Es gibt vier Ansätze, um parallele Sitzungen zu ermöglichen:

- Für jede neue Sitzung wird mit Hilfe von *fork*() ein neuer Prozess erzeugt, der sich um die Verbindung zu genau einem Klienten kümmert.
- Für jede neue Sitzung wird ein neuer Thread gestartet.
- Sämtliche Ein- und Ausgabe-Operationen werden asynchron abgewickelt mit Hilfe von *aio_read*, *aio_write* und dem *SIGIO*-Signal.
- Sämtliche Ein- und Ausgabe-Operationen werden in eine Menge zu erledigender Operationen gesammelt, die dann mit Hilfe von *poll* oder *select* ereignis-gesteuert abgearbeitet wird.

Im Rahmen dieser Vorlesung betrachten wir nur die erste und die letzte Variante.

- Diese Variante ist am einfachsten umzusetzen und von genießt daher eine gewisse Popularität.
- Beispiele sind etwa der Apache-Webserver, der jede HTTP-Sitzung in einem separaten Prozess abhandelt, oder verschiedene SMTP-Server, die für jede eingehende E-Mail einen separaten Prozess erzeugen.
- Es gibt fertige Werkzeuge wie etwa *tcpserver* von Dan Bernstein, die die Socket-Operationen übernehmen und für jede Sitzung ein angegebenes Kommando starten, das mit der Netzwerkverbindung über die Standardein- und ausgabe verbunden ist.
- Es ist auch sinnvoll, das in Form einer kleinen Bibliotheksfunktion zu verpacken.

service.h

```
#ifndef AFBLIB_SERVICE_H
#define AFBLIB_SERVICE_H

#include <afblib/hostport.h>

typedef void (*session_handler)(int fd, int argc, char** argv);

/*
 * listen on the given port and invoke the handler for each
 * incoming connection
 */
void run_service(hostport* hp, session_handler handler,
   int argc, char** argv);

#endif
```

- *run_service* eröffnet eine Socket mit der über den Hostport spezifizierten Adresse und startet *handler* in einem separaten Prozess für jede neu eröffnete Sitzung. Diese Funktion läuft permanent und hört nur im Fehlerfalle auf.
- Wenn der *handler* beendet ist, terminiert der entsprechende Prozess.

- Problem: Wir haben konkurrierende Prozesse (für jede Sitzung einen), die eine gemeinsame Menge von Semaphoren verwalten.
- Prinzipiell könnten die das über ein Protokoll untereinander regeln oder den Systemaufrufen für Semaphore (die es auch gibt).
- In diesem Fallbeispiel wird eine primitive und uralte Technik eingesetzt:
  - Für jede Sitzung wird eine Datei angelegt, die nach dem jeweiligen Benutzer benannt wird.
  - Wer eine Semaphore reservieren möchte, versucht, mit dem Systemaufruf *link* einen harten Link von der Datei zum Namen der Semaphore zu erzeugen. Da der Systemaufruf fehlschlägt, wenn der Zielname (der neue Link) bereits existiert, kann das maximal nur einem Prozess gelingen. Der hat dann den gewünschten exklusiven Zugriff.
  - Die anderen Prozesse verharren in einer Warteschleife und hoffen, dass irgendwann einmal die Semaphore wegfällt. Die primitive Lösung verwaltet keine Warteschlange.

Schnittstelle für die Semaphore

lockmanager.h

```
typedef struct lockset {
   char* dirname;
   char* myname;
   stralloc myfile;
   strhash locks;
} lockset;

/*
 * initialize lock set
 */
int lm_init(lockset* set, char* dirname, char* myname);

/* release all locks associated with set and allocated storage */
void lm_free(lockset* set);

/*
 * check status of the given lock and return
 * the name of the holder in holder if it's held
 * and an empty string if the lock is free
 */
int lm_stat(lockset* set, char* lockname, stralloc* holder);

/* block until 'lockname' is locked */
int lm_lock(lockset* set, char* lockname);

/* attempt to lock 'lockname' but do not block */
int lm_nonblocking_lock(lockset* set, char* lockname);

/* release 'lockname' */
int lm_release(lockset* set, char* lockname);
```

service.c

```
void run_service(hostport* hp, session_handler handler,
      int argc, char** argv) {
   int sfd = socket(hp->domain, SOCK_STREAM, hp->protocol);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &hp->addr,
             hp->namelen) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      return;
   }

   /* our childs shall not become zombies */
   struct sigaction action = {
      .sa_handler = SIG_IGN,
      .sa_flags = SA_NOCLDWAIT,
   };
   if (sigaction(SIGCHLD, &action, 0) < 0) return;

   /* ... accept incoming connections ... */
}
```

service.c

```
int fd;
while ((fd = accept(sfd, 0, 0)) >= 0) {
   pid_t child = fork();
   if (child == 0) {
      close(sfd);
      handler(fd, argc, argv);
      exit(0);
   }
   close(fd);
}
```

- Der übergeordnete Prozess wartet mit *accept* auf die jeweils nächste eingehende Netzwerkverbindung.
- Sobald eine neue Verbindung da ist, wird diese mit *fork* an einen neuen Prozess übergeben, der dann *handler* aufruft. Diese Funktion kümmert sich dann nur noch um eine einzelne Sitzung.

