Systemnahe Software II
SS 2019

Andreas F. Borchert
Universität Ulm

22. Juli 2019

Inhalte:

- Prozesse unter UNIX
- Signale
- Interprozess-Kommunikation mit einem besonderen Schwerpunkt auf TCP/IP

Syllabus: Ziele

- Eingehendes Verständnis der POSIX-Schnittstellen und Abstraktionen für Prozesse, Signale, Kommunikation und Synchronisierung.
- Sichere Programmierung mit C in diesen Bereichen und das Erkennen von potentiellen Sicherheitslücken.
- Grundkenntnisse in TCP/IP und der Gestaltung von Internet-Protokollen.
- Eingehendes Verständnis der Muster zur Verarbeitung paralleler Sitzungen über TCP/IP.

Syllabus: Voraussetzungen

- Teilnahme an Systemnahe Software I. Dazu gehören insbesondere
  - Grundlagen in C einschließlich der dynamischen Speicherverwaltung,
  - Grundkenntnisse der POSIX-Schnittstellen im Bereich von Ein- und Ausgabe (*open*, *read*, *write* und die darüber liegende Schicht der *stdio*) und
  - Grundkenntnisse in der sicheren Programmierung in C (mitsamt der *stralloc*-Bibliothek von Dan Bernstein)
- Freude daran, etwas auch an einem Rechner auszuprobieren und genügend Ausdauer, dass nicht beim ersten Fehlversuch aufgegeben wird.

Warum ist sichere Programmierung wichtig?

- Wir beschäftigen uns im Rahmen der Vorlesung auch mit Netzwerkanwendungen und der Umsetzung von Netzwerkprotokollen.
- Kontakte über das Netzwerk sind normalerweise weltweit über das Internet möglich.
- Entsprechend tragen wir die Verantwortung dafür, keine offenen Scheunentore zu hinterlassen.
- Das bedeutet, dass wir bei jeder Code-Zeile und bei jedem Detail genau wissen müssen, was wir da tun, welche Gefahren lauern und wie wir diese abwehren.
- Die Folgen können sonst unabsehbar sein wie beim Heartbleed-Bug...

- Das Heartbeat-Protokoll wurde in Ergänzung zum SSL-Protokoll definiert: RFC 6520
- Das Protokoll soll zwei Probleme lösen:
  - Eine schnellere Alternative zu *SO_KEEPALIVE*
  - Ein alternativer Ansatz zur *Path MTU Discovery*, nachdem die ursprünglich dafür gedachten ICMP-Pakete allzu häufig von Firewalls weggefiltert werden
- Im Rahmen des Protokolls können Pings geschickt werden mit Daten (Payload) und einer zufällig gewählten Ergänzung. Solche Pings werden dann beantwortet, wobei der Payload zusammen mit anderen zufälligen Daten zurückgeschickt wird.
- Der Payload hat eine variable Länge. Deswegen findet sich im Header eines Heartbeat-Pakets ein Feld mit zwei Bytes, das den Umfang der Payload-Daten spezifiziert.

ssl/ssl3.h

```
typedef struct ssl3_record_st
        {
/*r */  int type;               /* type of record */
/*rw*/  unsigned int length;    /* How many bytes available */
/*r */  unsigned int off;       /* read/write offset into 'buf' */
/*rw*/  unsigned char *data;    /* pointer to the record data */
/*rw*/  unsigned char *input;   /* where the decode bytes are */
/*r */  unsigned char *comp;    /* only used with decompression - malloc()ed *
/*r */  unsigned long epoch;    /* epoch number, needed by DTLS1 */
/*r */  unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
        } SSL3_RECORD;
```

- Eine typische Datenstruktur für einen Kommunikationspuffer, entnommen aus openssl-1.0.1f
- *data* zeigt auf (die bereits entschlüsselten) Daten, die wir über das Netzwerk erhalten haben.
- *length* gibt an, wieviele Bytes in *data* zum Lesen zur Verfügung stehen.

ssl/d1-both.c

```
        unsigned char *p = &s->s3->rrec.data[0], *pl;
        /* ... */
        /* Read type and payload length first */
        hbtype = *p++;
        n2s(p, payload);
        pl = p;
```

- *s−>s3−>rrec* ist vom Typ *SSL3_RECORD* und repräsentiert das eingelesene Datenpaket, in dem sich ein Heartbeat-Paket befindet.
- *p* zeigt auf den Anfang des Datenbereichs des eingelesenen Pakets.
- Dort ist zu Beginn der Typ des Heartbeat-Pakets (ein Byte) und der Umfang des beigefügten Payloads (zwei Bytes).
- *n2s* konvertiert zwei Bytes vom Netzwerk in *network byte order* in eine ganze Zahl (*short*).
- *payload* kann hier ein beliebiger Wert zwischen 0 und 65535 sein, der vollkommen frei von der anderen Seite gewählt werden kann.

ssl/d1-both.c

```
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,
           buffer, 3 + payload + padding);
```

- Hier wird ein Antwort-Paket geschnürt (in Reaktion zu einem Ping), bei der die erhaltene Payload zurückzuschicken ist mitsamt einer Ergänzung aus zufälligen Daten (*padding*).
- Mit Hilfe von *memcpy* wird von *pl* (zeigt an den Anfang der erhaltenen Payload) nach *bp* kopiert.
- Kopiert werden *payload* Bytes. Es wird nirgends überprüft, ob noch *payload* Bytes hinter *pl* belegt sind...

Kann ein Lesen (und Weitergeben) des Speicherinhalts jenseits des Eingabe-Puffers ein Problem darstellen?

- Ja! Ziemlich anschaulich erklärt es Randall Munroe in xkcd: `http://www.xkcd.com/1354/`
- Bruce Schneier dazu:

  "Catastrophic" is the right word. On the scale of 1 to 10, this is an 11.

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Montag von 14–16 Uhr in der Helmholtzstraße 22, Raum E.04.
- Die Übungen finden am Dienstag von 14–16 Uhr in der Helmholtzstraße 18, Raum E.44, statt.
- Die Vorlesung fällt am 10. Juni 2019 (Pfingstmontag) aus und wird dann am 11. Juni 2019 zum Zeitpunkt der Übungen in der Helmholtzstraße 22, Raum E.03, nachgeholt.
- Webseite:
  `https://www.uni-ulm.de/mawi/mawi-numerik/lehrenumerik/sommersemester-2019/vorlesung-systemnahe-software-ii/`

Syllabus: Übungen

- Es gibt ein- und gelegentlich auch zweiwöchige Übungsblätter.
- Wir haben weder Tutoren noch Korrekteure. Entsprechend können Ihre Lösungen nicht korrigiert werden.
- Es besteht aber die Möglichkeit, Lösungen elektronisch einzureichen. Das ermöglicht ein allgemeines Feedback in den Übungen.
- Gelegentlich werden elektronische Einreichungen auch mit Testsuites verknüpft sein.
- Bitte melden Sie sich für die Vorlesung bei SLC an.
- Sie sollten, sofern noch nicht vorhanden, sich um einen Shell-Zugang bei uns bemühen.

Syllabus: Prüfung

- Eine Vorleistung für die Teilnahme zur Prüfung ist nicht mehr erforderlich.
- Die Prüfungen erfolgen mündlich zu individuell vereinbarten Terminen.

Syllabus: Skript

- Es gibt ein Skript, das auf der Webseite kapitelweise veröffentlicht wird.
- Parallel gibt es Präsentationen (wie diese), die ebenfalls als PDF zur Verfügung gestellt werden.
- Wenn Sie das Skript oder die Präsentationen ausdrucken möchten, nutzen Sie dazu bitte die entsprechenden Einrichtungen des KIZ. Im Prinzip können Sie dort beliebig viel drucken, wenn Sie genügend Punkte dafür erworben haben.
- Das Druck-Kontingent, das Sie bei uns kostenfrei erhalten (das ist ein Privileg und kein natürliches Recht), darf für die Übungen genutzt werden, jedoch nicht für das Ausdrucken von Skripten oder Präsentationen.

