
Technische Informatik III:
Betriebssysteme und Rechnernetze WS 2007/08
Musterlösung zum Übungsblatt Nr. 5

Aufgabe 1: Begriffe

6 Punkte

Beschreiben Sie jeden der folgenden Begriffe durch maximal zwei Sätze: Bit, Signal, Multiplexing, Framing, Header, Piggybacking

- Bit: Bedeutet Binäre Ziffer und enthält die Information 0 oder 1.
- Signal: Ein elektrisches Signal ist die Änderung einer elektrischen Größe (Stärke oder Spannung) über die Zeit.
- Multiplexing: Wird verwendet um mehrere Signale gebündelt zu übertragen. Es gibt verschiedene Multiplexverfahren, die man auch teilweise kombiniert einsetzen kann.
- Framing: Bezeichnet das Einschließen von Daten in einen Rahmen, damit der Empfänger weiß, welche Daten relevant sind. Ein Rahmen ist eine ausgezeichnetes Signal (oder Bitfolge) als Einleitung und ein Signal als Ende.
- Header: Steht am Anfang eines Rahmen und enthält Metainformationen über diesen.
- Piggybacking: Nennt man das Verfahrenen, wenn ein Host b ein Datenpaket von Host a empfangen hat und das zugehörige ACK an ein Datenpaket an a hinzufügt.

Aufgabe 2: Kodierungsverfahren

8 Punkte

- a) Berechnen Sie die Frame Check Sequence des Generatorpolynoms $x^2 + 1$ für die Bitfolge 1111001. Fügen Sie diese an die Bitfolge an und kodieren sie die resultierende Bitfolge mit der Manchesterkodierung. Vervollständigen Sie die Bitfolge mit durch Framing mit dem Frame 111. Ist Bitstuffing nötig um die Einzigartigkeit des Frames zu gewährleisten?

$$x^2 + 1 \hat{=} 101$$

Füge zwei 0-en an die Bitfolge.

$$\begin{array}{r} 111100100:101=1100001 \\ 101 \\ \hline 101 \\ 101 \\ \hline 000100 \\ 101 \\ \hline 1 \end{array}$$

Da raus resultiert ein zu sendender String von 111100101. Fügen wir jetzt noch den Frame von 111 vorne und 111 hinten an, ergibt das 111111100101111.

Da nun die Einzigartigkeit des Frames nicht mehr gilt muss man Bitstuffing vornehmen, d. h. zum Beispiel nach jeder 1 in den Daten wird eine 0 hinzugefügt, die bei Empfangen gelöscht wird, also 111101010100010010111.

Manchesterkodierung:

1 1 1 1 0 1 0 1 0 1 0 0 0 1 0 0 1 0 1 1 1
101 0 1 0 1 0 0 1 1 0 0 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 1 0 1 0

Ergebnis:

101010100110011001100101011001011001101010

- b) Die CRC-gesicherte Bitfolge 1001110001010110 wurde empfangen. Ist bei der Übertragung ein Fehler aufgetreten, wenn das Generatorpolynoms $x^2 + 1$ verwendet wurde?

$$x^2 + 1 \hat{=} 101$$

$$\begin{array}{r}
 1001110001010110:101=10010000010001 \\
 101 \\
 \hline
 111 \\
 101 \\
 \hline
 101 \\
 101 \\
 \hline
 0000101 \\
 101 \\
 \hline
 00110 \\
 101 \\
 \hline
 11
 \end{array}$$

Da der Rest $11 \neq 0$ beträgt, liegt ein Fehler in der Übertragung vor.

Aufgabe 3: Datenrate

2 Punkte

Ein Kanal habe eine Bandbreite von 5 MHz. Wie viele Bit/s können gesendet werden, wenn digitale Signale mit 6 Levels verwendet werden (nehmen sie einen rauschlosen Kanal an)?