MXP-Dienst auf Basis von *run_service*

mutexd.c

```
#include <stdio.h>
#include <stdlib.h>
#include <afblib/hostport.h>
#include <afblib/service.h>
#include "mxpsession.h"

int main (int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 2) {
      fprintf(stderr, "Usage: %s hostport lockdir\n", cmdname);
      exit(1);
   }
   char* hostport_string = *argv++; --argc;
   hostport hp;
   if (!parse_hostport(hostport_string, &hp, 21021)) {
      fprintf(stderr, "%s: hostport in conformance to RFC 2396 expected\n",
         cmdname);
      exit(1);
   }

   /* pass lockdir argument to the service */
   run_service(&hp, mxp_session, argc, argv);
}
```

MXP-Dienst auf Basis von *run_service*

mxpsession.c

```
#define EQUAL(sa,str) (strncmp((sa.s),(str),(sa.len)) == 0)

void mxp_session(int fd, int argc, char** argv) {
   if (argc != 1) return;
   char* lockdir = argv[0];

   inbuf ibuf = {fd};
   outbuf obuf = {fd};
   lockset locks = {0};

   /* send greeting */
   mxp_response greeting = {MXP_SUCCESS};
   if (!write_mxp_response(&obuf, &greeting)) return;
   if (!outbuf_flush(&obuf)) return;

   /* ... rest of the session ... */

   /* release all locks */
   lm_free(&locks);
   /* free allocated memory */
   free_mxp_response(&response);
   stralloc_free(&myname);
}
```

mxpsession.c

```
/* receive identification */
mxp_request id = {{0}};
if (!read_mxp_request(&ibuf, &id)) return;
if (!EQUAL(id.keyword, "id")) return;
stralloc myname = {0};
stralloc_copy(&myname, &id.parameter);
stralloc_0(&myname);
int ok = lm_init(&locks, lockdir, myname.s);

/* send response to identification */
mxp_response response = {MXP_SUCCESS};
stralloc_copys(&response.message, "welcome");
if (!ok) response.status = MXP_FAILURE;
if (!write_mxp_response(&obuf, &response)) return;
if (!outbuf_flush(&obuf)) return;
if (!ok) return;
```

mxpsession.c

```
/* process regular requests */
mxp_request request = {{0}};
while (read_mxp_request(&ibuf, &request)) {
   stralloc lockname = {0};
   stralloc_copy(&lockname, &request.parameter);
   stralloc_0(&lockname);

   if (EQUAL(request.keyword, "stat")) {
      /* ... handling of stat ... */
   } else if (EQUAL(request.keyword, "lock")) {
      /* ... handling of lock ... */
   } else if (EQUAL(request.keyword, "release")) {
      /* ... handling of release */
   } else {
      response.status = MXP_FAILURE;
      stralloc_copys(&response.message, "unknown command");
   }
   if (!write_mxp_response(&obuf, &response)) break;
   if (!outbuf_flush(&obuf)) break;
}
```

mxpsession.c

```
      if (EQUAL(request.keyword, "stat")) {
         mxp_response info = {MXP_CONTINUATION};
         if (lm_stat(&locks, lockname.s, &info.message)) {
            response.status = MXP_SUCCESS;
            if (info.message.len == 0) {
               stralloc_copys(&response.message, "free");
            } else {
               if (!write_mxp_response(&obuf, &info)) break;
               stralloc_copys(&response.message, "held");
            }
         } else {
            response.status = MXP_FAILURE;
            stralloc_copys(&response.message,
               "unable to check lock status");
         }
         free_mxp_response(&info);
```

mxpsession.c

```
    } else if (EQUAL(request.keyword, "lock")) {
       if (lm_nonblocking_lock(&locks, lockname.s)) {
          response.status = MXP_SUCCESS;
          stralloc_copys(&response.message, "locked");
       } else {
          mxp_response notification = {MXP_CONTINUATION};
          stralloc_copys(&notification.message, "waiting");
          if (!write_mxp_response(&obuf, &notification)) break;
          if (!outbuf_flush(&obuf)) break;
          if (lm_lock(&locks, lockname.s)) {
             response.status = MXP_SUCCESS;
             stralloc_copys(&response.message, "locked");
          } else {
             response.status = MXP_FAILURE;
             stralloc_copys(&response.message, "");
          }
       }
    } else if (EQUAL(request.keyword, "release")) {
       stralloc_copys(&response.message, "");
       if (lm_release(&locks, lockname.s)) {
          response.status = MXP_SUCCESS;
       } else {
          response.status = MXP_FAILURE;
       }
```