Syllabus: Sprechstunden

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: `andreas.borchert@uni-ulm.de`
- Meine reguläre Sprechzeit ist am Mittwoch 10:00–11:30 Uhr. Zu finden bin ich in der Helmholtzstraße 20, Zimmer 1.23.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, dass ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

Syllabus: Nicht verzweifeln!

- Immer wieder kann es mal vorkommen, dass es zu scheinbar unlösbaren Problemen bei einer Übungsaufgabe kommt.
- Geben Sie dann bitte nicht auf! Kontaktieren Sie mich bitte stattdessen.
- Schicken Sie bitte in so einem Fall alle Quellen zu und vergessen Sie nicht, eine präzise Beschreibung des Problems mitzuliefern.
- Das kann auch am Wochenende funktionieren.

Syllabus: Feedback

- Feedback ist ausdrücklich erwünscht.
- Es besteht insbesondere auch immer die Möglichkeit, auf Punkte noch einmal einzugehen, die zunächst noch nicht klar geworden sind.
- Vertiefende Fragen und Anregungen sind auch willkommen.
- Ich spule hier nicht immer das gleiche Programm ab. Jede Vorlesung und jedes Semester verläuft anders und das hängt auch von Ihnen ab!

- Definition von Ritchie und Thompson, den Hauptentwicklern von UNIX:
  A process is the execution of an image.
- Zum *image* zählen der übersetzte Programmtext (Maschinencode und vorinitialisierte Daten) und der Ausführungskontext.

Ein Programm wird in einem bestimmten Kontext ausgeführt. Zu diesem Kontext gehören

- der Adressraum, in dem unter anderem der Programmtext (als Maschinencode) und die Daten untergebracht sind,
- pro Thread ein Satz Maschinenregister einschließlich der Stackverwaltung (Stack-Zeiger, Frame-Zeiger) und dem PC (*program counter*, verweist auf die nächste auszuführende Instruktion), *errno* und Signalmaske und
- weitere Statusinformationen, die vom Betriebssystem verwaltet werden wie beispielsweise Informationen über geöffnete Dateien.

- Zu einem Prozess können mehrere Ausführungsfäden (*Threads*) gehören, die ebenfalls vom Betriebssystem verwaltet werden. Entsprechend gibt es nicht nur Status-Informationen auf Prozess-Ebene, sondern auch (in einem geringeren Umfang) auf Thread-Ebene (wie beispielsweise die Signalmaske).
- Alle wesentlichen Status-Informationen wie etwa User-ID, die Gruppenzugehörigkeiten, die geöffneten Dateien und der Adressraum sind allen Threads eines Prozesses gemein.
- Deswegen wird ein Prozess auch als Rechtegemeinschaft betrachtet.

printpid.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
   printf("%d\n", (int) getpid());
}
```

- Jeder Prozess hat unter UNIX eine gleichbeibende identifizierende positive ganze Zahl, die mit *getpid*() abgefragt werden kann.
- Bei der Mehrheit der UNIX-Systeme liegt die Prozess-ID im Bereich von 1 bis 32767. Die Eindeutigkeit ist jedoch nur zu Lebzeiten garantiert. Sobald ein Prozess beendet wird, kann die gleiche Prozess-ID später einem neuen Prozess zugeordnet werden. Alle gängigen UNIX-Systeme vergeben Prozess-IDs reihum, wobei bereits vergebene Prozess-IDs übersprungen werden.

```
heim$ cat /proc/sys/kernel/pid_max
32768
heim$
```

- Bei Linux-Systemen ist der Wert abrufbar und kann bei 64-Bit-Systemen auf bis zu 2^{22} erhöht werden (*PID_MAX_LIMIT*).
- Andere Systeme wie Solaris erlauben hier ebenso bei Bedarf höhere Werte.
- Da die Prozess-IDs sequentiell vergeben werden, sind sie nicht wirklich geeignet für
  - die Generierung sicherheitsrelevanter Seed-Werte,
  - die Benennung temporärer Dateien oder
  - die Erzeugung von Session-IDs.

- Ein Prozess kann sich jederzeit mit *exit*() beenden und dabei einen Statuswert im Bereich von 0 bis 255 angeben.
- Die *exit*-Funktion kann in C-Programmen auch implizit aufgerufen werden: Ein **return** in der *main*-Funktion führt zu einem entsprechenden *exit* und wenn das Ende der *main*-Funktion erreicht wird, entspricht dies einem *exit*(0).
- Ein Exit-Wert von 0 deutet dabei eine erfolgreiche Terminierung an; andere Werte, insbesondere *EXIT_FAILURE*, werden als Misserfolg gewertet. Diese Konventionen orientieren sich zwar an UNIX, sind aber auch Bestandteil der ISO-Standards 9899-1999 und 9899-2011.

- Neue Prozesse können nur in Form eines Klon-Vorganges mit Hilfe des Systemaufrufs *fork*() erzeugt werden.
- Der Adressraum, die Maschinenregister und fast der gesamte Status des Betriebssystems für den erzeugenden Prozess werden dupliziert.
- Das bedeutet, dass beide Prozesse (der *fork*() aufrufende Prozess und der neu erzeugte Prozess) einen zu Beginn gleich aussehenden Adressraum vorfinden. Änderungen werden jedoch nur bei jeweils einem der beiden Prozesse wirksam.
- Um dies effizient umzusetzen und um einen hohen Kopieraufwand bei der *fork*-Operation zu vermeiden, kommt hier eine Verzögerungstechnik zum Zuge: *copy on write*.

- Einige Statusinformationen beim Betriebssystem betreffen beide Prozesse. So werden offene Dateiverbindungen vererbt und können nach dem Aufruf von *fork* gemeinsam genutzt werden.
- Dies bezieht sich aber nur auf Dateiverbindungen, die zum Zeitpunkt des *fork*-Aufrufs eröffnet waren und nicht auf Dateien, die später von einem der beiden Prozesse neu eröffnet werden.
- Einige Statusinformationen des Betriebssystems werden *nicht* weitergegeben. Dazu gehören beispielsweise Locks und anhängige Signale.
- Der Standard IEEE Std 1003.1 zählt in der Manualseite zu *fork(2)* alle Statusinformationen auf, die weitergegeben werden.

Neuer Prozess als Klon des alten Prozesses

clones.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

- Ein neuer Prozess beginnt nicht irgendwo mit einem neuen Programmtext bei *main*().
- Stattdessen finden wir nach *fork*() zwei weitgehend übereinstimmende Kopien eines Prozesses vor, die alle den gleichen Programmtext hinter dem Aufruf von *fork*() fortsetzen.
- Deswegen wird in diesem Beispiel das zweite *printf* doppelt ausgeführt.

clones.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

```
doolin$ clones | cat
I am feeling lonely!
Hey, I am cloned!
I am feeling lonely!
Hey, I am cloned!
doolin$
```

- Warum erhalten wir jetzt die Ausgabe „I am feeling lonely!“ nun doppelt?

clones.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

- Erfolgt die Ausgabe direkt auf ein Terminal, wird zeilenweise gepuffert. In diesem Falle erfolgt die Ausgabe des ersten *printf*() noch vor dem Aufruf von *fork*().
- Falls jedoch voll gepuffert wird — dies ist bei der Ausgabe in eine Datei oder in eine Pipeline der Fall — dann erfolgt vor dem *fork*() noch keine Ausgabe. Stattdessen wird der Puffer von *stdout* durch *fork*() dupliziert, womit die doppelte Ausgabe der ersten Zeile provoziert wird.

clones2.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n"); fflush(stdout);
   fork();
   fork();
   fork();
   printf("Hey, I am cloned!\n");
}
```

- Die doppelte Ausgabe eines ungeleerten Puffers lässt sich durch die rechtzeitige Leerung des Puffers mit Hilfe von *fflush*() vermeiden.

