- Semaphore als Instrument zur Synchronisierung von Prozessen gehen auf den niederländischen Informatiker Edsger Dijkstra zurück, der diese Kontrollstruktur Anfang der 60er-Jahre entwickelte.
- Eine Semaphore wird irgendeiner Ressource zugeordnet, auf die zu einem gegebenen Zeitpunkt nur ein Prozess zugreifen darf, d.h. Zugriffe müssen exklusiv erfolgen.
- Damit sich konkurrierende Prozesse beim Zugriff auf die Ressource nicht ins Gehege kommen, erfolgt die Synchronisierung über Semaphore, die folgende Operationen anbieten:

  P Der Aufrufer wird blockiert, bis die Ressource frei ist. Danach ist ein Zugriff möglich.

  V Gib die Ressource wieder frei.

```
P(sema); // warte, bis die Semaphore fuer uns reserviert ist
// ... Kritischer Bereich, in dem wir exklusiven Zugang
// zu der mit sema verbundenen Ressource haben ...
V(sema); // Freigabe der Semaphore
```

- Semaphores werden so verwendet, dass jeder exklusive Zugriff auf eine Ressource in die Operationen P und V geklammert wird.
- Intern werden typischerweise Semaphore repräsentiert durch eine Datenstruktur mit einer ganzen Zahl und einer Warteschlange. Wenn die ganze Zahl positiv ist, dann ist die Semaphore frei. Ist sie 0, dann ist sie belegt, aber niemand sonst wartet darauf. Ist sie negativ, dann entspricht der Betrag der Länge der Warteschlange.
- Bei P wird entsprechend der Zähler heruntergezählt und, falls der Zähler negativ wurde, der Aufrufer in die Warteschlange befördert. Ansonsten erhält er sofort Zugang zur Ressource.
- Bei V wird der Zähler hochgezählt und, falls der Zähler noch nicht positiv ist, das am längsten wartende Mitglied der Warteschlange daraus entfernt und aufgeweckt.

Anmerkungen zu den Namen P und V, die beide auf Edsger Dijkstra zurückgehen:

- P steht für „Prolaag“ und V für „Verhoog“.
- „Verhoog“ ist niederländisch und bedeutet übersetzt „hochzählen“.
- Da das niederländische Gegenstück „verlaag“ (übersetzt: „herunterzählen“) ebenfalls mit einem „v“ beginnt, schuf Dijkstra das Kunstwort „prolaag“.
- Die erste Notiz, in der Dijkstra diese Operationen und die Namen P und V definierte, findet sich unter `http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF`. Eine genaue Datierung liegt nicht vor, aber die Notiz muss wohl 1963 oder 1964 entstanden sein.
- 1968 erfolgte die erste Veröffentlichung in seinem Beitrag *Cooperating sequential processes* zur NATO-Konferenz über Programmiersprachen.

- Das *Mutual Exclusion Protocol* (MXP) sei ein Protokoll, das die Synchronisation einander fremder Prozesse über Semaphore erlaubt, die durch einen Netzwerkdienst verwaltet werden.
- Der Netzwerkdienst (in diesem Beispiel *mutexd* genannt) erlaubt beliebig viele Klienten, die sich jeweils namentlich identifizieren müssen.
- Jede der Klienten kann dann die bekannten P- und V-Operationen für beliebige Semaphoren absetzen oder den aktuellen Status einer Semaphore überprüfen.

- Das Protokoll sieht Anfragen (von einem Klienten an den Dienst) und Antworten (von dem Dienst an den Klienten) vor.
- Anfragen bestehen immer aus genau einer Zeile, die mit CR LF terminiert wird.
- Antworten bestehen aus einer oder mehrerer Zeilen, die ebenfalls mit CR LF terminiert werden.
- Die letzte Zeile einer Antwort beginnt immer mit dem Buchstaben „S“ oder „F“. „S“ steht für eine erfolgreich durchgeführte Operation, „F“ für eine fehlgeschlagene Operation.
- Wenn eine Antwort aus mehreren Zeilen besteht, dann beginnen alle Antwortzeilen mit Ausnahme der letzten Zeile mit dem Buchstaben „C“.

- Anfragen beginnen mit einer Folge von Kleinbuchstaben (dem Kommando), einem Leerzeichen und einem Parameter. Parameter sind beliebige Folgen von 8-Bit-Zeichen, die weder CR, LF noch Nullbytes enthalten dürfen.
- Antwortzeilen bestehen aus einem Statusbuchstaben („S“, „F“ oder „C“) und einer beliebigen Folge von 8-Bit-Zeichen, die weder CR, LF noch Nullbytes enthalten dürfen.