- Wenn es um sehr schnelle Reaktionen auf eingehende Verbindungen ankommt, erscheint u.U. die Sequenz von *accept* und *fork* zu langsam.
- Alternativ ist es auch denkbar, den Netzwerkdienst zuerst mit *socket*, *bind* und *listen* aufzusetzen und dann mehrere Prozesse im Voraus mit *fork* zu erzeugen, die alle die Socket erben.
- Dann kann jeder dieser Prozesse konkurrierend *accept* aufrufen. Wenn dann eine Netzwerkverbindung durch einen Klienten eröffnet wird, dann ist genau einer der *accept*-Aufrufe erfolgreich. Die anderen Prozesse warten weiter auf andere Klienten.
- Das Modell ist insbesondere durch den Apache-Webserver bekannt geworden.

- Die Zahl der Prozesse, die mit dem Prefork-Modell erzeugt worden ist, begrenzt zunächst die Zahl der parallelen Sitzungen. Das ist nicht befriedrigend.
- Es müssen also bei Bedarf weitere Prozesse erzeugt werden. Aber wie bekommt der Hauptprozess mit, wieviele Prozesse noch frei sind, um eine Verbindung entgegenzunehmen?
- Signale sind ungeeignet, da die sich gegenseitig auslöschen können. Es wird also irgendeine Interprozesskommunikation benötigt. Hierfür bieten sich u.a. Pipelines an, da die leicht vererbt werden können.
- Das bedeutet aber, dass der Hauptprozess mehrere Pipelines unter Beobachtung halten muss. Das ist mit *poll* denkbar.
- Wie können die Prozesse alle abgebaut werden? Wenn der Hauptprozess mit *SIGTERM* terminiert wird, sollten die anderen Prozesse, die nur auf Sitzungen warten, folgen. Bestehende Sitzungen sollten aber nicht unterbrochen werden.

- Dieses Modell kommt noch ohne *poll* aus.
- Zu Beginn wird die gewünschte Zahl von Prozessen erzeugt.
- Jeder der erzeugten Prozesse (Kind-Prozess) legt eine Pipeline an und erzeugt einen weiteren Prozess (Enkel-Prozess), der die Pipeline zum Schreiben offenlässt, während der Erzeuger aus der Pipeline nur liest.
- Der Enkel-Prozess ruft dann *accept* auf, um auf eine eingehende Verbindung zu warten. Sobald *accept* erfolgreich ist, wird die Pipeline geschlossen und die Sitzung gestartet.
- Der Kind-Prozess liest aus der Pipeline und wird damit blockiert, bis der Enkel-Prozess die Pipeline schließt. Danach kann ein neuer Enkel-Prozess erzeugt werden.
- Sollte einer der Kind-Prozesse terminieren, wird vom Hauptprozess ein Nachfolger erzeugt.
- Vorteil: Es sind immer n Prozesse bereit, eine Sitzung entgegenzunehmen. Nachteil: Wir benötigen insgesamt $2n + 1$ Prozesse.

preforked_service.c

```
void run_preforked_service(hostport* hp, session_handler handler,
      unsigned int number_of_processes, int argc, char** argv) {
   assert(number_of_processes > 0);
   int sfd = socket(hp->domain, SOCK_STREAM, hp->protocol);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &hp->addr, hp->namelen) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      close(sfd);
      return;
   }

   /* ... setup termination handler ... */
   /* ... create preforked processes ... */
   /* ... start a new preforked process for every one terminating ... */
   /* ... terminate everything ... */
}
```

preforked_service.c

```
/* setup termination handler */
struct sigaction action = {
   .sa_handler = termination_handler,
};
if (sigaction(SIGTERM, &action, 0) != 0) {
   return;
}

/* create preforked processes */
pid_t child_pid[number_of_processes];
for (int i = 0; i < number_of_processes; ++i) {
   pid_t pid = spawn_preforked_process(sfd, handler, argc, argv);
   if (pid < 0) return;
   child_pid[i] = pid;
}
```

preforked_service.c

```
/* start a new preforked process for every one terminating */
while (!terminate) {
   pid_t child; int wstat;
   if ((child = wait(&wstat)) > 0) {
      int index;
      for (index = 0; index < number_of_processes; ++index) {
         if (child_pid[index] == child) break;
      }
      if (index < number_of_processes) {
         child = spawn_preforked_process(sfd, handler, argc, argv);
         child_pid[index] = child;
         if (child < 0) break;
      }
   }
}

/* terminate everything */
for (int i = 0; i < number_of_processes; ++i) {
   if (child_pid[i] > 0) {
      kill(child_pid[i], SIGTERM);
   }
}
```