Für einen rauschlosen Kanal benutzt man die Formel von Nyquist Max. Datenrate $= 2 \cdot H \log_2 V$ bit/s mit V die Anzahl der verschiedenen Symbole (Levels) und H der verfügbaren Bandbreite in Hz. Also

$$\begin{aligned}
 \text{Max. Datenrate} &= 2 \cdot H \log_2 V \text{ bit/s} \\
 &= 2 \cdot 5.000.000 \log_2 6 \text{ bit/s} \\
 &\approx 10.000.000 \cdot 2,585 \text{ bit/s} \\
 &= 25,85 \text{ Mbit/s} \\
 (&\approx 3,23 \text{ Mbyte/s})
 \end{aligned}$$

Aufgabe 4: Fifty/Fifty **6 Punkte** Entscheiden Sie, ob folgenden Aussagen zutreffend oder nicht zutreffend sind, und begründen Sie Ihre Entscheidung:

- **UDP Pakete werden aufgrund des nicht benötigten Verbindungsaufbaus bevorzugt für Multimediaanwendungen verwendet.**

Richtig, da bei manchen Anwendungen (z.B. VoIP) der Geschwindigkeitsunterschied zwischen UDP und TCP einen Unterschied macht und man in der Regel bei Audio- und Videoübertragungen auch einzelne Paketverluste verschmerzen kann.

- **Nach dem Theorem von Nyquist können unbegrenzte Datenraten in der Praxis erreicht werden.**

Richtig, da es nach Nyquist in der Praxis kein Leitungsrauschen gibt.

- **Um die maximal mögliche Datenrate zu ermitteln, muß man das Maximum der nach den Formel von Nyquist und Shannon berechneten Werte nehmen.**

Falsch, die Formel von Shannon ermittelt die praktische maximale Datenrate, außerdem verwenden beide Formeln unterschiedliche Modelle, so dass sie eigentlich nicht direkt vergleichbar sind.

