
Technische Informatik III:

Betriebssysteme und Rechnernetze WS 2007/08

Musterlösung zum Übungsblatt Nr. 4

Aufgabe 1: Begriffe

4 Punkte

Beschreiben Sie jeden der folgenden Begriffe durch maximal zwei Sätze: BIOS, Master Boot Record, Cron Job, Protocol Stack.

- BIOS: Basic Input Output System, die Erste Software die beim PC-Start aufgeführt wird. Sie kann bestimmte Hardware (z.B. den Arbeitsspeicher) überprüfen und gibt die Kontrolle über die Hardware an den Bootload des eingestellten Laufwerks weiter.
- Master Boot Record: Erster Sector auf der Festplatte, enthält die Partitionstabelle und eventuell einen Bootloader.
- Cron Job: Sind Programme, die regelmäßig wieder ausgeführt werden. Das Programm *cron* verwaltet unter UNIX diese Programme.
- Protocol Stack: nennt man die Implementierung eines Netzwerkprotokolls.

Aufgabe 2: Protokolle des OSI-Schichtenmodells

8 Punkte

Ordnen Sie die folgenden Protokolle den Schichten des ISO/OSI Modelles zu: Ethernet, SMTP, DNS, SNMP, ICMP, UDP, IP, FTP. Beschreiben Sie stichpunktartig welche Funktionen die Protokolle auf der jeweiligen Schicht erfüllen.

Ethernet: Schicht 1–2

- Schicht 1: Bitübertragung z. B. nach dem *100BASE-TX*-Substandard.
- Schicht 2: Austausch von Datenframes nach dem *IEEE 802* Standard.

SMTP: Simple Mail Transfer Protocol, Schicht 7

- Schicht 7: Arbeitet mit einem geöffneten *TCP*-Socket.

DNS: Domain Name System, Schicht 5

- Schicht 5: Baut eine *UDP*-Verbindung (*TCP* geht auch) mit einem *Name*-Server auf, um dann zu dem angefragten Namen die *IP*-Adresse zu erfahren.

SNMP: Simple Network Management Protocol, Schicht 7

- Schicht 7: Baut ebenfalls *UDP*-Verbindung (*TCP* geht auch) mit einem anderem Netzwerkgerät auf, um Wartungsfunktionen zu realisieren.

ICMP: Internet Control Message Protocol, Schicht 3

- Schicht 3: *ICMP* ist ein Teil des *Internet Protokolls*, um Wartungsfunktionen mit Internetgateways zu realisieren.

UDP: User Datagram Protocol, Schicht 4

- Schicht 4: Es realisiert verbindungslose Paketübertragungen von Daten einer höheren Schicht und benutzt selber das *Internet Protokoll* für die Kommunikation.

IP: Internet-Protokoll, Schicht 3

- Schicht 3: Es realisiert den Datenaustausch zwischen zwei Netzwerkteilnehmern über das lokale Netzwerk hinaus. Es benutzt Ethernet als darunterliegende Schicht.

FTP: File Transfer Protocol, Schicht 7

- Schicht 7: Realisiert Datei-Transfer über *TCP*.

Aufgabe 3: Designprinzipien des Internets

8 Punkte

Nennen Sie die vier Designprinzipien des Internets. Beschreiben Sie für jedes Prinzip in ein, zwei Sätzen welche Probleme auftreten könnten, wenn man sich nicht an es halten würde.

- Minimalistisch und Autonom.

Wenn diese Prinzip nicht eingehalten würde, müßte man bei jeder Netzwerkänderung von einem eventuell etwas an der Implementierung des Netzwerk bei anderen ändern. Das ist schwer zu realisieren da sich das Netzwerk ja ständig ändert. Dann müßte man Arbeit aufwenden, um Funktionen erhalten zu können.

- „Nach bester Möglichkeit“

Das Gegenteil wären garantierte Datenverbindungen, die aufgrund der dezentralen Natur des Internets nicht möglich sind.

- Zustandslos

Wenn Router sich den Zustand von einer Verbindung merken müßten, wäre das ein umheimlicher Mehraufwand, da ein Router in der Regel sehr viele Verbindungen bearbeitet.