preforked_service.c

```
static pid_t spawn_preforked_process(int sfd, session_handler handler,
      int argc, char** argv) {
   pid_t child = fork();
   if (child) return child;

   /* our childs shall not become zombies */
   struct sigaction action = {
      .sa_handler = SIG_IGN,
      .sa_flags = SA_NOCLDWAIT,
   };
   if (sigaction(SIGCHLD, &action, 0) < 0) exit(1);

   while (!terminate) {
      /* ... */
   }
   exit(0);
}
```

preforked_service.c

```
while (!terminate) {
   /* now create another process and share a pipeline with it */
   int pipe_fds[2];
   if (pipe(pipe_fds) < 0) exit(1);
   pid_t pid = fork();
   if (pid < 0) exit(1);
   if (pid == 0) {
      /* grandchild of the original process */
      close(pipe_fds[0]); /* close reading side of pipe */
      int fd = accept(sfd, 0, 0);
      close(sfd);
      if (fd < 0) exit(1);
      /* now close the writing side of the pipe to indicate that
         we are busy with running a session */
      close(pipe_fds[1]);
      /* run the session and exit */
      handler(fd, argc, argv);
      exit(0);
   }
   close(pipe_fds[1]); /* close writing side of the pipe */
   /* now wait for the child process to accept a connection;
      we get notified by the closure of the pipe */
   char ch;
   if (read(pipe_fds[0], &ch, 1) < 0 && errno == EINTR && terminate) {
      kill(pid, SIGTERM); /* propagate termination */
   }
   close(pipe_fds[0]);
}
```

- Ein- und Ausgabe-Operationen blockieren normalerweise, bis sie durchgeführt werden können.
- Dies erschwert die Parallelisierung solcher Operationen bzw. die Möglichkeit, auf unterschiedliche Ein- und Ausgabe-Ereignisse zu reagieren.
- Mit den Systemaufrufen *poll* und *select* gibt es die Möglichkeit, zu warten, bis wir mindestens eine von beliebig vielen geplanten Ein- und Ausgabe-Operationen durchführen können, ohne blockiert zu werden.
- Der Vorteil dieser Schnittstelle liegt darin, dass wir die synchrone Arbeitsweise nicht aufgeben müssen.
- Wir betrachten hier im weiten *poll*, da dieser Systemaufruf eine etwas elegantere Schnittstelle als *select* bietet.

multiplexor.c

```
    if (poll(mpx.pollfds, count, -1) <= 0) return;
```

- *poll* erhält drei Parameter:
  - Einen Zeiger auf ein Array mit Einträgen des Datentyps **struct** *pollfd*,
  - einer natürlichen Zahl, die die Länge des Arrays angibt, und
  - einer zeitlichen Beschränkung in Millisekunden. (Hier wird -1 angegeben, wenn keine Befristung gewünscht wird.)
- Der Datentyp **struct** *pollfd* umfasst folgende Felder:

fd	Dateideskriptor
events	Menge der Ereignisse, auf die gewartet wird
revents	Menge der Ereignisse, die eingetreten sind

- Im Erfolgsfalle liefert *poll* die Zahl der eingetretenen Ereignisse zurück. Falls die zeitliche Beschränkung erreicht wurde, ohne dass eines der Ereignisse eintrat, wird 0 zurückgeliefert. Im Falle von Fehlern wird -1 zurückgegeben.

- Relevant sind nur *POLLIN* und *POLLOUT*. Prinzipiell kann *poll* noch Unterscheidungen treffen, ob priorisierte Pakete über die Netzwerkverbindung ankamen, aber das wird normalerweise nicht verwendet.
- Das Ereignis *POLLIN* bedeutet, dass ein *read*-Systemaufruf für den Dateideskriptor abgesetzt werden kann, ohne dass der Prozess blockiert wird.
- Analog bedeutet *POLLOUT*, dass ein *write*-Systemaufruf abgesetzt werden kann, ohne Gefahr zu laufen, blockiert zu werden.
- Bei mit *listen* vorbereiteten Sockets kann ebenfalls *POLLIN* verwendet werden. Das Ereignis tritt dann ein, sobald sich eine neue Netzwerkverbindung anbahnt und *accept* blockierungsfrei aufgerufen werden kann.

- Die Umsetzung des Prefork-Modells lässt sich mit Hilfe von *poll* verbessern, da wir dann keinen Wächterprozess pro Prozess benötigen, der bereit ist, eine Verbindung mit *accept* entgegenzunehmen.
- Bei n Prozessen, die bereit sein sollen, eine Sitzung entgegenzunehmen, werden jetzt nur noch insgesamt $n + 1$ Prozesse benötigt, d.h. es kommt nur noch der Hauptprozess hinzu.
- Der Hauptprozess erzeugt selbst alle weiteren Prozesse und beobachtet dann mit Hilfe von *poll* die Pipeline-Verbindungen zu den einzelnen Prozessen.
- Sobald die letzte offene Schreibverbindung einer Pipeline geschlossen wird, tritt auf der lesenden Seite das *POLLIN*-Ereignis ein, damit das Eingabe-Ende erkannt werden kann. (Ein *read* würde dann blockierungsfrei eine 0 zurückliefern.)