Aufgabe 5: Scheduler

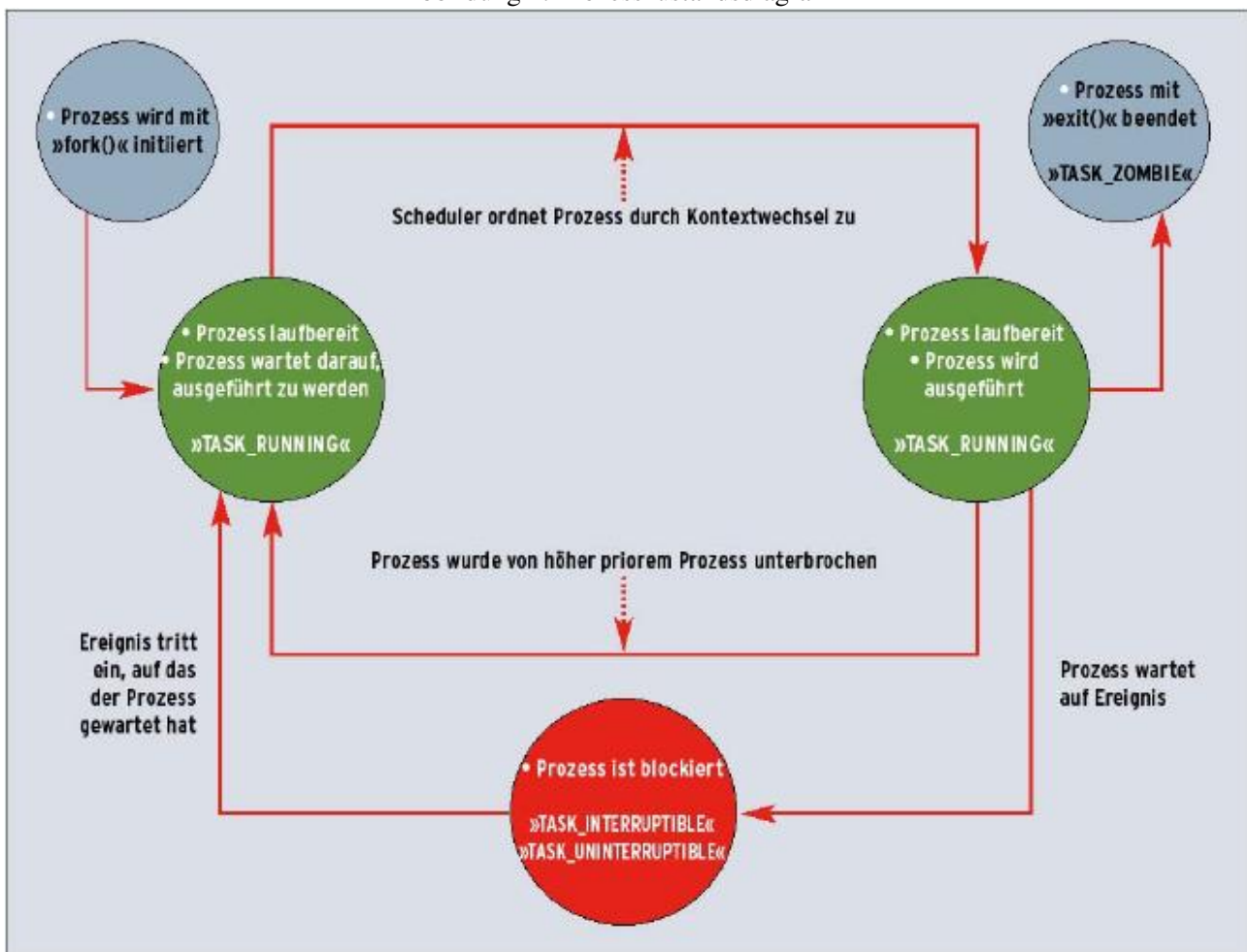
12 Punkte

Betrachten Sie die Implementierung des Schedulers des Linux-Kernels 2.6.18 (im Quelltext zum Download erhältlich unter <http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.1.tar.bz2>), konkret die Dateien `kernel/sched.c` und `include/linux/sched.h`. Zusätzliche Dokumentation ist auf der Homepage der Veranstaltung verlinkt.

Erstellen Sie analog zu den in der Vorlesung zum Thema „Prozesse“ besprochenen Diagrammen ein Prozesszustandsdiagramm für diese konkrete Implementierung eines Schedulers. Vernachlässigen Sie hierbei die für das Auslagern von Prozessen verwendeten Zustände. Geben Sie stattdessen an, wo im Quelltext (Funktionsname, Zeilennummer) welche Zustandsübergänge implementiert sind.

Erstellen Sie wiederum analog zu den in der Vorlesung zum Thema „Scheduling“ besprochenen Diagrammen ein weiteres Diagramm, das die verwendeten Warteschlangen und sonstigen Datenstrukturen des Schedulers darstellt. Verwenden Sie zur Beschriftung sowohl die entsprechenden Namen der Variablen aus dem Quelltext als auch kurze Kommentare zur Funktion innerhalb des Schedulers.

Abbildung 1: Prozesszustandsdiagramm



Die Prozesszustände sind in der `include/linux/sched.h` ab Zeile 141 definiert.

- `TASK_RUNNING` kennzeichnet den Prozess als lauffähig. Er muss auf kein Ereignis warten und kann daher vom Scheduler der CPU zugeordnet werden. Alle Prozesse im Zustand `TASK_RUNNING` zieht der Scheduler für die Ausführung in Betracht.
- Ein blockierter Prozess befindet sich im Zustand `TASK_INTERRUPTIBLE`, da er auf ein Ereignis wartet und vor dessen Eintreten nicht ablaufen kann. Ein Prozess im Zustand `TASK_INTERRUPTIBLE` wird über zwei unterschiedliche Wege in den Zustand `TASK_RUNNING` versetzt: Entweder tritt das Ereignis ein, auf das er gewartet hat, oder der Prozess wird durch ein Signal aufgeweckt.
- `TASK_UNINTERRUPTIBLE` gleicht dem Zustand `TASK_INTERRUPTIBLE`, mit dem Unterschied, dass ein Signal den Prozess nicht aufwecken kann. Der Zustand `TASK_UNINTERRUPTIBLE` wird nur verwendet, wenn zu erwarten ist, dass das Ereignis, auf das der Prozess wartet, zügig eintritt, oder wenn der Prozess ohne Unterbrechung warten soll.

