182.709 Betriebssysteme UE
1. Test

13. November 2017

Matrikelnummer

Zuname, Vorname

Ges. (60)	1.)(20)	2.)(40)

Allgemeines

Kopieren Sie sich die Angaben des Beispiels mit dem Shellkommando

`$ fetch <1|2>` (z.B. `fetch 1`)

in Ihr Homeverzeichnis, wobei 1 bzw. 2 die Nummer des Beispiels ist, das Sie in Folge bearbeiten möchten. Sie können `fetch` mehrmals ausführen. Eventuelle Änderungen werden gesichert (`<Datei>`→`<Datei>.~1~`).

Um Ihre Lösung zu erstellen, verwenden Sie `make`. Testen und debuggen Sie Ihr Programm wie gewohnt. Auch `gdb` steht Ihnen zur Verfügung.

Um Ihre Lösung zu prüfen, führen Sie

`$ deliver <1|2>` (z.B. `deliver 1`)

aus. Sie können `deliver` beliebig oft ausführen. Sind Sie mit der Bewertung **beider** Beispiele zufrieden, melden Sie sich bei der Aufsicht, die das Ergebnis entgegennimmt und Sie ausloggen wird.

Sie können den Prüfungsbogen für Notizen verwenden. Diese Eintragungen werden nicht bewertet.

Beachten Sie weiters folgende bei beiden Beispielen angewandten Bewertungskriterien:

- Ihr Programm muss ohne Fehler kompilieren, sonst erhalten Sie keine Punkte.
- Ihr Programm muss in jedem Fall ohne Segmentation Fault (Speicherzugriffsverletzung) terminieren, sonst erhalten Sie keine Punkte.
- Compiler Warnings (das Programm wird mit `-Wall` übersetzt) führen zu Punkteabzügen.
- Deaktivieren Sie vor der Bewertung durch `deliver` jegliche Debugausgaben auf `stdout`! Die Bewertung erfolgt unter anderem durch Prüfung der Ausgaben der Programme.

Einleitung

Sie wollen für sich eine Home-Automation Anlage einrichten. Ein Server soll den Zustand der Geräte (z.B.: Alarmanlage, Licht) verwalten. Clients können sich zum Server verbinden und mittels Kommandos den Zustand der Geräte verändern oder abfragen. Geräte können ein- (1) bzw. ausgeschaltet (0) werden. Der Zustand mancher Geräte kann auch auf einen Prozentsatz gesetzt werden (z.B. könnten die Jalousien zu 50% geöffnet werden).

Es liegen zwei getrennte Beispiele vor, die Sie unabhängig voneinander lösen können. Im ersten Beispiel ist die *Argumentebehandlung* des Clients zu entwickeln. Das zweite Beispiel fordert die Implementierung einer Client-Server-Kommunikation über *Sockets* und das Updaten eines Gerätezustands.

1 Argumentebehandlung (20)

Vervollständigen Sie das Programm in `client.c` mit der Argumentebehandlung.

Synopsis

```
./client [-p PORT] {-g|-s VALUE} ID
```

Der Port des Servers wird optional als Argument `PORT` $\in [1024,$ `UINT16_MAX`$]$ der Option `-p` dem Client übergeben. Per Default ist der Port 2017.

Es gibt zwei Kommandos: *GET*) Lesen des Zustands eines Geräts, und *SET*) Setzen des Zustands eines Geräts. *GET* und *SET* kann mit den Optionen `-g` bzw. `-s` ausgewählt werden. Mit dem Argument `VALUE` $\in [0, 127]$ der Option `-s` kann der Zustand eines Geräts auf einen bestimmten Wert gesetzt werden. *Genau eine* der Optionen `-g` und `-s` muss angegeben werden.

Das Argument `ID` $\in [0, 63]$ gibt an auf welches Gerät das Kommando ausgeführt werden soll.

Speichern Sie ID, Kommando, gegebenenfalls den Wert (beim Kommando *SET*) und das Port (als Zahl und String) in die Variable `arguments`.

Im Falle eines fehlerhaften Aufrufs soll das Programm mit `EXIT_FAILURE` terminieren.