Prefork-Modell mit Hilfe von *poll*

preforked_service.c

```
static pid_t spawn_preforked_process(int sfd, int pipefds[2],
      session_handler handler, int argc, char** argv) {
   if (pipe(pipefds) < 0) return -1;
   pid_t child = fork();
   if (child) {
      close(pipefds[1]);
      return child;
   }
   close(pipefds[0]);

   int fd = accept(sfd, 0, 0); close(sfd);
   if (fd < 0) exit(1);
   /* now close the writing side of the pipe to indicate that
      we are busy with running a session */
   close(pipefds[1]);
   /* run the session and exit */
   handler(fd, argc, argv);
   exit(0);
}
```

- Die Funktion *spawn_preforked_process* vereinfacht sich, da nur noch ein Prozess erzeugt wird.

preforked_service.c

```
/* create preforked processes */
pid_t child_pid[number_of_processes];
struct pollfd pollfds[number_of_processes];
for (int i = 0; i < number_of_processes; ++i) {
   /* a pipe is used to signal that one of the
      preforked processes accepted a connection */
   int pipefds[2];
   pid_t pid = spawn_preforked_process(sfd, pipefds, handler,
      argc, argv);
   pollfds[i] = (struct pollfd) { .fd = pipefds[0], .events = POLLIN};
   if (pid < 0) return;
   child_pid[i] = pid;
}
```

- Der Hauptprozess erzeugt hier zu Beginn die gewünschte Zahl von Prozessen.
- Dabei wird gleichzeitig die *pollfds*-Datenstruktur aufgebaut, um all die Pipelines gleichzeitig beobachten zu können.

preforked_service.c

```
while (!terminate) {
   if (poll(pollfds, number_of_processes, -1) <= 0) break;
   for (int i = 0; i < number_of_processes; ++i) {
      if (pollfds[i].revents == 0) continue;
      close(pollfds[i].fd);
      int pipefds[2];
      pid_t pid = spawn_preforked_process(sfd, pipefds, handler,
         argc, argv);
      if (pid < 0) return;
      pollfds[i] = (struct pollfd) {
         .fd = pipefds[0], .events = POLLIN};
      child_pid[i] = pid;
   }
}
```

- Mit *poll* warten wir darauf, dass die schreibende Seite eine der Pipes geschlossen wird.
- Dies ist das Signal, dass ein neuer Prozess zu starten ist, dessen Pipeline dann in *pollfds* ersatzweise eingetragen wird.

- In manchen Fällen ist es vorteilhaft, wenn alle Sitzungen einen gemeinsamen Adressraum verwenden, damit sitzungsübergreifende Datenstrukturen leichter verwaltet werden können.
- Prinzipiell lässt sich das mit Hilfe des Systemaufrufs *poll* erreichen, mit dem auf das Eintreten eines Ein- oder Ausgabe-Ereignisses gewartet werden kann.
- Dies führt zu einem grundlegend anderen Programmierstil, bei dem Ein- und Ausgaben ereignisgesteuert abgewickelt werden.
- Da bei jedem Ereignis entsprechende Behandler neu aufgerufen werden, kann der Sitzungskontext nicht in lokalen Variablen verwaltet werden. Stattdessen sind dafür dynamische Datenstrukturen zu verwenden, die bei jedem Aufruf erst lokalisiert werden müssen.

multiplexor.h

```
typedef void (*multiplexor_handler)(connection* link);
void run_multiplexor(int socket, multiplexor_handler open_handler,
   multiplexor_handler input_handler, multiplexor_handler close_handler,
   void* mpx_handle);
bool write_to_link(connection* link, char* buf, unsigned int len);
ssize_t read_from_link(connection* link, char* buf, unsigned int len);
void close_link(connection* link);
```

- Es ist sinnvoll, die Verwendung von *poll* in eine geeignete Bibliothek zu verpacken.
- Die Funktion *run_multiplexor* läuft dann permanent und übernimmt somit die vollständige Kontrolle des Programms. Es werden nur noch Behandler aufgerufen, wenn
  - neue Netzwerkverbindungen eröffnet werden,
  - neue Eingaben vorliegen oder
  - eine Verbindung beendet wird.
- Eine Rückkehr von *run_multiplexor* gibt es nur im Fehlerfalle.

multiplexor.h

```
typedef void (*multiplexor_handler)(connection* link);
void run_multiplexor(int socket, multiplexor_handler open_handler,
   multiplexor_handler input_handler, multiplexor_handler close_handler,
   void* mpx_handle);
bool write_to_link(connection* link, char* buf, unsigned int len);
ssize_t read_from_link(connection* link, char* buf, unsigned int len);
void close_link(connection* link);
```