- Wurde ein Prozess beendet, dessen Elternprozess noch nicht den Systemaufruf *wait4()* ausgeführt hat, verbleibt er im Zustand **TASK_ZOMBIE**. So kann auch nach dem Beenden eines Kindprozesses der Elternprozess noch auf seine Daten zugreifen. Nachdem der Elternprozess *wait4()* aufgerufen hat, wird der Kindprozess endgültig beendet, seine Datenstrukturen werden gelöscht. Endet ein Elternprozess vor seinen Kindprozessen, bekommt jedes Kind einen neuen Elternprozess zugeordnet. Dieser ist nunmehr dafür verantwortlich, *wait4()* aufzurufen, sobald der Kindprozess beendet wird. Ansonsten könnten die Kindprozesse den Zustand **TASK_ZOMBIE** nicht verlassen und würden als Leichen im Hauptspeicher zurückbleiben.
- Den Zustand **TASK_STOPPED** erreicht ein Prozess, wenn er beendet wurde und nicht weiter ausführbar ist. In diesen Zustand tritt der Prozess ein, sobald er eines der Signale **SIGSTOP**, **SIGTST**, **SIGTTIN** oder **SIGTTOU** erhält.

TASK_RUNNING (CPU) entspricht hierbei dem rechten **TASK_RUNNING** Zustand, der Prozess wird auf der CPU ausgeführt. **TASK_RUNNING** (links) bedeutet, dass der Prozess laufen könnte, sich jedoch nicht auf einer CPU befindet.

Tabelle 1: Funktionen mit Zustandsübergängen

Von	Zustände	Funktion	Datei	Ort	Zeile
init	TASK_RUN.	do_fork()	kernel/fork.c		1338
TASK_RUN.	TASK_RUN. (CPU)	schedule()	kernel/sched.c		3294
TASK_RUN. (CPU)	TASK_ZOMBIE	do_exit()	kernel/exit.c		845
TASK_RUN. (CPU)	TASK_INT.	schedule()	kernel/sched.c		3294
		interruptible_sleep_on()	kernel/sched.c		3795
TASK_RUN. (CPU)	TASK_UNINT.	schedule()	kernel/sched.c		3294
		sleep_on()	kernel/sched.c		3822
TASK_INT.	TASK_RUN.	wake_up_interruptible()	include/linux/wait.h		149
TASK_UNINT.	TASK_RUN.	wake_up()	include/linux/wait.h		146

Jeder Prozess besitzt einen Prozessdeskriptor. Der Deskriptor hält alle Informationen seines Prozesses fest: Prozessidentifikator (PID), Adressraum, Prozessstatus, Priorität und Signalhandler. Definition in *include/linux/sched.h* ab Zeile 767.

include/linux/sched.h

```

1 struct task_struct {
2     /* Status */
3     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
4
5     [...]
6     /* Verschiedene Prioritäten */
7     int prio, static_prio, normal_prio;
8     struct list_head run_list;
9     struct prio_array *array;
10
11     unsigned short ioprio;
12     unsigned int btrace_seq;
13
14     unsigned long sleep_avg;
15     unsigned long long timestamp, last_ran;
16     unsigned long long sched_time; /* sched_clock time spent running */
17     enum sleep_type sleep_type;
18     /* Art des Scheduling */
19     unsigned long policy;
20
21     [...]
22     /* Für die Speicherverwaltung */
23     struct mm_struct *mm, *active_mm;
24
25     [...]
26     /* Prozessnummern */
27     pid_t pid;
28     pid_t tgid;
29     /*
30      * pointers to (original) parent process, youngest child, younger sibling,
31      * older sibling, respectively. (p->father can be replaced with
32      * p->parent->pid)
33     */

```

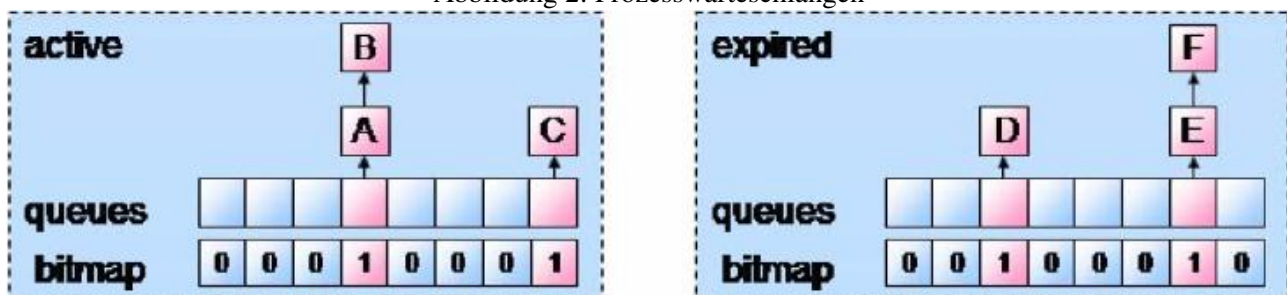
```

34 struct task_struct *real_parent; /* real parent process (when being debugged) */
35 struct task_struct *parent; /* parent process */
36 /*
37  * children/sibling forms the list of my children plus the
38  * tasks I'm ptracing.
39  */
40 struct list_head children; /* list of my children */
41
42 [...]
43 /* Priorität für soft-rt Scheduling */
44 unsigned long rt_priority;
45
46 [...]
47
48 };