- Dezentrale Kontrolle.

Gäbe es eine zentrale Instanz die das Internet kontrollieren würde, kann sie je nach politischer Stimmung Teilnehmer ausschließen bzw. Zensur betreiben. Außerdem ist die dezentrale Organisation garant dafür, dass das Internet auch dann funktioniert, wenn ein wichtiger Knotenpunkt ausfällt.

Aufgabe 4: Systemdienste

4 Punkte

Erstellen Sie für ein gängiges Betriebssystem Ihrer Wahl eine Liste von vier typischen Systemdiensten und beschreiben Sie die jeweilige Aufgabe des Dienstes. Welche dieser Dienste lassen sich ohne Beeinträchtigung des Gesamtsystems deaktivieren? Welches Fehlverhalten beobachten Sie bei Deaktivierung der übrigen Dienste?

- Druckerserver:

Der Druckerserver reicht Druckaufträge an den Drucker weiter. Wenn dieser Dienst nicht funktioniert, kann man eventuell nicht drucken, ein Fehlverhalten hat keine Beeinträchtigung auf das Gesamtsystem.

→ Kann deaktiviert werden.

- Prozess-Scheduler:

Der Prozess-Scheduler ist Teil des Prozessmanagement des Betriebssystem, der den einzelnen Prozessen Rechenzeit zuteilt. Zeigt er Fehlverhalten können die Prozesse nicht ordentlich abgearbeitet werden und das Gesamtsystem kann inkonsistent werden.

→ Darf nicht deaktiviert werden.

- login-Prozess:

Der Prozess zum Einloggen in das System wird auch als Systemdienst angesehen. Wenn dieser Fehlverhalten aufweist, könnte ein unauthorisierter Benutzer *root*-Rechte bekommen oder sich einfach keiner Einloggen und der Computer ist nicht benutzbar.

→ Darf nicht deaktiviert werden.

- Protokoll-Dienst:

Dieser Prozess hält das Systemverhalten und -ereignisse in Protokolldateien fest. Wenn dieser Dienst nicht funktioniert, dann stellt das keine Beeinträchtigung des Gesamtsystems. Nur die Fehlersuche bzw. Fehlerfeststellung gestaltet sich schwieriger.

→ Kann deaktiviert werden.

Aufgabe 5: Fifty/Fifty 6 Punkte

Entscheiden Sie, ob folgenden Aussagen zutreffend oder nicht zutreffend sind, und begründen Sie Ihre Entscheidung:

- **Ein Druckerdämon ist ein Virus, der sich auf das Stören der Funktionsweise eines Druckers spezialisiert hat.**

Falsch, ein Druckerdämon oder Druckerserver oder auch Spooler ist ein Systemdienst, der Druckaufträge an den Drucker weiter reicht.

- **Fehler in der Implementierung von Systemdiensten sind in ihren sicherheitsrelevanten Auswirkungen auf den jeweiligen Systemdienst begrenzt.**

Richtig, da die Systemdienste in der Regel nur Benutzerrechte haben und keine root-Rechte, womit die in den anderen sicherheitsrelevanten Teilen Zugriffsrechte hätten.

- Nach dem Socketaufbau ist es für die durch den Socket kommunizierenden Prozesse egal, ob sie auf dem selben oder auf unterschiedlichen Rechnern laufen.

Richtig, da der Socket die Kommunikation der Prozesse abstrahiert und das Netzwerk nach Verbindungsaufbau so verdeckt.

Aufgabe 6: Dateioperationen

8 Punkte

Implementieren Sie einfache Varianten der Befehle *cp* und *mv*, die jeweils zwei Dateinamen als Parameter erhalten und unter dem zweiten Namen eine Kopie der ersten Datei anlegen bzw. die erste Datei in die zweite umbenennt. Beachten Sie dabei, dass erster und zweiter Parameter unterschiedliche Pfade im Dateisystem enthalten können.