- Konkret ruft *run_multiplexor* den Behandler *open_handler* für neue Verbindungen, *input_handler* für neue Eingaben und *close_handler* für beendete Verbindungen auf.
- Die Behandler dürfen selbst nichts direkt auf eine Netzwerkverbindung ausgeben, da dies zu längeren Blockaden führen könnte. Stattdessen muss dies durch *write_to_link* erfolgen, das dafür Warteschlangen unterhält.
- Der Parameter *mpx_handle* dient als Zeiger auf eine eigene Datenstruktur, die den Behandlern unter *connection−>mpx_handle* zur Verfügung gestellt wird.

Datenstruktur für eine Netzwerkverbindung 256

multiplexor.h

```
typedef struct connection {
   int fd;
   void* handle; /* may be freely used by the application */
   void* mpx_handle; /* corresponding parameter from run_multiplexor */
   bool eof;
   struct output_queue_member* oqhead;
   struct output_queue_member* oqtail;
   struct connection* next;
   struct connection* prev;
} connection;
```

- Für jede Netzwerkverbindung gibt es eine zugehörige Datenstruktur.
- Neben der Netzwerkverbindung *fd* und den beiden benutzerdefinierten Zeigern *handle* und *mpx_handle*, kommen noch folgende Felder hinzu:

eof	wird auf *true* gesetzt, sobald ein Eingabeende erkannt wurde
oqhead und *oqtail*	Zeiger auf das erste und letzte Element der Warteschlange mit den auszugebenden Puffern
next und *prev*	doppelt verkettete Liste aller Netzwerkverbindungen

multiplexor.c

```
typedef struct output_queue_member {
   char* buf;
   unsigned int len;
   unsigned int pos;
   struct output_queue_member* next;
} output_queue_member;
// ...
int write_to_link(connection* link, char* buf, unsigned int len);
```

- Jedes Element der Warteschlange weist auf einen Puffer.
- Zu Beginn ist die Position *pos* gleich 0 und *len* entspricht der Länge, die an *write_to_link* übergeben worden ist.
- Wenn jedoch der entsprechende Aufruf von *write* nicht vollständig umgesetzt werden kann, dann wird *pos* um die übertragene Quantität erhöht und *len* entsprechend gesenkt.
- Sobald die Schreiboperation abgeschlossen ist, wird nicht nur das Warteschlangen-Element, sondern auch der Puffer freigegeben.

multiplexor.c

```
typedef struct multiplexor {
   /* parameters passed to run_multiplexor */
   int socket;
   multiplexor_handler ohandler, ihandler, chandler;
   void* mpx_handle;
   /* additional administrative fields */
   bool socketok; /* becomes false when accept() fails */
   connection* head; /* double-linked linear list of connections */
   connection* tail; /* its last element */
   int count; /* number of connections */
   struct pollfd* pollfds; /* parameter for poll() */
   unsigned int pollfdslen; /* allocated len of pollfds */
   connection** pollcs; /* of the same len as pollfds */
} multiplexor;
```

- Es gibt nur ein Objekt dieser Datenstruktur, das von *run_multiplexor* zu Beginn angelegt wird.
- Neben den Parametern von *run_multiplexor* werden in der doppelt verketteten Liste mit *head* und *tail* alle offenen Verbindungen verwaltet. In *count* findet sich deren Zahl.
- *pollfds* zeigt auf ein dynamisch belegtes Feld mit *pollfdslen* Elementen. Dies dient der Verwaltung der *poll* zu übergebenden Datenstruktur.

Vorbereitung der Datenstruktur für *poll*

multiplexor.c

```
/* prepare fields pollfds and pollfdslen in mpx in
   dependence of the current set of connections */
static int setup_polls(multiplexor* mpx) {
   int len = mpx->count;
   if (mpx->socketok) ++len;
   if (len == 0) return 0;
   /* weed out links which have been closed
      and where our output queue is empty */
   connection* link = mpx->head;
   while (link) {
      connection* next = link->next;
      if (link->eof && link->oqhead == 0) remove_link(mpx, link);
      link = next;
   }
   /* allocate or enlarge pollfds, if necessary */
   if (mpx->pollfdslen < len) {
      mpx->pollfds = realloc(mpx->pollfds, sizeof(struct pollfd) * len);
      if (mpx->pollfds == 0) return 0;
      mpx->pollcs = realloc(mpx->pollcs, sizeof(connection*) * len);
      if (mpx->pollcs == 0) return 0;
      mpx->pollfdslen = len;
   }

   /* ... */
}
```