```

Alle lauffähigen Prozesse verwaltet der Scheduler in zwei Run-Queues **active** und **expired** (siehe *struct rq* ab Zeile 205). Zusätzlich gibt es jeweils ein Bitmap (Array von Bits) für die Prioritäten (siehe *struct prio_array* ab Zeile 192). Zuerst werden alle Prozesse der Priorität nach von der active Queue abgearbeitet. Ist diese leer, so werden die Queues vertauscht und der Vorgang geht von vorn los.

Abbildung 2: Prozesswarteschlangen



Quellen:

- Josh Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. February 2005
- http://www.linux-magazin.de/heft_abo/ausgaben/2004/02/die_reihenfolge_zaeHLT

Aufgabe 6: Webserver

16 Punkte

Implementieren Sie in C einen einfachen Webserver. Das Programm soll als Kommandozeilenparameter eine Portnummer und ein Arbeitsverzeichnis erhalten und auf dem angegebenen Port auf einfache HTTP GET-Anfragen warten und diese korrekt bearbeiten (vgl. Folie 7.29). Der Arbeitsverzeichnisparameter soll angeben, an welcher Stelle im lokalen Dateisystem sich die HTML Seiten befinden. Geben Sie geeignete Ausgaben über die Abläufe innerhalb Ihres Webserver auf stdout aus, um die Funktionsweise zu verdeutlichen.

Testlauf:

```

chris@firechris-nb:~/server$ ls -l
insgesamt 32
-rw----- 1 root  root    0 2008-01-12 21:00 forbidden.html
-rwxr-x--- 1 chris chris  537 2007-11-07 02:25 index.html
-rwxrwx--- 1 chris chris  276 2008-01-03 16:17 makefile
-rwxr-xr-x 1 chris chris 14926 2008-01-12 21:01 simpleserv
-rwxr-xr-x 1 chris chris  6930 2008-01-09 17:24 simpleserv.c
chris@firechris-nb:~/server$ ./simpleserv 8080 .
Start server on port 8080 in path .
127.0.0.1:52856: Request
127.0.0.1:52856: GET /index.html
127.0.0.1:52856: Send File: ./index.html
127.0.0.1:52856: processed
127.0.0.1:52857: Request
127.0.0.1:52857: GET /huhu.html

```

```

127.0.0.1:52857: Send 404 Not Found: ./huhu.html
127.0.0.1:52857: processed
127.0.0.1:46273: Request
127.0.0.1:46273: GET /forbidden.html
127.0.0.1:46273: Send 403 Forbidden: ./forbidden.html
127.0.0.1:46273: processed