Hinweise:

- Es steht Ihnen die Funktion `usage()` zur Verfügung, die Sie im Fehlerfall aufrufen können.
- Zum Parsen von `PORT`, `VALUE` und `ID` steht die Funktion `parse_number()` bereits zur Verfügung.

1.1 Beispiele

```
$ ./client -g 1
port = 2017 (2017), id = 1, GET
$ ./client -g -s 100 2
Usage: ./client [-p PORT] {-g|-s VALUE} ID
$ ./client -s 130 1
Usage: ./client [-p PORT] {-g|-s VALUE} ID
```

1.2 Quellcode

1.2.1 client.h

```
#ifndef _CLIENT_H_
#define _CLIENT_H_

#include <stdint.h>

/** Defines the command type. */
typedef enum {
  GET = 0,
  SET = 1,
  UNDEF = 2
} cmd_t;

/** Structure for the arguments. */
struct args {
  uint16_t portnum;   /**< port number */
  const char *portstr;/**< port number as string */
  cmd_t cmd;          /**< command (GET, SET) */
  uint8_t value;      /**< set value */
  uint8_t id;         /**< device id */
};

/** Prints the usage message and exits with EXIT_FAILURE. */
void usage(void);

/** Parses a string to a long integer. Returns -1 on error. */
long parse_number(const char *str);

/** Applies command according to arguments. */
void apply_command(struct args arguments);

#endif
```

1.2.2 client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <stdbool.h>

#include "client.h"

/** default port number */
#define DEFAULT_PORTNUM     (2017)
#define DEFAULT_PORTNUM_STR "2017"

/** name of the executable (for printing messages) */
char *program_name = "<not yet set>";

int main(int argc, char **argv)
{
    struct args arguments =
        { DEFAULT_PORTNUM, DEFAULT_PORTNUM_STR, UNDEF, 0, 0 };

    /* set program_name */
```

```
    if (argc > 0) {
        program_name = argv[0];
    }

    /*********************************************************************
     * Task 1
     * ------
     * Implement argument parsing for the client. Synopsis:
     *   ./client [-p PORT] {-g|-s VALUE} ID
     *
     * Call usage() if invalid options or arguments are given (there is no
     * need to print a description of the problem).
     *
     * Hints: getopt(3), UINT16_MAX, parse_number (client.h),
     *        struct args (client.h)
     *********************************************************************
        */

    /* COMPLETE AND EXTEND THE FOLLOWING CODE */
    int c;
    int opt_p = 0;
    long num;

    while ((c = getopt(argc, argv, "p:")) != -1) {
        switch (c) {
        case 'p':
            if (opt_p != 0)
                usage();

            num = parse_number(optarg);
            if (num < 1024 || num > UINT16_MAX)
                usage();

            arguments.portnum = (uint16_t) num;
            arguments.portstr = optarg;
            opt_p = 1;
            break;

        }
    }

    /* DO NOT FORGET ABOUT THE POSITIONAL ARGUMENTS */


    /* DO NOT REMOVE THE FOLLOWING LINE */
    apply_command(arguments);