Vorbereitung der Datenstruktur für *poll*

multiplexor.c

```
/* prepare fields pollfds and pollfdslen in mpx in
   dependence of the current set of connections */
static int setup_polls(multiplexor* mpx) {
   /* ... */

   int index = 0;
   /* look for new network connections as long accept()
      returned no errors so far */
   if (mpx->socketok) {
      mpx->pollcs[index] = 0;
      mpx->pollfds[index++] = (struct pollfd) {mpx->socket, POLLIN};
   }
   /* look for incoming network connections and
      check whether we can write any pending output packets
      without blocking */
   link = mpx->head;
   while (link) {
      short events = 0;
      if (!link->eof) events |= POLLIN;
      if (link->oqhead) events |= POLLOUT;
      mpx->pollcs[index] = link;
      mpx->pollfds[index++] = (struct pollfd) {link->fd, events};
      link = link->next;
   }
   return index;
}
```

multiplexor.c

```
static bool add_connection(multiplexor* mpx) {
   int newfd;
   if ((newfd = accept(mpx->socket, 0, 0)) < 0) {
      mpx->socketok = false; return true;
   }
   connection* link = malloc(sizeof(connection));
   if (link == 0) return false;
   *link = (connection) {
      .fd = newfd, .handle = 0, .mpx = mpx,
      .mpx_handle = mpx->mpx_handle,
      .eof = false, .oqhead = 0, .oqtail = 0,
      .next = 0, .prev = mpx->tail,
   };
   if (mpx->tail) {
      mpx->tail->next = link;
   } else {
      mpx->head = link;
   }
   mpx->tail = link; ++mpx->count;
   if (mpx->ohandler) (*mpx->ohandler)(link);
   return true;
}
```

Entfernung einer Verbindung

multiplexor.c

```
/* remove a connection from the double-linked linear
   list of connections
*/
static void remove_link(multiplexor* mpx, connection* link) {
   close(link->fd);
   if (link->prev) {
      link->prev->next = link->next;
   } else {
      mpx->head = link->next;
   }
   if (link->next) {
      link->next->prev = link->prev;
   } else {
      mpx->tail = link->prev;
   }
   if (mpx->chandler) (*mpx->chandler)(link);
   free(link);
   --mpx->count;
}
```

multiplexor.c

```
/* read one input packet from the given network connection */
ssize_t read_from_link(connection* link, char* buf, unsigned int len) {
   if (link->eof) return 0;
   ssize_t nbytes = read(link->fd, buf, len);
   if (nbytes <= 0) {
      link->eof = true;
      if (link->oqhead == 0) remove_link((multiplexor*)link->mpx, link);
   }
   return nbytes;
}
```

- Wenn *poll* signalisiert hat, dass wir von einer Verbindung einlesen dürfen, dann wird der entsprechende Behandler aufgerufen, der wiederum *read_from_link* aufruft, um die Eingabe in den eigenen Puffer einzulesen.

multiplexor.c

```
/* write one pending output packet to the given network connection */
static void write_to_socket(multiplexor* mpx, connection* link) {
   ssize_t nbytes = write(link->fd,
      link->oqhead->buf + link->oqhead->pos,
      link->oqhead->len - link->oqhead->pos);
   if (nbytes <= 0) {
      remove_link(mpx, link);
   } else {
      link->oqhead->pos += nbytes;
      if (link->oqhead->pos == link->oqhead->len) {
         output_queue_member* old = link->oqhead;
         link->oqhead = old->next;
         if (link->oqhead == 0) {
            link->oqtail = 0;
         }
         free(old->buf); free(old);
         if (link->oqhead == 0 && link->eof) {
            remove_link(mpx, link);
         }
      }
   }
}
```

Schreiben auf eine Verbindung

multiplexor.c

```
bool write_to_link(connection* link, char* buf, unsigned int len) {
   assert(len >= 0);
   if (len == 0) {
      free(buf); return true;
   }
   output_queue_member* member = malloc(sizeof(output_queue_member));
   if (!member) return false;
   member->buf = buf; member->len = len; member->pos = 0;
   member->next = 0;
   if (link->oqtail) {
      link->oqtail->next = member;
   } else {
      link->oqhead = member;
   }
   link->oqtail = member;
   return true;
}
```

- Diese Funktion ist von den Behandlern aufzurufen, wenn etwas auf eine der Netzwerkverbindungen auszugeben ist.
- Der Ausgabepuffer wird dann in die entsprechende Warteschlange eingereiht.

multiplexor.c

```
void close_link(connection* link) {
   link->eof = 1;
   shutdown(link->fd, SHUT_RD);
}
```