- Grundsätzlich ist *fork* nicht unproblematisch, weil der gesamte Zustand der Datenstrukturen dupliziert wird.
- Dies verschärft sich, wenn Datenstrukturen mit Ressourcen (wie etwa Dateideskriptoren oder Locks) verknüpft sind.
- Besonders problematisch ist dies in Verbindung mit Threads, da hier *fork* ohne besondere Vorkehrungen völlig asynchron zu den anderen Threads ausgeführt wird.
- Eine saubere Lösung existiert hier nicht. Wenn Threads mit *fork* kombiniert werden, sollte auf *fork* möglichst unmittelbar ein *exec* folgen und dazwischen müsste alles *async-signal-safe* sein (mehr dazu später).
- Im POSIX-Standard gibt es *pthread_atfork* aus der Thread-Bibliothek, das jedoch entgegen der ursprünglichen Intention nicht geeignet ist für Thread-Ressourcen (wie etwa ein Mutex), aber für andere Datenstrukturen (wie etwa die der *stdio*) verwendet werden kann.

Puffer leeren mit *pthread_atfork*

clones3.c

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void flush_all() {
   fflush(0); /* flush all streams */
}

int main() {
   /* invoke flush_all implicitly before fork() is executed */
   pthread_atfork(flush_all, 0, 0);

   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

- *pthread_atfork* hat drei Funktionsparameter des Typs **void** (∗)(**void**).
- Die erste Funktion wird vor *fork* implizit aufgerufen, die anderen beim Elternprozess bzw. Kindprozess nach *fork*.
- Ein Nullzeiger kann angegeben werden, wenn eine der Funktionen entfällt.

Wie können Ursprungsprozess und Klon getrennte Wege gehen? 32

clones4.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t parent;

   printf("I am feeling lonely!\n"); fflush(stdout);
   parent = getpid();
   fork();
   if (getpid() == parent) {
      printf("I am the parent process!\n");
   } else {
      printf("I am the child process!\n");
   }
}
```

- Damit der ursprüngliche Prozess und der mit *fork* erzeugte Klon getrennte Wege verfolgen können, müssen sie sich voneinander unterscheiden können. Ein naheliegendes Mittel ist hier die Prozess-ID, da der ursprüngliche Prozess seine behält und der Klon eine neue erhält.

Der Rückgabewert von *fork*

fork.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
   pid_t pid;

   pid = fork();
   if (pid == -1) {
      perror("unable to fork"); exit(1);
   }
   if (pid == 0) {
      /* child process */
      printf("I am the child process: %d.\n", (int) getpid());
      exit(0);
   }
   /* parent process */
   printf("The pid of my child process is %d.\n", (int) pid);
}
```

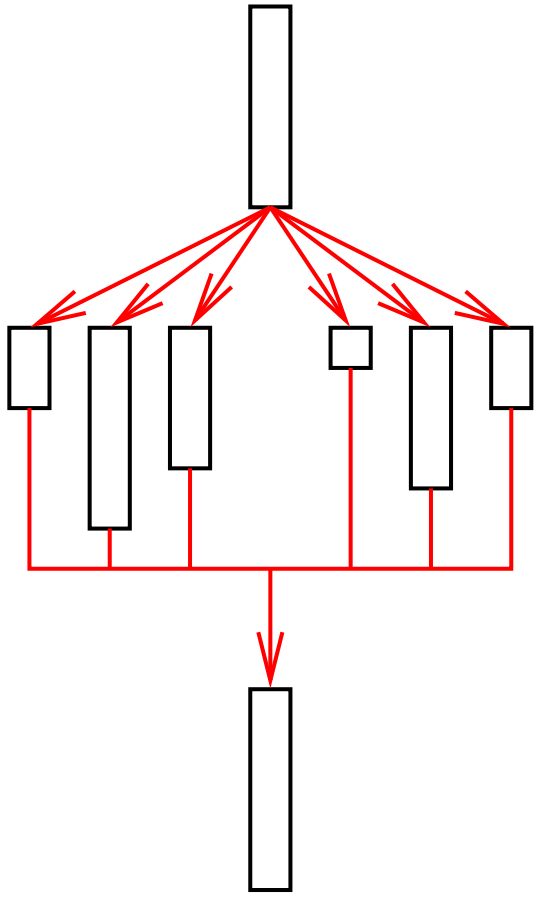
- *fork*() liefert -1 im Falle von Fehlern, 0 für den neu erzeugten Prozess und die Prozess-ID des neu erzeugten Prozesses beim alten Prozess.

Der Rückgabewert von *fork*

fork.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
   pid_t pid;

   pid = fork();
   if (pid == -1) {
      perror("unable to fork"); exit(1);
   }
   if (pid == 0) {
      /* child process */
      printf("I am the child process: %d.\n", (int) getpid());
      exit(0);
   }
   /* parent process */
   printf("The pid of my child process is %d.\n", (int) pid);
}
```

- Ein explizites *exit*() beim neu erzeugten Prozess verhindert, dass der Klon hinter der **if**-Anweisung den für den Erzeuger vorgesehenen Programmtext ausführt.



zu Beginn nur ein Prozeß

Erzeugen neuer Prozesse

Warten, bis alle neu erzeugten Prozesse beendet sind

- Es mag Fälle geben, bei denen neue Prozesse erzeugt und dann „vergessen“ werden. Im Normalfall jedoch stößt das weitere Schicksal des neuen Prozesses auf Interesse und insbesondere ist es nicht unüblich, dass der erzeugende Prozess auf das Ende der von ihm erzeugten Prozesse warten möchte.
- Dies macht insbesondere dann Sinn, wenn mehrere Prozesse erzeugt werden, die parallel Teilprobleme des Gesamtproblems lösen. Dann wartet der erzeugende Prozess nach Erzeugung all der Unterprozesse, bis sie alle ihre Teilaufgaben erledigt haben. Dieses Muster wird „Fork and Join“ genannt.

forkandwait.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
   pid_t child, pid; int stat;

   child = fork();
   if (child == -1) {
      perror("unable to fork"); exit(1);
   }
   if (child == 0) {
      /* child process */
      srand(getpid());
      exit(rand());
   }

   /* parent process */
   pid = wait(&stat);
   if (pid == child) {
      if (WIFEXITED(stat)) {
         printf("exit code of child = %d\n", WEXITSTATUS(stat));
      } else {
         printf("child terminated abnormally\n");
      }
   } else {
      perror("wait");
   }
}
```

forkandwait.c

```
if (child == 0) {
   /* child process */
   srand(getpid());
   exit(rand());
}
```

- Der neu erzeugte Prozess initialisiert den Pseudo-Zufallszahlengenerator mit *srand* und holt sich dann mit *rand* eine pseudo-zufällige Zahl ab.
- Da der Exit-Wert nur 8 Bit und entsprechend nur die Werte von 0 bis 255 umfasst, werden die höherwertigen Bits der Pseudo-Zufallszahl implizit weggeblendet.

forkandwait.c

```
/* parent process */
pid = wait(&stat);
if (pid == child) {
   if (WIFEXITED(stat)) {
      printf("exit code of child = %d\n", WEXITSTATUS(stat));
   } else {
      printf("child terminated abnormally\n");
   }
} else {
   perror("wait");
}
```

- Die Funktion *wait* wartet auf die Terminierung eines beliebigen Unterprozesses, der noch *nicht* von *wait* zurückgeliefert wurde.
- Falls es einen solchen Prozess nicht mehr gibt, wird -1 zurückgeliefert.
- Ansonsten liefert *wait* die Prozess-ID des terminierten Prozesses und innerhalb von *stat* den zugehörigen Status.