    return 0;
}
```

2 Sockets (40)

Sie sollen in diesem Beispiel die Kommunikation via Sockets im Client implementieren und im Server das Updaten eines Gerätezustands realisieren.

Der Client verbindet sich zum Server und sendet das Kommando zum Ändern oder Abfragen eines Gerätezustands. Der Server antwortet mit einer Bestätigung. Nach der Auswertung der Bestätigung vom Server, terminiert der Client.

Die Optionsbehandlung ist bereits implementiert, wobei alle relevanten Informationen (Serverport, Art der Anfrage, Geräte-ID) in der Variable `arguments` enthalten sind.

Die Aufgabenstellung ist in drei Teilaufgaben gegliedert, die in beliebiger Reihenfolge gelöst werden können. Für alle Tasks stehen Musterlösungen zur Verfügung (`task_1_demo()`, `task_2_demo()`, `task_3_demo()`), die Sie durch Ihre eigene Lösung ersetzen sollen. Das ermöglicht es Ihnen beispielsweise, Task 2 vor Task 1 zu lösen.

Task 1: Verbindungsaufbau (15 Punkte)

Zu bearbeitende Datei: **client.c**

Erzeugen Sie einen TCP Socket (Adressfamilie `AF_INET`, Sockettyp `SOCK_STREAM`) und verbinden Sie sich zum Server. Der Server wird auf *localhost* (127.0.0.1) ausgeführt.

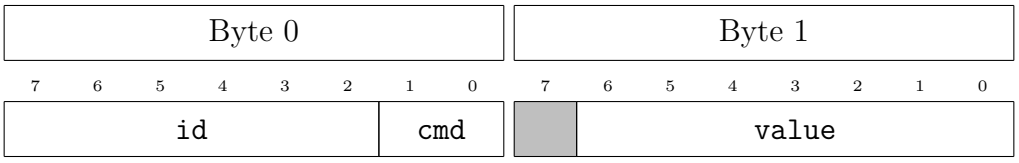
Hinweise:

- Es steht Ihnen die Funktion `error_exit()` zur Verfügung, die Sie im Fehlerfall aufrufen können.
- Für weitere Infos zum IP Protokoll siehe *ip(7)*.

Task 2: Packen, Senden und Empfangen (10 Punkte)

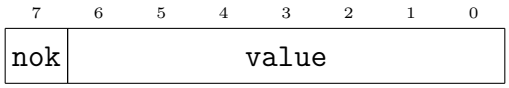
Zu bearbeitende Datei: **client.c**

Senden Sie die ID des Geräts (`id`), das Kommando (`cmd` ∈ {`GET`, `SET`}) und den Wert (`value`), gepackt in 2 Bytes, an den Server.



Byte 1 enthält gegebenenfalls den Wert zum Kommando `SET`. Bit 7 bleibt davon ungenutzt (`value` kann nur Werte von 0 bis 127 annehmen).

Danach empfangen Sie die Bestätigung des Servers bestehend aus einem Byte.



Zerlegen Sie die Antwort und speichern Sie diese in die dafür vorgesehenen Variablen `nok` und `value`. Im Detail soll Bit 7 in die Variable `nok` und Bit 0..6 in die Variable `value` gespeichert werden.

Hinweis:

- Voraussetzung für Task 2 ist ein vorangegangener korrekter Verbindungsaufbau (Task 1). Bearbeiten Sie diesen Task nur, wenn Sie Task 1 korrekt implementiert haben oder mit Verwendung der Musterlösung zu Task 1.

Task 3: Gerätestatus setzen (15)

Zu bearbeitende Datei: **server.c**

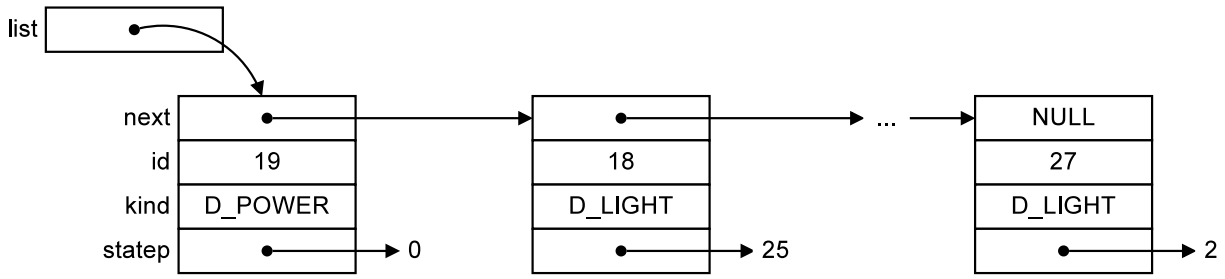
Der Server verwaltet eine Liste der verfügbaren Geräte. Implementieren Sie folgende Funktionalität in der Funktion `update_device_status()` in `server.c`:

- Durchlaufen Sie die Liste der Geräte, bis Sie das Gerät mit der angegebenen ID (Variable `id`) finden.
- Setzen Sie den Gerätezustand auf den angegebenen Wert, sofern sich dieser im gültigen Wertebereich befindet und retournieren Sie `true`. Andernfalls geben Sie `false` zurück und lassen den Gerätestatus unverändert.
- Wird außerdem die ID nicht gefunden, soll auch `false` zurückgeliefert werden.

Hinweise:

- Die Funktion `update_device_status()` wird beim Kommando *GET* nicht aufgerufen. Sie müssen diesen Fall daher nicht behandeln.
- Voraussetzung für Task 3 ist ein korrekter Verbindungsaufbau (Task 1) und eine korrekte Kommunikation mit dem Server (Task 2). Bearbeiten Sie diesen Task nur, wenn Sie Task 1 und 2 korrekt implementiert haben oder mit Verwendung der Musterlösungen von Task 1 und 2.

Geräteliste Die Geräteliste ist als einfach verkettete Liste implementiert. Jedes Listenelement (`device_t`) entspricht einem Gerät und enthält neben dem Zeiger auf das nächste Listenelement (`next`), die Geräte-ID (`id`), die Geräteart (`kind`), und den Zeiger auf den Gerätezustand (`statep`). Der Zustand eines Geräts wird durch Lesen von bzw. Schreiben auf den Zeiger `statep` abgefragt bzw. verändert. Die Geräteliste wird beim Starten des Servers erzeugt und steht wie in der Abbildung skizziert zur Verfügung.



Das Argument `list` der Funktion `update_device_status()` zeigt auf das erste Element der Liste. Jedes Listenelement enthält einen Zeiger (`next`) auf das nächste Element. Das letzte Element schließt die Liste mit `next` gleich `NULL` ab.

Notiz: Die Geräte sind hierbei weder nach Geräte-ID noch nach Geräteart sortiert, sondern stehen in beliebiger Reihenfolge in der Liste. Die IDs sind eindeutig, beginnen jedoch nicht zwangsläufig bei 0.

Geräteart Je nach Geräteart (siehe `server.h`) sind unterschiedliche Wertebereiche gültig. Zum Beispiel, kann ein Gerät der Art `D_POWER` nur ein- (1) oder ausgeschaltet (0) werden. Lampen können auch gedimmt werden, d.h., Geräte der Art `D_LIGHT` können auf Werte von 0 bis 100 gesetzt werden.

In nachfolgender Tabelle sind die gültigen Wertebereiche je nach Geräteart aufgeschlüsselt.

Geräteart (`kind`)	Gültiger Wertebereich
`D_LIGHT`	$[0, \ldots, 100]$
`D_POWER`	$[0, 1]$
`D_SUNBLIND`	$[0, \ldots, 100]$
`D_LOCK`	$[0, 1]$
`D_ALARM`	$[0, 1]$

Synopsis und Beispiele

Zuerst muss der Server gestartet werden, bevor der Client ausgeführt werden kann.