Testlauf:

```
$ ls -l
insgesamt 24
-rwxr-xr-x 1 chris chris 11259 2007-12-05 13:57 cp
-rwxr-xr-x 1 chris chris 6840 2007-12-05 13:57 mv
-rw-r--r-- 1 chris chris 97 2007-12-05 13:59 test.txt
$ cat test.txt
Datei zum Testen von Datei operationen
```

Wenn sie das hier lesen können war der Test erfolgreich.

```
$ ./cp test.txt test2.txt
$ cat test2.txt
Datei zum Testen von Datei operationen
```

Wenn sie das hier lesen können war der Test erfolgreich.

```
$ ./mv test2.txt test3.txt
$ cat test3.txt
Datei zum Testen von Datei operationen
```

Wenn sie das hier lesen können war der Test erfolgreich.

```
$ ls -l
insgesamt 28
-rwxr-xr-x 1 chris chris 11259 2007-12-05 13:57 cp
-rwxr-xr-x 1 chris chris 6840 2007-12-05 13:57 mv
-rw-r--r-- 1 chris chris 97 2007-12-05 14:01 test3.txt
-rw-r--r-- 1 chris chris 97 2007-12-05 13:59 test.txt
$
```

- *cp*:

cp.c

```
1  /**
2  * @file cp.c
3  *
4  * TI III: Betriebssysteme und Rechnernetze
5  * WS 2007/08
6  * Übungsblatt Nr. 4,
7  * Aufgabe 6: Dateioperationen
8  *
9  * cp
10 * Kopiert eine Datei
11 *
12 * @author Christian Grümmel
13 * @see http://cst.mi.fu-berlin.de/teaching/WS0708/19513-V-TI-III/index.html
14 * @date 2007-11-18
15 */
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>
19 #define MAX 512 /* Blocklänge */
20
```

```

21  /** @fn int check_file(char * path)
22  * Diese Funktion überprüft, ob die Datei path existiert
23  * oder ob sie überschrieben werden darf.
24  * Sie gibt eine 1 zurück, falls die Datei überschrieben
25  * werden darf, sonst 0.
26  *
27  * @param path Datei zum Überschrieben
28  * @return 0 falls die Datei nicht überschrieben werden darf, sonst 1
29  */
30  int check_file(char * path)
31  {
32      FILE *tmp; /* Temporärer file-descriptor */
33      char buffer[MAX];
34
35      tmp = fopen(path, "r"); /* Öffne Datei zum lesen */
36
37      if(tmp == NULL)
38      {
39          /* Datei existiert nicht */
40          return 1;
41      }
42
43      /* Frage user */
44      do {
45          printf("File %s already exists. Overwrite? (y/n)\n", path);
46          fgets(buffer, MAX - 1, stdin);
47      } while(buffer[0] != 'y' && buffer[0] != 'Y' &&
48              buffer[0] != 'n' && buffer[0] != 'N');
49
50      fclose(tmp); /* Schließe file-descriptor */
51
52      if(buffer[0] == 'y' || buffer[0] == 'Y')
53      {
54          /* Überschreiben erwünscht */
55          return 1;
56      }
57      else
58      {
59          /* Überschreiben nicht erwünscht */
60          return 0;
61      }
62  }
63
64  /** @fn int copy_file(char* src_path, char* dest_path)
65  * Diese Funktion liest die Datei src_path und schreibt sie
66  * nach dest_path. Bei einem Fehler gibt sie EXIT_FAILURE zurück,
67  * sonst EXIT_SUCCESS.
68  *
69  * @param src_path Datei, die gelesen wird
70  * @param dest_path Datei, die geschrieben wird
71  * @return EXIT_SUCCESS, bei Fehler EXIT_FAILURE
72  */
73  int copy_file(char* src_path, char* dest_path)
74  {
75      FILE *src; /* file-descriptor für den Input */
76      FILE *dest; /* file-descriptor für den Output */
77      size_t bytes_read; /* Anzahl gelesener Bytes */
78      size_t bytes_wrote; /* Anzahl geschriebener Bytes */
79      short error = 0; /* Zum Speichern, ob es einen Fehler gab */
80      char buffer[MAX]; /* Lesebuffer */
81
82      /* Versuche 1. Argument als Datei zum binären Lesen zu öffnen */
83      src = fopen(src_path, "rb");
84
85      /* Überprüfe, ob das Öffnen der Datei erfolgreich war */
86      if(src == NULL)
87      {
88          /* Fehlerausgabe auf 'stderr' */
89          fprintf(stderr, "Error: Cannot read file %s\n", src_path);
90
91          return EXIT_FAILURE;
92      }
93
94      /* Überprüfe, ob die zu schreibene Datei bereits existiert */
95      if(check_file(dest_path) == 0)
96      {
97          fclose(src); /* Schließe file-descriptor */
98
99          return EXIT_SUCCESS;
100      }
101
102      /* Versuche 2. Argument als Datei zum schreiben zu öffnen */