Zurückgelieferter Status bei *wait*

Der in *stat* abgelegte Status des Unterprozesses besteht aus mehreren Komponenten, die angeben,

- wie ein Prozess sein Leben beendete (durch *exit*() oder durch ein Signal (bei einem Crash oder Verwendung von *kill*()) oder ob der Prozess nur gestoppt wurde,
- welcher Wert bei *exit*() angegeben wurde, falls *exit*() benutzt wurde und
- welches Signal das Leben des Prozesses terminierte bzw. stoppte, falls der Prozess nicht mit *exit*() endete.

- Prozesse können auch zu Prozessgruppen zusammengefasst werden.
- Für die ID einer Prozessgruppe kann die eines Prozesses aus der Gruppe gewählt werden.
- Dies ist bei Fork-and-Join sinnvoll, da dann gezielt auf Prozesse einer Gruppe gewartet werden kann.

Prozessgruppen zur Synchronisierung

forkandjoin.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>

int main() {
   pid_t group = 0;
   for (int i = 0; i < 10; ++i) {
      pid_t pid = fork();
      if (pid < 0) {
         perror("fork"); exit(1);
      }
      if (pid == 0) {
         exit(getpid() % 255);
      }
      setpgid(pid, group);
      if (group == 0) {
         group = pid;
      }
   }
   int wstat;
   while (waitpid(-group, &wstat, 0) > 0) {
      if (WIFEXITED(wstat)) {
         printf(" %d", WEXITSTATUS(wstat));
      } else {
         printf(" X");
      }
   }
   printf("\n");
}
```

forkandjoin.c

```
setpgid(pid, group);
if (group == 0) {
   group = pid;
}
```

- *pid* ist die Prozess-ID eines frisch erzeugten Kindprozesses, *group* ist zu Beginn 0.
- Mit *setpgid* kann die Prozess-ID von *pid* gesetzt werden. *pid* muss dabei der eigene Prozess und ein Kindprozess sein. Wenn *group* den Wert 0 hat, wird implizit als Gruppen-ID der Wert von *pid* gewählt.

forkandjoin.c

```
int wstat;
while (waitpid(-group, &wstat, 0) > 0) {
   /* ... */
}
```

- *waitpid* bietet eine verallgemeinerte Schnittstelle im Vergleich zu *wait*.
- Der erste Parameter spezifiziert, worauf gewartet wird:
  - $pid > 0$: es wird auf den genannten Prozess gewartet
  - $pid == 0$: es wird auf einen Prozess aus der gleichen Prozessgruppe wie der eigene Prozess gewartet
  - $pid == -1$: es wird (wie bei *wait*) auf einen beliebigen Kindprozess gewartet
  - $pid < -1$: es wird auf ein Prozess der Prozessgruppe $-pid$ gewartet.
- Der zweite Parameter ist (wie bei *wait*) ein Zeiger auf den zurückzuliefernden Status und der dritte Parameter erlaubt die Angabe von Optionen. Hier wäre ggf. *WNOHANG* interessant – dann ist *waitpid* nicht-blockierend und liefert -1, wenn noch keiner der spezifizierten Kindprozesse soweit ist.

- Was geschieht mit dem Rückgabewert bei *exit*() und dem sonstigen Endstatus eines Prozesses, wenn der übergeordnete Prozess nicht zeitig *wait*() aufruft?
- Das UNIX-System lässt solche toten Prozesse noch in seiner Verwaltung weiterleben, so dass der Endstatus noch bewahrt wird, aber die nicht mehr benötigten Ressourcen freigegeben werden.
- Prozesse, die sich in diesem Stadium befinden, werden als Zombies bezeichnet.

genzombie.c

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

int main() {
   pid_t child = fork();
   if (child == -1) {
      perror("fork"); exit(1);
   }
   if (child == 0) exit(0);
   printf("%d\n", child);
   sleep(60);
}
```

- Der neu erzeugte Prozess verabschiedet sich hier sofort mit *exit*(), während der übergeordnete Prozess mit Hilfe eines *sleep*()-Aufrufes sich für 60 Sekunden zur Ruhe legt.
- Während dieser Zeit verbleibt der Unterprozeß im Zombie-Status.

```
doolin$ genzombie&
[1]     24489
doolin$ 24490

doolin$ ps -ylp 24489,24490
 S   UID   PID  PPID  C PRI NI   RSS     SZ    WCHAN TTY      TIME CMD
 S   120 24489 23591  0  64 28   616    936        ? pts/31   0:00 genzombi
 Z   120 24490 24489  0   0                            0:00 <defunct>
doolin$
```

- In der ersten Spalte gibt *ps* bei dieser Aufrufvariante den Status eines Prozesses an.
- „*Z*“ steht dabei für Zombie, „*S*“ für schlafend.
- Weitere Varianten sind „*O*“ für gerade arbeitend, „*R*“ für arbeitsbereit und „*T*“ für gestoppt.

orphan.c

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

int main() {
   pid_t child;
   child = fork();
   if (child == -1) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      printf("Hi, my parent is %d\n", (int) getppid());
      sleep(5);
      printf("My parent is now %d\n", (int) getppid());
      exit(0);
   }
   sleep(3);
   exit(0);
}
```

- Wenn sich der übergeordnete Prozess verabschiedet, dann wird ihm der Prozess mit der Prozess-ID 1 als neuer übergeordneter Prozess zugewiesen.

- Der Prozess mit der Prozess-ID 1 spielt eine besondere Rolle unter UNIX. Es ist der erste Prozess, der vom Betriebssystem selbst erzeugt wird. Er führt den unter */etc/init* oder */sbin/init* zu findenden Programmtext aus.
- Dieser Prozess startet weitere Prozesse anhand einer Konfigurationsdatei (bei uns unter */etc/inittab*) und ruft ansonsten *wait*() auf, um den Status der von ihm selbst erzeugten Prozesse oder den von Waisenkindern entgegenzunehmen.
- Auf diese Weise wird dann auch der Zombie-Status eines Prozesses beendet, wenn es zum Waisenkind wird.

Mit *fork*() ist es möglich, neue Prozesse zu erzeugen. Allerdings teilen die neuen Prozesse sich den Programmtext mit ihrem Erzeuger. Wie ist nun der Wechsel zu einem anderen Programmtext möglich? Die Lösung dafür ist der Systemaufruf *exec*(), der

- den gesamten virtuellen Adressraum des aufrufenden Prozesses auflöst,
- an seiner Stelle einen neuen einrichtet mit einem angegebenen Programmtext,
- sämtliche Maschinenregister für den Prozess neu initialisiert und
- Statusinformationen des Betriebssystems weitgehend unverändert belässt

datum.c

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

int main() {
   execl(
      "/usr/bin/date",  /* path of the program */
      "/usr/bin/date",  /* name of the program, i.e. argv[0] */
      "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
      0                 /* terminate list of arguments */
   );
   /* not reached except if execl failed */
   perror("/usr/bin/date"); exit(1);
}
```

- Dieses Programm ersetzt seinen eigenen Programmtext durch den von *date*.

datum.c

```
execl(
   "/usr/bin/date",  /* path of the program */
   "/usr/bin/date",  /* name of the program, i.e. argv[0] */
   "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
   0                 /* terminate list of arguments */
);
```

- *execl* erlaubt die Angabe beliebig vieler Kommandozeilenargumente in der Form einzelner Funktionsparameter. Mit einem Nullzeiger wird die Liste der Parameter beendet.
- Dabei ist zu beachten, dass der Pfadname des auszuführenden Programms und der später unter *argv*[0] zu findende Kommandoname getrennt angegeben werden. Normalerweise sind beide gleich, es gibt aber auch Ausnahmen.