```
$ ./server
./server: Waiting for client.
```

Der Client kann in einem weiteren Terminal gestartet werden.

```
$ ./client
Usage: ./client [-p PORT] {-g|-s VALUE} ID
```

Der Client hat dieselbe Synopsis wie in Beispiel 1. Zum Beispiel kann der Zustand eines Geräts mit ID 24 wie folgt auf 50% gesetzt werden:

```
$ ./client -s 50 24
OK
```

Der aktuelle Zustand des Geräts mit ID 31 wird wie folgt gelesen:

```
$ ./client -g 31
OK
41
```

2.1 Source Code

2.1.1 common.h

```c
#ifndef COMMON_H_
#define COMMON_H_

/** textual representation of the server's IP address */
#define SERVER_IPADDR_STR   "127.0.0.1"

/** default port number */
#define DEFAULT_PORTNUM     (2017)
#define DEFAULT_PORTNUM_STR "2017"

/** size of the request packet in bytes
 * device id + command (1), value (1) */
#define REQUEST_SIZE  (2)

/** size of the response packet in bytes
 * nok + value (1) */
#define RESPONSE_SIZE (1)

/** Defines the command. */
typedef enum {
    GET = 0,
    SET = 1,
    UNDEF
} cmd_t;

/** Prints an error message and exits with EXIT_FAILURE. If errno is not 0,
 * also strerror(errno) is printed. */
void error_exit(const char *msg);