Folgende Anfragen werden unterstützt:

id *login*	Anmelden mit eindeutigem Namen. Dies muss als erstes erfolgen.
stat *sema*	Liefert den Status der genannten Semaphore. Wenn die Semaphore frei ist, wird „Sfree“ als Antwort zurückgeliefert. Ansonsten eine C-Zeile mit dem Namen desjenigen, der sie gerade reserviert hat, gefolgt von „Sheld“.
lock *sema*	Wartet, bis die Semaphore frei wird, und blockiert sie dann für den Aufrufer. Falls gewartet werden muss, gibt es sofort eine Antwortzeile „Cwaiting“. Sobald die Semaphore für den Aufrufer reserviert ist, folgt die Antwortzeile „Slocked“.
release *sema*	Gibt eine reservierte Semaphore wieder frei. Antwort ist ein einfaches „S“.

```
←  S
→  id alice
←  Swelcome
→  stat beer
←  Sfree
→  stat wine
←  Cbob
←  Sheld
→  lock beer
←  Slocked
→  lock wine
←  Cwaiting
←  Slocked
→  release wine
←  S
→  release cake
←  F
→  release beer
←  S
```

mxprequest.h

```
#ifndef MXP_REQUEST_H
#define MXP_REQUEST_H

#include <stdbool.h>
#include <stralloc.h>
#include <afblib/inbuf.h>
#include <afblib/outbuf.h>

typedef struct mxp_request {
   stralloc keyword;
   stralloc parameter;
} mxp_request;

/* read one request from the given input buffer */
bool read_mxp_request(inbuf* ibuf, mxp_request* request);

/* write one request to the given outbuf buffer */
bool write_mxp_request(outbuf* obuf, mxp_request* request);

/* release resources associated with request */
void free_mxp_request(mxp_request* request);

#endif
```

mxprequest.c

```
/* read one request from the given input buffer */
bool read_mxp_request(inbuf* ibuf, mxp_request* request) {
   return
      inbuf_scan(ibuf, "([a-z]+) ([^\r\n]*)\r\n",
         &request->keyword, &request->parameter) == 2;
}

/* write one request to the given outbuf buffer */
bool write_mxp_request(outbuf* obuf, mxp_request* request) {
   return
      outbuf_printf(obuf, "%.*s %.*s\r\n",
         request->keyword.len, request->keyword.s,
         request->parameter.len, request->parameter.s) > 0;
}

/* release resources associated with request */
void free_mxp_request(mxp_request* request) {
   stralloc_free(&request->keyword);
   stralloc_free(&request->parameter);
}
```

mxpresponse.h

```
#ifndef MXP_RESPONSE_H
#define MXP_RESPONSE_H

#include <stdbool.h>
#include <afblib/inbuf.h>
#include <afblib/outbuf.h>

typedef enum mxp_status {
   MXP_SUCCESS = 'S',
   MXP_FAILURE = 'F',
   MXP_CONTINUATION = 'C',
} mxp_status;

typedef struct mxp_response {
   mxp_status status;
   stralloc message;
} mxp_response;

/* write one (possibly partial) response to the given output buffer */
bool write_mxp_response(outbuf* obuf, mxp_response* response);

/* read one (possibly partial) response from the given input buffer */
bool read_mxp_response(inbuf* ibuf, mxp_response* response);

void free_mxp_response(mxp_response* response);

#endif
```

MXP-Pakete: Antworten

mxpresponse.c

```
bool read_mxp_response(inbuf* ibuf, mxp_response* response) {
   int ch = inbuf_getchar(ibuf);
   switch (ch) {
      case MXP_SUCCESS:
      case MXP_FAILURE:
      case MXP_CONTINUATION:
         response->status = ch;
         break;
      default:
         return false;
   }
   return inbuf_scan(ibuf, "([^\r\n]*)\r\n", &response->message) == 1;
}

bool write_mxp_response(outbuf* obuf, mxp_response* response) {
   return outbuf_printf(obuf, "%c%.*s\r\n", response->status,
      response->message.len, response->message.s) > 0;
}

void free_mxp_response(mxp_response* response) {
   stralloc_free(&response->message);
}
```

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 traditionell 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.

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);
   perror("run_service"); exit(1);
}
```

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.

```
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 ... */
}
```

```
/* 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;
}
```

```
/* 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);
   }
}
```

```
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);
}
```

```
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 Weiteren *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.