datum.c

```
execl(
   "/usr/bin/date",  /* path of the program */
   "/usr/bin/date",  /* name of the program, i.e. argv[0] */
   "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
   0                 /* terminate list of arguments */
);
/* not reached except if execl failed */
perror("/usr/bin/date"); exit(1);
```

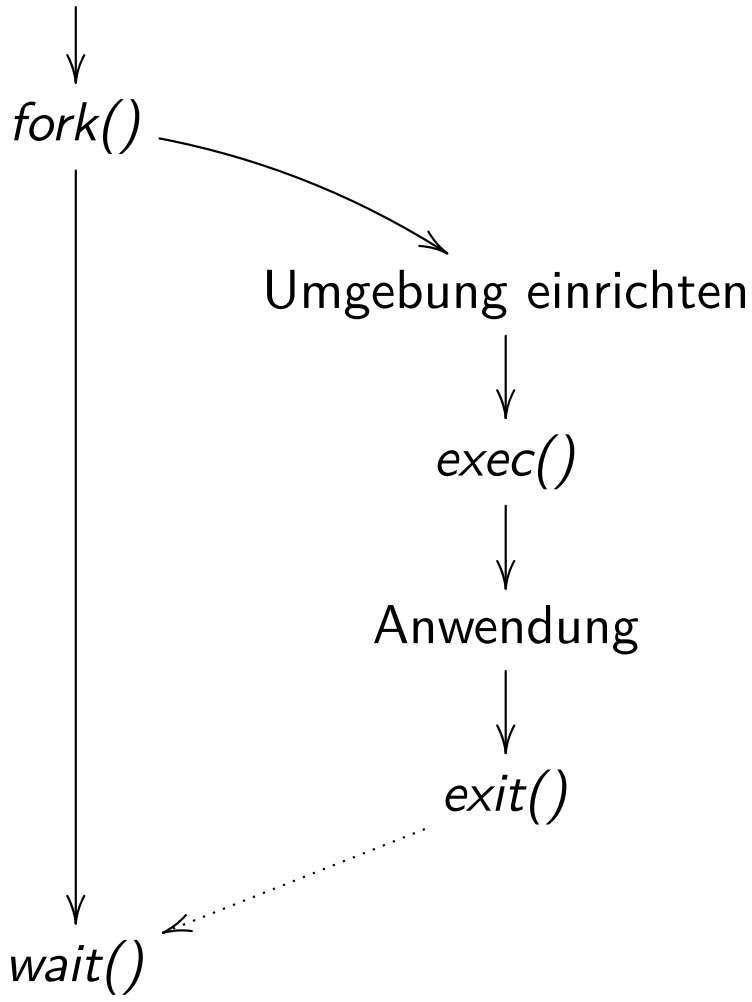
- Normalerweise geht es im Programmtext nach einem Aufruf von *execl*() nicht weiter, weil im Erfolgsfalle das Programm ausgetauscht wurde. Nur bei einem Fehler (weil z.B. das *date*-Kommando nicht gefunden wurde) wird das Programm hinter dem Aufruf von *execl*() fortgesetzt.

- Auf den ersten Blick erscheinen diese vier Systemaufrufe seltsam. Warum ist eine Kombination aus *fork*() und *exec*() notwendig, um einen neuen Prozess mit einem neuen Programmtext in Gang zu setzen?
- Wäre es nicht besser und einfacher, nur einen einzigen Systemaufruf dafür zu haben?
- Die Frage verschärft sich, wenn berücksichtigt wird, dass in der Zeit der frühen UNIX-Implementierungen die Technik des „*copy on write*“ noch nicht zur Verfügung stand. Stattdessen war es bei *fork*() notwendig, den gesamten Speicher zu kopieren.
- Bei BSD wurde deswegen zeitweise *fork1*() eingeführt, das diesen Kopiervorgang unterdrückte, um die typische Kombination von *fork*() und *exec*() nicht zu teuer werden zu lassen.

```
//IS198CPY JOB (IS198T30500),'COPY JOB',CLASS=L,MSGCLASS=X
//COPY01   EXEC PGM=IEBGENER
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DSN=OLDFILE,DISP=SHR
//SYSUT2   DD  DSN=NEWFILE,
//           DISP=(NEW,CATLG,DELETE),
//           SPACE=(CYL,(40,5),RLSE),
//           DCB=(LRECL=115,BLKSIZE=1150)
//SYSIN    DD  DUMMY
```

- UNIX ist keinesfalls das erste Betriebssystem, das Prozesse unterstützte. Die älteren Systeme boten in der Tat die Kombination aus *fork*() und *exec*() in einem Systemaufruf an.
- Das Beispiel zeigt ein Kopierkommando in der JCL (Job Command Language) aus der IBM-Mainframe-Welt (von der Wikipedia übernommen). Hieran zeigt sich, dass dies die Kommandosprache deutlich verkompliziert. Der Haken liegt darin, dass Prozesse häufig eine Umgebung erwarten, die mehr umfaßt als eine Kommandozeile. Wichtiger Bestandteil der Umgebung sind bereits im Vorfeld eingerichtete Ein- und Ausgabeverbindungen und die Zuteilung von Ressourcen.

- So sieht die traditionelle Erzeugung eines Prozesses aus:
  - Erzeuge einen neuen Prozess mit einem gegebenen Programmtext mit einem Systemaufruf, der *fork*() und *exec*() kombiniert.
  - Einrichtung der Umgebung für den neuen Prozess.
  - Start des neuen Prozesses.
- Entsprechend ist es notwendig, alle wichtigen Systemaufrufe für die Einrichtung einer Umgebung einschließlich dem Öffnen von Ein- und Ausgabeverbindungen in zwei Varianten zu unterstützen: Die eine Variante bezieht sich auf den eigenen Prozess, die andere für einen untergeordneten Prozess, der noch nicht gestartet wurde.



- Die Trennung in *fork*() und *exec*() erlaubt die Konfiguration der Umgebung des aufzurufenden Programms innerhalb der Shell mit ganz normalen Systemaufrufen, die sich auf den eigenen Prozess beziehen.

```
theon$ tinysh
% date
Mon Apr 29 11:09:30 CEST 2019
% date >out
% cat out
Mon Apr 29 11:09:33 CEST 2019
% awk {print$4} <out
11:09:33
% theon$
theon$
```

- Die kleine Shell *tinysh* erlaubt
  - den Aufruf von Kommandos mit beliebig vielen Parametern, die durch Leerzeichen getrennt werden,
  - die Umlenkung der Standard-Ein- und Ausgabe, wobei auch das Anhängen unterstützt wird und
  - die Auswertung des *wait*-Systemaufrufs.
- Die Konfiguration des aufzurufenden Programms erfolgt hier zwischen *fork* und *exec*.

tinysh.c

```
int main() {
   inbuf in = {0}; outbuf out = {1};
   stralloc line = {0}; strlist tokens = {0};
   while (outbuf_printf(&out, "%% ") >= 0 && outbuf_flush(&out) &&
         inbuf_sareadline(&in, &line)) {
      stralloc_0(&line); /* required by tokenizer() */
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      pid_t child = fork();
      if (child == -1) {
         print_error("fork"); continue;
      }
      if (child == 0) {
         // setup child and argv.list ...
         execvp(cmdname, argv.list);
         print_error(cmdname); exit(255);
      }

      /* wait for termination of child */
      // ...
   }
} // main
```

inbuf_sareadline.c

```
bool inbuf_sareadline(inbuf* ibuf, stralloc* sa) {
   sa->len = 0;
   for(;;) {
      int ch;
      if ((ch = inbuf_getchar(ibuf)) < 0) return false;
      if (ch == '\n') break;
      if (!stralloc_readyplus(sa, 1)) return false;
      sa->s[sa->len++] = ch;
   }
   return true;
}
```

- Diese Funktion erlaubt das Einlesen beliebig langer Zeilen auf Basis der *inbuf*-Bibliothek.
- Mit *stralloc_readyplus* wird jeweils Platz für mindestens ein weiteres Zeichen geschaffen.
- Die resultierende Zeichenkette ist *nicht* durch ein Nullbyte terminiert.