```

simplserv.c

```

1  /**
2  * @file simplserv.c
3  *
4  * TI III: Betriebssysteme und Rechnernetze
5  * WS 2007/08
6  * Übungsblatt Nr. 5,
7  * Aufgabe 6: Webserver
8  *
9  * Dieses Programm implementiert einen HTTP/0.9 - Server.
10 * Es sendet auf GET Anfragen angefragte Dateien oder einen
11 * HTTP-Fehlercode.
12 *
13 * @author Christian Grümme
14 * @see http://cst.mi.fu-berlin.de/teaching/WS0708/19513-V-TI-III/index.html
15 * @date 2007-12-04
16 */
17 #include <stdlib.h>
18 #include <stdio.h>
19 #include <fcntl.h>
20 #include <unistd.h> /* Für close(socket) */
21 #include <string.h>
22 #include <errno.h> /* Für die Fehlerbehandlung */
23 #include <sys/types.h> /* Allgemeine Systemtypen z.B. ssize_t*/
24 #include <sys/socket.h> /* Für die Sockets */
25 #include <netdb.h> /* Für gethostbyname() */
26 #include <netinet/in.h> /* Für Strukturen z.B. sockaddr_in */
27 #include <arpa/inet.h> /* Funktionen mit Network Byte Order */
28
29 #define BUFFER_SIZE 1024 /* Größe des Socketpuffers */
30
31 #define LISTEN_QUEUE 5 /* Wartende Anfragen */
32
33 /** @fn void send_file(int sock, int file)
34 * Schreibt Daten blockweise vom offenen Filedescriptor in den
35 * offenen Socket. Die Descriptor werden nicht geschlossen.
36 *
37 * @param sock Der offene Verbindungssocket
38 * @param path Der offene Filedescriptor
39 */
40 void send_file(int sock, FILE* file)
41 {
42     char buffer[BUFFER_SIZE]; /* Zwischenspeicher */
43     size_t bytes = 1; /* Anzahl gelesener Bytes */
44
45     while(bytes > 0)
46     {
47         /* Lese von Datei */
48         bytes = fread(buffer, 1, BUFFER_SIZE, file);
49         /* Schreibe in den Socket */
50         bytes = send(sock, buffer, bytes, 0);
51     }
52 }
53
54 /** @fn int process(int sock, struct sockaddr_in client, char* path)
55 *
56 * @param sock Der offene Verbindungssocket
57 * @param client Struktur die Informationen zum Clienten enthält
58 * @param path Der Dokumentenpfad
59 */
60 void process(int sock, struct sockaddr_in client, char* path)
61 {
62     int bytes, i = 0;
63     char buffer[BUFFER_SIZE] = { 0 }; /* Puffer für die Anfrage */
64     /* HTTP-Header bei Fehler */
65     char ans403[] = "HTTP/0.9 403 Forbidden\r\n";
66     char ans404[] = "HTTP/0.9 404 Not Found\r\n";
67     char ans501[] = "HTTP/0.9 501 Not Implemented\r\n";
68
69     char file_request[BUFFER_SIZE]; /* String für angefragte Datei */
70     char file_path[BUFFER_SIZE]; /* String für absoluten Pfad */
71     FILE* fd;

```

```

72 char* cl_addr = inet_ntoa(client.sin_addr); /* IP des Klienten */
73 int cl_port = ntohs(client.sin_port); /* Port des Klienten*/
74
75 printf("%s:%d: Request\n", cl_addr, cl_port);
76
77 bytes = recv(sock, buffer, BUFFER_SIZE-2, 0);
78 /* SchlieÙe */
79 buffer[bytes] = '\0';
80
81 if(buffer[0] == 'G' && buffer[1] == 'E' &&buffer[2] == 'T')
82 {
83     i = 4;
84     /* GET-Anfrage, hole Dateiargument */
85     while(buffer[i] != ' ' && buffer[i] != '\r'
86           && buffer[i] != '\n' && buffer[i] != '\0')
87     {
88         file_request[i-4] = buffer[i];
89         ++i;
90     }
91     file_request[i-4] = '\0';
92
93     printf("%s:%d: GET %s\n", cl_addr, cl_port, file_request);
94
95     /* Füge Dokumentenpfad an die angefragte Datei */
96     strcpy(file_path, path);
97     strcat(file_path, file_request);
98
99     /* Öffne Datei */
100    fd = fopen(file_path, "rb");
101
102    /* Überprüfe, ob das Öffnen der Datei erfolgreich war */
103    if(fd == NULL)
104    {
105        /* Lese globale Fehlervariable aus */
106        if(errno == EACCES)
107        {
108            /* Keine Berechtigung - Send 403 */
109            printf("%s:%d: Send 403 Forbidden: %s\n",
110                  cl_addr, cl_port, file_path);
111            send(sock, ans403, sizeof(ans403), 0);
112        }
113        else
114        {
115            /* Sonstiger Fehler - Send 404 */
116            printf("%s:%d: Send 404 Not Found: %s\n",
117                  cl_addr, cl_port, file_path);
118            send(sock, ans404, sizeof(ans404), 0);
119        }
120    }
121    else
122    {
123        printf("%s:%d: Send File: %s\n", cl_addr,
124              cl_port, file_path);
125        send_file(sock, fd);
126        fclose(fd);
127    }
128 }
129 else
130 {
131     /* Keine GET-Anfrage, sende 501*/
132     printf("%s:%d: Send 501 Not Implemented\n", cl_addr,cl_port);
133     send(sock, ans501, sizeof(ans501), 0);
134 }
135 /* SchlieÙe Socket */
136 close(sock);
137
138 printf("%s:%d: processed\n", cl_addr, cl_port);
139 }
140
141 /** @fn int start_server(int port, char* path)
142  * Erstellt ein Socket auf dem übergebenen Port, der in einer
143  * unendlichen while-Schleife auf anfragen wartet.
144  * Bei einer Anfrage wird der neue Verbindungssocket der
145  * Funktionen process übergeben und wartet auf eine neue
146  * Verbindung.
147  *
148  * @param port Der zu verbindende Port
149  * @param path Der Dokumentenpfad
150  * @return Fehlercode
151  */
152 int start_server(int port, char* path)
153 {