#endif /* COMMON_H_ */
```

2.1.2 client.h

```c
#ifndef _CLIENT_H_
#define _CLIENT_H_

#include <stdint.h>

#include "common.h"

/** Structure for the arguments. */
struct args {
  uint16_t portnum;   /**< port number */
  const char *portstr;/**< port number as string */
  cmd_t cmd;          /**< command (GET, SET) */
  uint8_t value;      /**< set value */
  uint8_t id;         /**< device id */
};

/** Parse program arguments and fill an arguments struct.
 * @param argc  argc as provided to main
 * @param argv  argv as provided to main
 * @param res   pointer to a struct args that should be filled
 */
void parse_arguments(int argc, char **argv, struct args *res);

/* declarations of demo solutions */
void task_1_demo(int *sockfd, struct args *arguments);
void task_2_demo(int *sockfd, struct args *arguments, uint8_t *nok, uint8_t
```

```
    *value);

#endif
```

2.1.3 client.c

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>

#include "client.h"
#include "common.h"

/** name of the executable (for printing messages) */
char *program_name = "client";

/** program entry point */
int main(int argc, char **argv)
{
    struct args arguments;

    /* parse program arguments and fill 'arguments' [given] */
    parse_arguments(argc, argv, &arguments);


    /*********************************************************************
     * Task 1
     * ------
     * Connect to server.
     *
     * - Resolve host address, set port. Macros exist (SERVER_IPADDR_STR)
       and
     *   variables exist (arguments).
     * - Create socket. Use variable 'sockfd' for creation. Socket type
     *   is SOCK_STREAM.
     * - Connect.
     *
     * See also: getaddrinfo(3), socket(2), connect(2), ip(7),
     * error_exit (common.h)
     *********************************************************************/

    /* file descriptor of socket */
    int sockfd;

    /* REPLACE FOLLOWING LINE WITH YOUR SOLUTION */
    task_1_demo(&sockfd, &arguments);


    /*********************************************************************
     * Task 2
     * ------
```

```
     * Pack and send command to server and receive acknowledge.
     *
     * - Pack the command from the arguments into a buffer.
     * - Send the buffer to the server.
     * - Receive response from the server.
     *   Save response to variables 'nok' and 'value' accordingly.
     *
     * See also: send(2), recv(2)
     ************************************************************/

    uint8_t nok;
    uint8_t value;

    /* REPLACE FOLLOWING LINE WITH YOUR SOLUTION */
    task_2_demo(&sockfd, &arguments, &nok, &value);


    /* DO NOT CHANGE THE FOLLOWING LINES */
    /* print server response */
    puts((nok) ? "NOK" : "OK");

    if (arguments.cmd == GET && !nok)
        printf("%d\n", value);

    /* cleanup: close socket */
    close(sockfd);

    exit(EXIT_SUCCESS);
}
```

2.1.4 server.h

```
#ifndef _SERVER_H_
#define _SERVER_H_

#include <stdint.h>

/* kinds of devices */
enum devicekind {
  D_LIGHT = 0,
  D_POWER,
  D_SUNBLIND,
  D_LOCK,
  D_ALARM,
  NUM_DEVICEKIND
};

/* device list element */
struct device {
  struct device *next;
  uint8_t id;
  enum devicekind kind;
  uint8_t *statep;
};

typedef struct device device_t;


#endif /* _SERVER_H_ */
```

2.1.5 server.c

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#include "server.h"
#include "common.h"

/** name of the executable (for printing messages) */
extern const char *program_name;

/* declarations of demo solutions */
bool task_3_demo(device_t *, uint8_t, uint8_t);

bool update_device_status(device_t * list, uint8_t id, uint8_t value)
{
    /*****************************************************************
     * Task 3
     * ------
     * Update device status.
     *
     * - Go through device list until device with given id is found.
     * - Update the status of the device according to the given value.
     * - Return true if the update was sucessful, otherwise false
     *   (device with id not existing, value out of range).
     *****************************************************************/

    /* REPLACE FOLLOWING LINE WITH YOUR SOLUTION */
    return task_3_demo(list, id, value);
}
```