Erzeugung der Liste mit Kommandozeilenparametern 61

- Die Funktion *execl* ist für die *tinysh* ungeeignet, da die Zahl der Kommandozeilenparameter nicht feststeht. Diese soll auch nicht durch das Programm künstlich begrenzt werden.
- Alternativ zu *execl* gibt es *execv*, das einen Zeiger auf eine Liste mit Zeigern auf Zeichenketten erwartet, die am Ende mit einem Null-Zeiger abzuschliessen ist.
- Die in der *tinysh* verwendete Funktion *execvp* (mit zusätzlichem p) sucht im Gegensatz zu *execv* nach dem Programm in allen Verzeichnissen, die die Umgebungsvariable *PATH* aufzählt.

Erzeugung einer Liste mit Zeigern auf Zeichenketten 62

strlist.h

```
#ifndef STRLIST_H
#define STRLIST_H

#include <stddef.h>
#include <stdbool.h>

typedef struct strlist {
   char** list;
   size_t len; /* # of strings in list */
   size_t allocated; /* allocated length for list */
} strlist;

/* assure that there is at least room for len list entries */
bool strlist_ready(strlist* list, size_t len);

/* assure that there is room for len additional list entries */
bool strlist_readyplus(strlist* list, size_t len);

/* truncate the list to zero length */
void strlist_clear(strlist* list);

/* append the string pointer to the list */
bool strlist_push(strlist* list, char* string);
#define strlist_push0(list) strlist_push((list), 0)

/* free the strlist data structure but not the strings */
void strlist_free(strlist* list);

#endif
```

Erzeugung einer Liste mit Zeigern auf Zeichenketten 63

strlist.h

```
typedef struct strlist {
   char** list;
   size_t len; /* # of strings in list */
   size_t allocated; /* allocated length for list */
} strlist;

bool strlist_ready(strlist* list, size_t len);
bool strlist_readyplus(strlist* list, size_t len);
void strlist_clear(strlist* list);
bool strlist_push(strlist* list, char* string);
void strlist_free(strlist* list);
```

- Die *strlist*-Bibliothek folgt weitgehend dem Vorbild der *stralloc*-Bibliothek.

Erzeugung einer Liste mit Zeigern auf Zeichenketten 64

strlist.c

```
/* assure that there is at least room for len list entries */
bool strlist_ready(strlist* list, size_t len) {
   if (list->allocated < len) {
      size_t wanted = len + (len>>3) + 8;
      char** newlist = (char**) realloc(list->list,
         sizeof(char*) * wanted);
      if (newlist == 0) return false;
      list->list = newlist;
      list->allocated = wanted;
   }
   return true;
}

/* assure that there is room for len additional list entries */
bool strlist_readyplus(strlist* list, size_t len) {
   return strlist_ready(list, list->len + len);
}
```

Erzeugung einer Liste mit Zeigern auf Zeichenketten 65

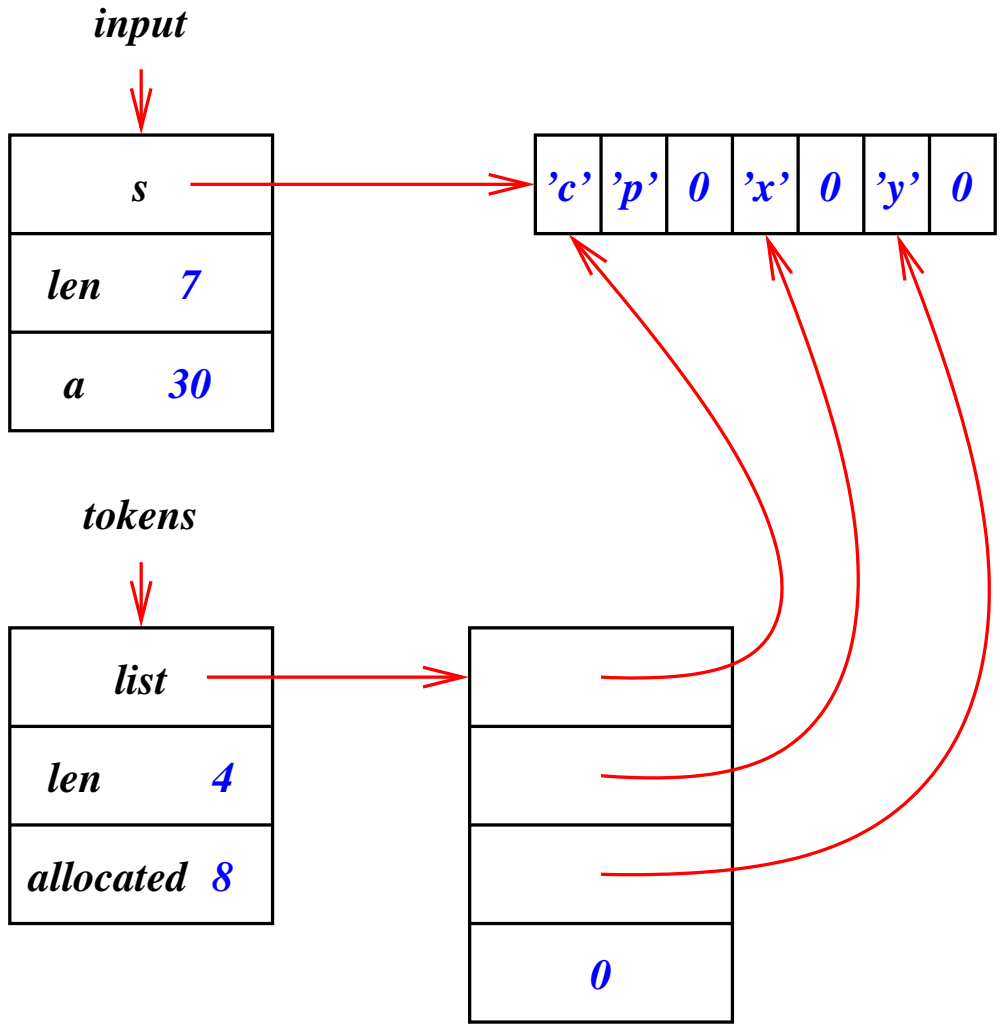
strlist.c

```
void strlist_clear(strlist* list) {
   list->len = 0;
}

/* append the string pointer to the list */
bool strlist_push(strlist* list, char* string) {
   if (!strlist_ready(list, list->len + 1)) return false;
   list->list[list->len++] = string;
   return true;
}

/* free the strlist data structure but not the strings */
void strlist_free(strlist* list) {
   free(list->list); list->list = 0;
   list->allocated = 0;
   list->len = 0;
}
```

tokenizer.h

```
#ifndef AFBLIB_TOKENIZER_H
#define AFBLIB_TOKENIZER_H

#include <stdbool.h>
#include <stralloc.h>
#include <afblib/strlist.h>
bool tokenizer(stralloc* input, strlist* tokens);

#endif
```

- Die Funktion *tokenizer* zerlegt die Eingabezeile in *input* in einzelne (durch Leerzeichen getrennte) Wörter und fügt diese in die Liste *tokens*.
- Wesentlich ist hier, dass die einzelnen Zeichenketten nicht dupliziert werden, sondern innerhalb der Eingabezeile verbleiben. Zu diesem Zweck werden Leerzeichen durch Nullbytes ersetzt.



- Das Diagramm zeigt die resultierende Datenstruktur des Wortzerlegers am Beispiel „cp x y“.