```

```

103 dest = fopen(dest_path, "wb");
104
105 /* Überprüfe, ob das Öffnen der Datei erfolgreich war */
106 if(dest == NULL)
107 {
108     /* Fehlerausgabe auf 'stderr' */
109     fprintf(stderr, "Error: Cannot write file %s\n", dest_path);
110
111     return EXIT_FAILURE;
112 }
113
114 /* Lese Blockweise von src bis 0 Bytes gelesen werden */
115 do {
116     bytes_read = fread(buffer, 1, MAX, src);
117     /* Schreibe Blockweise nach dest */
118     bytes_wrote = fwrite(buffer, 1, bytes_read, dest);
119
120     /* Überprüft, ob ein Fehler beim Schreiben auftrat */
121     if(bytes_wrote != bytes_read)
122     {
123         fprintf(stderr, "Error while writing %s\n", dest_path);
124         error = 1;
125     }
126 }
127 while(bytes_read > 0 && error == 0);
128
129 /* Überprüft, ob ein Fehler beim Lesen auftrat */
130 if(ferror(src) != 0)
131 {
132     fprintf(stderr, "Error while reading %s\n", src_path);
133     error = 1;
134 }
135
136 fclose(dest); /* Schließe file-descriptor */
137 fclose(src); /* Schließe file-descriptor */
138
139 if(error == 1)
140 {
141     /* Lösche geschriebene Datei */
142     if(remove(dest_path) != 0)
143     {
144         fprintf(stderr, "Error: Cannot delete %s", dest_path);
145     }
146
147     return EXIT_FAILURE;
148 }
149 else
150 {
151     return EXIT_SUCCESS;
152 }
153 }
154
155 /** @fn int main(int argc, char *argv[])
156  * Main-Funktion, Einstieg in das Programm.
157  *
158  * @param argc Anzahl der Argumente
159  * @param argv 1. Argument wird als Datei eingelesen, 2. als Datei geschrieben
160  * @return Status der Terminierung
161  */
162 int main(int argc, char *argv[]) {
163     /* Überprüfe, ob ein ein Argument übergeben wurde */
164     if(argc == 3)
165     {
166         return copy_file(argv[1], argv[2]);
167     }
168     else
169     {
170         /* Zu wenig/zu viele Argumente übergeben, beende */
171         printf("Usage: %s src dest\n", argv[0]);
172
173         return EXIT_SUCCESS;
174     }
175 }

```

- mv:

mv.c

```

1 /**
2  * @file mv.c
3  *

```

```

4  * TI III: Betriebssysteme und Rechnernetze
5  * WS 2007/08
6  * Übungsblatt Nr. 4,
7  * Aufgabe 6: Dateioperationen
8  *
9  * mv
10 * Benennt eine Datei um
11 *
12 * @author Christian Grümme
13 * @see http://cst.mi.fu-berlin.de/teaching/WS0708/19513-V-TI-III/index.html
14 * @date 2007-11-18
15 */
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>
19 #define MAX 512    /* Blocklänge */
20
21
22
23 /** @fn int main(int argc, char *argv[])
24  * Main-Funktion, Einstieg in das Programm.
25  *
26  * @param argc Anzahl der Argumente
27  * @param argv 1. Argument wird als Datei zum 2. Argument umbenannt
28  * @return Status der Terminierung
29  */
30 int main(int argc, char *argv[]) {
31     /* Überprüfe, ob ein ein Argument übergeben wurde */
32     if(argc == 3)
33     {
34         if(rename(argv[1], argv[2]) != 0)
35         {
36             fprintf(stderr, "Error renaming file %s\n", argv[1]);
37
38             return EXIT_FAILURE;
39         }
40     }
41     else
42     {
43         /* Zu wenig/zu viele Argumente übergeben, beende */
44         printf("Usage: %s src dest\n", argv[0]);
45     }
46
47     return EXIT_SUCCESS;
48 }