- Bei bidirektionalen Netzwerkverbindungen ist es möglich, nur eine Seite zu schließen.
- Dies geht nicht mit *close*, das sofort beide Seiten schließen würde, sondern mit *shutdown*, mit dem eine spezifizierte Seite geschlossen werden kann.
- Hier wird aus der Sicht des Aufrufers die lesendende Seite geschlossen, also die Verbindung vom Klienten zum Dienst. Danach können keine weiteren Anfragen mehr eintreffen, aber die Warteschlange der abzuarbeitenden Ausgabe-Puffer kann noch abgearbeitet werden.
- Erst wenn die Warteschlange ganz leer ist, dann wird (von *remove_link*) die Verbindung vollständig geschlossen.

multiplexor.c

```
void run_multiplexor(int socket, multiplexor_handler open_handler,
      multiplexor_handler input_handler, multiplexor_handler close_handler,
      void* mpx_handle) {
   multiplexor mpx = {
      .socket = socket, .ohandler = open_handler,
      .ihandler = input_handler, .chandler = close_handler,
      .mpx_handle = mpx_handle, .socketok = true,
   };
   int count;
   while ((count = setup_polls(&mpx)) > 0) {
      if (poll(mpx.pollfds, count, -1) <= 0) return;
      for (int index = 0; index < count; ++index) {
         if (mpx.pollfds[index].revents == 0) continue;
         int fd = mpx.pollfds[index].fd;
         if (fd == mpx.socket) {
            if (!add_connection(&mpx)) return;
         } else {
            connection* link = mpx.pollcs[index]; assert(link);
            if (mpx.pollfds[index].revents & POLLIN) {
               (*mpx.ihandler)(link);
            }
            if (mpx.pollfds[index].revents & POLLOUT) {
               write_to_socket(&mpx, link);
            }
         }
      }
   }
}
```

- Der *input_handler* wird für jedes eingehende Paket aufgerufen.
- Da Pakete fragmentiert sein können, sind dies möglicherweise Bruchstücke einer Anfrage oder auch Teile mehrerer Anfragen.
- Entsprechend muss die Eingabe wieder gepuffert und zerlegt werden, da normalerweise eine Reaktion erst bei einer vollständig übermittelten Anfrage erfolgen sollte.
- Eine ereignisgesteuerte Behandlung wäre daher aus Anwendungssicht leichter zu programmieren, wenn sie auf vollständigen Anfragen beruhen würde.
- Die Erkennung einer vollständigen Anfrage ist im allgemeinen Fall nicht ganz trivial zu spezifizieren. Im folgenden wird eine Lösung auf Basis regulärer Ausdrücke vorgestellt, die für textbasierte Protokolle gut geeignet ist.

mpx_session.h

```
typedef void (*mpx_handler)(session* s);

int mpx_session_scan(session* s, ...);
int mpx_session_printf(session* s, const char* restrict format, ...);
void close_session(session* s);

void run_mpx_service(hostport* hp, const char* regexp,
   mpx_handler ohandler, mpx_handler rhandler, mpx_handler hhandler,
   void* global_handle);
```

- *run_mpx_service* erhält einen regulären Ausdruck, der eine Anfrage spezifiziert.
- Dieser reguläre Ausdruck darf mit Hilfe runder Klammern beliebig viele Elemente der Anfrage herausgreifen – analog zu *inbuf_scan*.
- Der *rhandler* (*request handler*) wird dann für jede vollständig vorliegende Anfrage aufgerufen und kann dann mit *mpx_session_scan* die herausgegriffenen Elemente in *stralloc*-Objekte hineinkopieren lassen.

mxprequest.h

```
#define MXP_REQUEST_RE "([a-z]+) (.*?)\r?\n"
```

mutexd.c

```
   run_mpx_service(&hp, MXP_REQUEST_RE,
      mxp_session_open, mxp_session_read, mxp_session_hangup,
      locks);
```

- Beim Aufruf von *run_mpx_service* wird der reguläre Ausdruck zum Erkennen einer Anfrage mitgegeben.

mxpsession.c

```
void mxp_session_read(session* s) {
   struct mxp_session* ms = s->handle; assert(ms);
   if (!read_mxp_request(s, &ms->request)) {
      close_session(s); return;
   }
   /* ... process request and generate response ... */
   if (!write_mxp_response(s, &ms->response)) {
      close_session(s);
   }
}
```

- Der Behandler *mxp_session_read* wird jetzt nur aufgerufen, wenn eine vollständige Anfrage vorliegt. Entsprechend sollte *read_mxp_request* eine Anfrage einlesen können.

mxprequest.c

```
bool read_mxp_request(session* s, mxp_request* request) {
   return
      mpx_session_scan(s, &request->keyword, &request->parameter) == 2;
}
```

mxresponse.c

```
/* write one (possibly partial) response to */
bool write_mxp_response(session* s, mxp_response* response) {
   return mpx_session_printf(s, "%c%.*s\r\n", response->status,
      response->message.len, response->message.s) > 0;
}
```

- Die Einlese-Operation für Anfragen und die Ausgabe-Operation für Antworten verwenden hier die entsprechenden Funktionen aus *mpx_session.h*