Zerlegung einer Eingabezeile

tokenizer.c

```
/*
 * Simple tokenizer: Take a 0-terminated stralloc object and return a
 * list of pointers in tokens that point to the individual tokens.
 * Whitespace is taken as token-separator and all whitespaces within
 * the input are replaced by null bytes.
 * afb 4/2003
 */

#include <ctype.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stralloc.h>
#include <afblib/strlist.h>
#include <afblib/tokenizer.h>

bool tokenizer(stralloc* input, strlist* tokens) {
   char* cp;
   int white = 1;

   strlist_clear(tokens);
   for (cp = input->s; *cp && cp < input->s + input->len; ++cp) {
      if (isspace((int) *cp)) {
         *cp = '\0'; white = 1; continue;
      }
      if (!white) continue;
      white = 0;
      if (!strlist_push(tokens, cp)) return false;
   }
   return true;
}
```

tinysh.c

```
while (outbuf_printf(&out, "%% ") >= 0 && outbuf_flush(&out) &&
       inbuf_sareadline(&in, &line)) {
   stralloc_0(&line); /* required by tokenizer() */
   if (!tokenizer(&line, &tokens)) break;
   if (tokens.len == 0) continue;
   // ...
}
```

- Da der Wortzerleger nullbyte-terminierte Zeichenketten liefert, muss mit *stralloc_0* noch ein Nullbyte angehängt werden.
- Falls keine Wörter zu finden sind, wird sofort die nächste Zeile eingelesen.
- Die Erzeugung der Kommandozeilenparameterliste wird dem neu zu erzeugenden Prozess überlassen.

Konfiguration des neuen Prozesses

tinysh.c

```
if (child == 0) {
   strlist argv = {0}; /* list of arguments */
   char* cmdname = 0; /* first argument */
   char* path; /* of output files */
   int oflags;
   for (int i = 0; i < tokens.len; ++i) {
      switch (tokens.list[i][0]) {
         case '<':
            fassign(0, &tokens.list[i][1], O_RDONLY, 0);
            break;
         case '>':
            path = &tokens.list[i][1];
            oflags = O_WRONLY|O_CREAT;
            if (*path == '>') {
               ++path; oflags |= O_APPEND;
            } else {
               oflags |= O_TRUNC;
            }
            fassign(1, path, oflags, 0666);
            break;
         default:
            strlist_push(&argv, tokens.list[i]);
            if (cmdname == 0) cmdname = tokens.list[i];
      }
   }
   if (cmdname == 0) exit(0);
   strlist_push0(&argv);
   execvp(cmdname, argv.list);
   print_error(cmdname);
   exit(255);
}
```

tinysh.c

```
/* assign an opened file with the given flags and mode to fd */
void fassign(int fd, char* path, int oflags, mode_t mode) {
   int newfd = open(path, oflags, mode);
   if (newfd < 0) {
      print_error(path); exit(255);
   }
   if (dup2(newfd, fd) < 0) {
      print_error("dup2"); exit(255);
   }
   close(newfd);
}
```

- Mit dem Systemaufruf *dup2* lässt sich ein Dateideskriptor auf einen gegebenen anderen Deskriptor duplizieren, die dann beide auf den gleichen Eintrag in der *Open File Table* verweisen.
- So lassen sich neu eröffnete Datei-Verbindungen mit vorgegebenen Dateideskriptoren wie etwa 0 (stdin) oder 1 (stdout) verknüpfen.

Signale werden für vielfältige Zwecke eingesetzt. Sie können verwendet werden,

- um den normalen Ablauf eines Prozesses für einen wichtigen Hinweis zu unterbrechen,
- um die Ausführung eines Prozesses zu suspendieren,
- um die Terminierung eines Prozesses zu erbitten oder zu erzwingen und
- um schwerwiegende Fehler bei der Ausführung zu behandeln wie z.B. den Verweis durch einen invaliden Zeiger.

- Signale sind unter UNIX die einzige Möglichkeit, den normalen Programmablauf eines Prozesses zu unterbrechen.
- Signale werden durch kleine natürliche Zahlen repräsentiert, die in jeder UNIX-Umgebung fest vordefiniert sind.
- Darüber hinaus stehen kaum weitere Informationen zur Verfügung. Signale ersetzen daher keine Interprozeßkommunikation.
- Signale können von verschiedenen Parteien ausgelöst werden: Von anderen Prozessen, die die dafür notwendige Berechtigung haben (entweder der gleiche Benutzer oder der Super-User), durch den Prozess selbst entweder indirekt (durch einen schwerwiegenden Fehler) oder explizit oder auch durch das Betriebssystem.

- Der ISO-Standard 9899-2011 für die Programmiersprache C definiert eine einfache und damit recht portable Schnittstelle für die Behandlung von Signalen. Hier gibt es neben der Signalnummer selbst keine weiteren Informationen.
- Der IEEE Standard 1003.1 (POSIX) bietet eine Obermenge der Schnittstelle des ISO-Standards an, bei der wenige zusätzliche Informationen (wie z.B. die Angabe des invaliden Zeigers) dabei sein können und der insbesondere eine sehr viel feinere Kontrolle der Signalbehandlung erlaubt.

Die Terminalschnittstelle unter UNIX wurde ursprünglich für ASCII-Terminals mit serieller Schnittstelle entwickelt, die nur folgende Eingabemöglichkeiten anboten:

- Einzelne ASCII-Zeichen, jeweils ein Byte (zusammen mit etwas Extra-Kodierung wie Prüf- und Stop-Bits).
- Ein BREAK, das als spezielles Signal repräsentiert wird, das länger als die Kodierung für ein ASCII-Zeichen währt.
- Ein HANGUP, bei dem ein Signal wegfällt, das zuvor die Existenz der Leitung bestätigt hat. Dies benötigt einen weiteren Draht in der seriellen Leitung.

Diese Eingaben werden auf der Seite des Betriebssystems vom Terminal-Treiber bearbeitet, der in Abhängigkeit von den getroffenen Einstellungen

- die eingegebenen Zeichen puffert und das Editieren der Eingabe ermöglicht (beispielsweise mittels BACKSPACE, CTRL-u und CTRL-w) und
- bei besonderen Eingaben Signale an alle Prozesse schickt, die mit diesem Terminal verbunden sind.

Ziel war es, dass im Normalfall ein BREAK zu dem Abbruch oder zumindest der Unterbrechung der gerade laufenden Anwendung führt. Und ein HANGUP sollte zu dem Abbruch der gesamten Sitzung führen, da bei einem Wegfall der Leitung keine Möglichkeit eines regulären Abmeldens besteht.

Heute sind serielle Terminals rar geworden, aber das Konzept wurde dennoch beibehalten:

- Zwischen einem virtuellen Terminal (beispielsweise einem xterm) und den Prozessen, die zur zugehörigen Sitzung gehören, ist ein sogenanntes Pseudo-Terminal im Betriebssystem geschaltet, das der Sitzung die Verwendung eines klassischen Terminals vorspielt.
- Da es BREAK in diesem Umfeld nicht mehr gibt, wird es durch ein beliebiges Zeichen ersetzt wie beispielsweise CTRL-c.
- Wenn das virtuelle Terminal wegfällt (z.B. durch eine gewaltsame Beendigung der xterm-Anwendung), dann gibt es weiterhin ein HANGUP für die Sitzung.

