibt und übergeben diese als zweiten Parameter an `list_print ()`. Die Funktion muss folgenden Prototyp haben:

  ```
  void print_string (char *data);
  ```

  Die Ausgabe einer Liste, die die Elemente “erstes”, “zweites” und “drittes” enthält, soll dann wie folgt aussehen (linksbündig ohne Einrückung und Leerzeichen):

  ```
  $ ./lt
  1:erstes
  2:zweites
  3:drittes
  ```

  Zum Testen erzeugen Sie die Liste beispielsweise mit folgendem Code:

```
int main (int argc, char *argv [], char *envp []) {
    list_t      *li;

    if ((li = list_init ()) == NULL) {
        perror ("Cannot allocate memory");
        exit (-1);
    }
    if (list_append (li, "erstes") == NULL ||
        list_append (li, "zweites") == NULL ||
        list_append (li, "drittes") == NULL) {
        perror ("Cannot allocate memory");
        exit (-1);
    }
    list_print (li, print_string);
    exit (0);
}
```

Nach erfolgreichem Testen ersetzen Sie obige main()-Funktion durch eine, die eine Liste erzeugt und diese entsprechend den auf der Kommandozeile angebenen Kommandos füllt bzw. leert. Drei Kommandos sollen unterstützt werden:

- `-a <Wert>` fügt den angebenen Wert mittels `list_append` hinten an die Liste an.
- `-i <Wert>` fügt den angebenen Wert mittels `list_insert` vorne in die Liste ein.
- `-r <Wert>` findet nur das erste Element mit dem angegebenen Wert mittels `list_find` und entfernt nur dieses eine Element mitteln `list_remove` aus der Liste.

Abschließend gibt das Programm die Liste mittels `list_print()` aus.

Beispielablauf:

```
$ ./lt -i Happy -i GBS -i everyone -r GBS -a Now
1:Everyone
2:Happy
3:Now
```

Verwenden Sie eine Header-Datei `list.h` zum Definieren der Datenstrukturen und Funktionsprototypen und dann eine Datei `list.c` für die Implementierung. Erzeugen Sie eine separate Datei `lt.c` für die `main ()`-Funktion.

Quellen: `lt.c list.c list.h`

Executable: `lt`

3.3 Speicherbetrachung (keine Abgabe)

Nutzen Sie die unter 3.2 geschriebene Funktion, um die Datenstrukturen der Liste einzeln auszugeben. Hierzu können Sie nicht die Funktion `list_print()` verwenden, denn diese übergibt eine nur die

Adresse der Daten an die `list_print()`-Funktion, nicht aber die Adresse des jeweiligen Listenelements.

Sie können sich eine ähnliche Funktion `list_dump()` schreiben, die einer übergebenen Funktion einen Pointer auf das jeweiligen Listenelement übergibt. Oder aber Sie iterieren in einer Schleife in Ihrem Hauptprogramm durch die Liste.

Beantworten Sie sich die Frage: Wo liegen die einzelnen Elemente im Speicher?

Ein Beispiel:

```
$ ./lt -i Happy -i GBS -i Everyone -r GBS -a Now
Element 1: Everyone
Adresse: 0x7ffaea500110 Laenge: 16
000000 : f0 00 50 ea fa 7f 00 00 d0 db 74 e2 fe 7f 00 00   ..P.......t.....
Element 2: Happy
Adresse: 0x7ffaea5000f0 Laenge: 16
000000 : 20 01 50 ea fa 7f 00 00 c0 db 74 e2 fe 7f 00 00    .P.......t.....
Element 3: Now
Adresse: 0x7ffaea500120 Laenge: 16
000000 : 00 00 00 00 00 00 00 00 e3 db 74 e2 fe 7f 00 00   ..........t.....
```

3.4 Speicheranalyse mit gdb (keine Abgabe)

Übersetzen Sie Ihr Programm mit dem Compiler-Flag -g. Laden Sie Ihr Programm von 3.3 in den gdb. Setzen Sie einen breakpoint in gdb so, dass das Programm anhält, nachdem Sie ihre Liste aufgebaut haben. Wählen Sie beispielsweise die Zeile, in der Sie das erste Mal `list_print()` aufrufen.

Dann starten Sie das Programm. Untersuchen Sie, ob Sie dort dieselben Informationen über Adressen und Speicherinhalte erzielen, indem Sie die einzelnen Elemente Ihrer Liste Schritt für Schritt betrachten.

Ein Beispiel (Sie werden natürlich eine andere Zeilennummer verwenden müssen und auch potentiell andere Adressen angezeigt bekommen):

```
$ gdb lt
[...]
Reading symbols from lt...done.
done.
(gdb) break 77
Breakpoint 1 at 0xc36: file list.c, line 77.
(gdb) run -a Tutorium -a fuer -a Betriebssysteme
Starting program: /home/student/gbs/uebungsbetrieb/ws18/01/
c-files/lt -a Tutorium -a fuer -a Betriebssysteme

Breakpoint 1, main (argc=4, argv=0x7fffffffe588, envp=0x7fffffffe5b0) at lt.c:77
179         list_print (l1, print_string);
(gdb) print l1
$1 = (list_t *) 0x555555757010
(gdb) print *l1
$2 = {first = 0x5555557570d0, last = 0x555555757050}
(gdb) print l1->first
$3 = (struct list_elem *) 0x5555557570d0
(gdb) print *l1->first
$4 = {next = 0x555555757090, data = 0x7fffffffe7ff "Betriebssysteme"}
(gdb) print *l1->first->next
```

```
$5 = {next = 0x555555757050, data = 0x7fffffffe7fa "fuer"}
(gdb) print *l1->first->next->next
$6 = {next = 0x0, data = 0x7fffffffe7f1 "Tutorium"}
```