```

- Alternativ kann man *mv* analog zu *cp* implementieren:

mv2.c

```

1
2 ...
3
4 /** @fn int main(int argc, char *argv[])
5  * Main-Funktion, Einstieg in das Programm.
6  *
7  * @param argc Anzahl der Argumente
8  * @param argv 1. Argument wird als Datei zum 2. Argument verschoben
9  * @return Status der Terminierung
10 */
11 int main(int argc, char *argv[]) {
12     /* Überprüfe, ob ein ein Argument übergeben wurde */
13     if(argc == 3)
14     {
15         if(copy_file(argv[1], argv[2]) == EXIT_SUCCESS)
16         {
17             /* Lösche gelesene Datei */
18             if((remove(argv[1])) != 0)
19             {
20                 fprintf(stderr, "Error: Cannot delete %s", argv[1]);
21
22                 return EXIT_FAILURE;
23             }
24         }
25     }
26     else
27     {
28         /* Zu wenig/zu viele Argumente übergeben, beende */
29         printf("Usage: %s src dest\n", argv[0]);
30     }
31 }

```

```
32 |     return EXIT_SUCCESS;
33 | }
```

Aufgabe 7: Speicherverwaltung

16 Punkte Punkte

Implementieren Sie das Programm `get_string`, das als Parameter einen Hostnamen und eine Portnummer erhält und dort von einem laufenden Server einen String abholt und ausgibt.

Testlauf:

```
$ ./get_string paste.mi.fu-berlin.de 22
SSH-2.0-OpenSSH_3.8.1p1 Debian-krb5 3.8.1p1-7sarge1
```