```

```

154 int sock, client_sock, opt;
155 unsigned int size;
156 struct sockaddr_in name = { 0 };
157 struct sockaddr_in client_name = { 0 };
158
159 /* Setzte IPv4-Address-Familie */
160 name.sin_family = AF_INET;
161 /* Akzeptiere Verbindung von jeden */
162 name.sin_addr.s_addr = htonl( INADDR_ANY );
163 /* Setze Port */
164 name.sin_port = htons( port );
165
166 /* Initialisiere Socket */
167 sock = socket(AF_INET, SOCK_STREAM, 0);
168 if(sock == -1)
169 {
170     fprintf(stderr, "Error: Cannot create socket\n");
171
172     return EXIT_FAILURE;
173 }
174
175 /* Versuche Port wieder zu benutzen, wenn er schon verwendet wurde */
176 opt = 1;
177 if(setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
178     (char *)&opt, sizeof(opt)) < 0)
179 {
180     fprintf(stderr, "Error: Cannot reuse address\n");
181     return EXIT_FAILURE;
182 }
183
184 /* Weise Socket einen Namen zu */
185 if(bind(sock, (struct sockaddr*) &name, sizeof(name)) < 0)
186 {
187     fprintf(stderr, "Error: Cannot bind socket\n");
188
189     return EXIT_FAILURE;
190 }
191
192 /* Mache Socket lauschfähig */
193 if(listen(sock, LISTEN_QUEUE) < 0)
194 {
195     fprintf(stderr, "Error: Socket cannot listen\n");
196
197     return EXIT_FAILURE;
198 }
199
200 /* Laut Definition von accept nötig */
201 size = sizeof(client_name);
202
203 /* Schleife die auf Verbindungen warten*/
204 while(1)
205 {
206     client_sock = accept(sock,
207         (struct sockaddr*) &client_name, &size);
208     if(client_sock < 0)
209     {
210         fprintf(stderr, "Error: Socket cannot accept\n");
211
212         return EXIT_FAILURE;
213     }
214
215     process(client_sock, client_name, path);
216 }
217 }
218
219 /** @fn int main(int argc, char *argv[])
220 * Main-Funktion, Einstieg in das Programm.
221 * 1. Argument wird als Port, 2. als Dokumentenpfad eingelesen.
222 * Wenn das zweite Argument weggelassen wird, wird der Dokumentenpfad
223 * auf das Arbeitsverzeichnis gesetzt.
224 *
225 * @param argc Anzahl der Argumente
226 * @param argv 1. Argument wird als Port, 2. als Dokumentenpfad eingelesen
227 * @return Status der Terminierung
228 */
229 int main(int argc, char *argv[])
230 {
231     int port;
232
233     /* Überprüfe richtige Anzahl der Argumente */
234     if(argc < 2 || argc > 3)
235     {

```



```
236     fprintf(stderr, "Usage: %s port home_path\n", argv[0]);
237
238     return EXIT_SUCCESS;
239 }
240
241 port = atoi(argv[1]);
242
243 if(port < 1 || port > 65535)
244 {
245     fprintf(stderr, "Error: Port invalid: %s\n", argv[1]);
246
247     return EXIT_FAILURE;
248 }
249
250 printf("Start server on port %d ", port);
251
252 if(argc == 3)
253 {
254     printf("in path %s\n", argv[2]);
255
256     return start_server(port, argv[2]);
257 }
258 else
259 {
260     printf("in path ./\n");
261
262     return start_server(port, ".");
263 }
264 }
```