get_string.c

```
1  /**
2  * @file get_string.c
3  *
4  * TI III: Betriebssysteme und Rechnernetze
5  * WS 2007/08
6  * Übungsblatt Nr. 4,
7  * Aufgabe 7: Sockets
8  *
9  * get_string
10 *
11 * Dieses Programm verbindet sich mit dem übergebenen Server über
12 * den übergebenen Port und gibt aus, was der Server mitteilt.
13 *
14 * @author Christian Grümme
15 * @see http://cst.mi.fu-berlin.de/teaching/WS0708/19513-V-TI-III/index.html
16 * @date 2007-12-04
17 */
18 #include <stdlib.h>
19 #include <stdio.h>
20 #include <unistd.h> /* Für close(socket) */
21 #include <sys/types.h> /* Allgemeine Systemtypen z.B. ssize_t */
22 #include <sys/socket.h> /* Für die Sockets */
23 #include <netdb.h> /* Für gethostbyname() */
24 #include <netinet/in.h> /* Für Strukturen z.B. sockaddr_in */
25 #include <arpa/inet.h> /* Funktionen mit Network Byte Order */
26
27 #define BUFFER_SIZE 1024 /* Größe des Socketpuffers */
28
29 /* Rückgabewerte von my_connect */
30 #define CONNECTED 0
31 #define FAILURE_HOST 1
32 #define FAILURE_CONNECT 2
33
34 /** @fn int my_connect(int sock, char* url, char* port)
35 * Verbindet übergebenen Socket mit der übergebener Url
36 * und dem übergebenen Port. Wenn alles funktioniert hat,
37 * gibt diese Funktion die Konstante CONNECTED zurück,
38 * ansonsten FAILURE_HOST, falls das Auflösen der Adresse
39 * nicht funktioniert oder FAILURE_CONNECT, falls das
40 * Verbinden gescheitert ist.
41 *
42 * @param sock Anzahl der Argumente
43 * @param url Der zu verbindende Host
44 * @param port Der zu verbindende Port
45 * @return Fehlercode
46 */
47 int my_connect(int sock, char* url, char* port)
48 {
49     int is_connected; /* Ob Socket verbunden */
50     struct sockaddr_in server; /* Komplette Serveradresse */
51     struct hostent *host; /* Für die Auflösung von DNS */
52     struct in_addr address; /* IP */
53
54     /* Löse Namen in IP auf */
55     host = gethostbyname(url);
56     if (host == NULL)
57     {
58         return FAILURE_HOST;
59     }
60
61     /* Nehme IP aus dem DNS */
62     address = *(struct in_addr*) host->h_addr;
63     /* Setze IP für den Server */
```

```

64     server.sin_addr = address;
65     /* Konvertiere zweites Argument als Port nach Network byte order */
66     server.sin_port = htons( (unsigned short int) atol(port));
67     /* Setze Protokollfamilie */
68     server.sin_family = AF_INET;
69
70     /* Verbinde mit dem Server */
71     is_connected = connect(sock, (struct sockaddr*) &server, sizeof(server));
72
73     if(is_connected == -1)
74     {
75         return FAILURE_CONNECT;
76     }
77     else
78     {
79         return CONNECTED;
80     }
81 }
82
83 /** @fn int my_receive(int sock)
84  * Diese Funktion empfängt Daten von einem bereitsverbundenen Socket
85  * und gibt diese aus.
86  *
87  * @param sock Der verbundene Socket
88  * @return -1 bei einem Fehler, sonst 0
89  */
90 int my_receive(int sock)
91 {
92     char answer[BUFFER_SIZE]; /* Puffer für die Antwort */
93     ssize_t bytes;             /* Anzahl empfangener Bytes */
94
95     /* Empfange */
96     bytes = recv(sock, answer, sizeof(answer) - 1, 0);
97     if(bytes == -1)
98     {
99         return -1;
100    }
101
102    /* Terminiere String */
103    answer[bytes] = '\0';
104    /* Antwortausgabe */
105    printf("%s", answer);
106
107    return 0;
108 }
109
110 /** @fn int main(int argc, char *argv[])
111  * Main-Funktion, Einstieg in das Programm.
112  *
113  * @param argc Anzahl der Argumente
114  * @param argv 1. Argument wird als Host, 2. als Port eingelesen
115  * @return Status der Terminierung
116  */
117 int main(int argc, char *argv[])
118 {
119     int sock;           /* Der Socket */
120     int is_connected;
121
122
123     /* Überprüfe richtige Anzahl der Argumente */
124     if(argc != 3)
125     {
126         fprintf(stderr, "usage: %s host port\n", argv[0]);
127         return EXIT_SUCCESS;
128     }
129
130     /* Initialisiere Socket */
131     sock = socket(AF_INET, SOCK_STREAM, 0);
132     if(sock == -1)
133     {
134         fprintf(stderr, "Error: Cannot create socket\n");
135
136         return EXIT_FAILURE;
137     }
138
139     /* Verbinde */
140     is_connected = my_connect(sock, argv[1], argv[2]);
141     /* Fehlerbehandlung */
142     if(is_connected == FAILURE_HOST)
143     {
144         fprintf(stderr, "Error: Cannot resolve host %s\n", argv[1]);
145

```



```
146     return EXIT_FAILURE;
147 }
148 else if(is_connected == FAILURE_CONNECT)
149 {
150     fprintf(stderr, "Error: Cannot connect to %s",argv[1]);
151
152     return EXIT_FAILURE;
153 }
154
155 if(my_receive(sock) == -1)
156 {
157     fprintf(stderr, "Error: Faliure receiving\n");
158
159     return EXIT_FAILURE;
160 }
161
162 /* SchlieÙe Socket */
163 close(sock);
164
165 return EXIT_SUCCESS;
166 }
```