- Auf fast alle Signale können Prozesse, die sie erhalten, auf dreierlei Weise reagieren:
  - Voreinstellung: Normalerweise die Terminierung des Prozesses. (*SIG_DFL*)
  - Ignorieren. (*SIG_IGN*)
  - Bearbeitung durch einen Signalbehandler.
- Es mag harsch erscheinen, dass die Voreinstellung fast durchweg zur Terminierung eines Prozesses führt. Aber genau dies führt bei normalen Anwendungen genau zu den gewünschten Effekten wie Abbruch des laufenden Programms bei BREAK (die Shell ignoriert das Signal) und Abbau der Sitzung bei HANGUP.
- Wenn ein Prozess diese Signale ignoriert, sollte es genau wissen, was es tut, da der Nutzer auf diese Weise eine wichtige Kontrollmöglichkeit seiner Sitzung verliert.

sigint.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

volatile sig_atomic_t signal_caught = 0;

void signal_handler(int signal) {
   signal_caught = signal;
}

int main() {
   if (signal(SIGINT, signal_handler) == SIG_ERR) {
      perror("unable to setup signal handler for SIGINT");
      exit(1);
   }
   printf("Try to send a SIGINT signal!\n");
   int counter = 0;
   while (!signal_caught) {
      for (int i = 0; i < counter; ++i);
      ++counter;
   }
   printf("Got signal %d after %d steps!\n", signal_caught, counter);
}
```

- Dieses Beispiel demonstriert die Behandlung des Signals *SIGINT*, das dem BREAK entspricht.

sigint.c

```
volatile sig_atomic_t signal_caught = 0;

void signal_handler(int signal) {
   signal_caught = signal;
}
```

- Die Deklaration für *signal_caught* wird noch genauer diskutiert. Zunächst kann davon ausgegangen werden, dass es sich dabei um eine globale ganzzahlige Variable handelt, die zu Beginn mit 0 initialisiert wird.
- Die Funktion *signal_handler* ist ein Signalbehandler. Als einziges Argument erhält sie die Nummer des eingetroffenen Signals, das es zu behandeln gilt. Einen Rückgabewert gibt es nicht.

sigint.c

```
if (signal(SIGINT, signal_handler) == SIG_ERR) {
   perror("unable to setup signal handler for SIGINT");
   exit(1);
}
```

- Mit der Funktion *signal* kann für eine Signalnummer (hier *SIGINT*) ein Signalbehandler (hier *signal_handler*) spezifiziert werden.
- Wenn die Operation erfolgreich war, wird der zuletzt eingesetzte Signalbehandler zurückgeliefert.
- Im Fehlerfall liefert *signal* den Wert *SIG_ERR*. (Damit ist normalerweise nicht zu rechnen, es sei denn, es werden nicht zulässige Einstellungen vorgenommen, wie etwa das Ignorieren von *SIG_KILL*.)

sigint.c

```
printf("Try to send a SIGINT signal!\n");
int counter = 0;
while (!signal_caught) {
   for (int i = 0; i < counter; ++i);
   ++counter;
}
printf("Got signal %d after %d steps!\n", signal_caught, counter);
```

- Das Hauptprogramm arbeitet eine Endlosschleife ab, die nur beendet werden kann, wenn auf „magische“ Weise die Variable *signal_caught* einen Wert ungleich 0 erhält.

sigint.c

```
while (!signal_caught) {
   for (int i = 0; i < counter; ++i);
   ++counter;
}
```

- Wenn ein optimierender Übersetzer die Schleife analysiert, könnten folgende Punkte auffallen:
  - Die Schleife ruft keine externen Funktionen auf.
  - Innerhalb der Schleife wird *signal_caught* nirgends verändert.
- Daraus könnte vom Übersetzer der Schluss gezogen werden, dass die Schleifenbedingung nur zu Beginn einmal überprüft werden muss. Findet der Eintritt in die Schleife statt, könnte der weitere Test der Bedingung ersatzlos wegfallen.
- Analysen wie diese sind für heutige optimierende Übersetzer Pflicht, um guten Maschinen-Code erzeugen zu können.
- Es wäre fatal, wenn darauf nur wegen der Existenz von asynchron aufgerufenen Signalbehandlern verzichtet werden würde.

sigint.c

```
volatile sig_atomic_t signal_caught = 0;
```

- Um beides zu haben, die fortgeschrittenen Optimierungstechniken und die Möglichkeit, Variablen innerhalb von Signalbehandlern setzen zu können, wurde in C die Speicherklasse **volatile** eingeführt.
- Damit lassen sich Variablen kennzeichnen, deren Wert sich jederzeit ändern kann — selbst dann, wenn dies aus dem vorliegenden Programmtext nicht ersichtlich ist.
- Entsprechend gilt dann auch in C, dass alle anderen Variablen, die nicht als **volatile** klassifiziert sind, sich nicht durch „magische“ Effekte verändern dürfen.

Damit die Effekte eines Signalbehandlers wohldefiniert sind, schränken sich die Möglichkeiten stark ein. So ist es nur zulässig,

- lokale Variablen zu verwenden,
- mit **volatile** deklarierte Variablen zu benutzen und
- Funktionen aufzurufen, die sich an die gleichen Spielregeln halten.

- Die Verwendung von Ein- und Ausgabe innerhalb eines Signalbehandlers ist nicht zulässig.
- Der ISO-Standard 9899-2011 nennt nur *abort*(), *_Exit*(), *quick_exit*() und *signal*() als zulässige Bibliotheksfunktionen.
- Beim POSIX-Standard werden noch zahlreiche weitere Systemaufrufe genannt.
- Auf den Manualseiten von Solaris wird dies dokumentiert durch die Angabe „Async-Signal-Safe“ bei „MT-Level“.
- Ansonsten ist nach expliziten Hinweisen zu suchen, ob eine Funktion mehrfach parallel ausgeführt werden darf, d.h. ob sie *reentrant* ist.
- Beispiele von Funktionen der Standard-Bibliothek, die nicht *reentrant* sind: *ctime* und *strtok*. Hier gibt es die alternativen Fassungen *ctime_r* und *strtok_r*, die *reentrant* sind.

- Variablenzugriffe sind nicht notwendigerweise atomar.
- Das hat zur Konsequenz, dass eine unterbrochende Variablenzuweisung möglicherweise nur teilweise durchgeführt worden ist. Auf einer 32-Bit-Maschine mit einem 32 Bit breiten Datenbus wäre es etwa denkbar, dass eine 64-Bit-Größe (etwa **long long** oder **double**) nur zur Hälfte kopiert ist, wenn eine Unterbrechung eintrifft.
- Dies bedeutet, dass im Falle einer Unterbrechung eine Variable nicht nur einen alten oder neuen Wert haben kann, sondern auch einen undefinierten.
- Um solche Probleme auszuschließen, bietet der ISO-Standard 9899-1999 den ganzzahligen Datentyp *sig_atomic_t* an, der in <*signal.h*> definiert ist.
- Bei Zugriffen auf Variablen dieses Typs wird im Falle einer Unterbrechung nur der alte oder der neue Wert beobachtet, jedoch nie ein undefinierter.
- *sig_atomic_t* wird typischerweise in Kombination mit **volatile** verwendet.

Wecksignale mit *alarm*

sigalrm.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static volatile sig_atomic_t time_exceeded = 0;

static void alarm_handler(int signal) {
   time_exceeded = 1;
}

int main() {
   if (signal(SIGALRM, alarm_handler) == SIG_ERR) {
      perror("unable to setup signal handler for SIGALRM");
      exit(1);
   }
   alarm(2);
   puts("Na, koennen Sie innerhalb von zwei Sekunden etwas eingeben?");
   int ch = getchar();
   if (time_exceeded) {
      puts("Das war wohl nichts.");
   } else {
      puts("Gut!");
   }
}
```

sigalrm.c

```
if (signal(SIGALRM, alarm_handler) == SIG_ERR) {
   perror("unable to setup signal handler for SIGALRM");
   exit(1);
}
alarm(2);
```

- Für jeden Prozess verwaltet UNIX einen Wecker, der entweder ruht oder zu einem spezifizierten Zeitpunkt sich mit dem Signal *SIGALRM* meldet.
- Der Wecker wird mit *alarm* gestellt. Dabei wird die zu verstreichende Zeit in Sekunden angegeben.
- Mit einer Angabe von 0 lässt sich der Wecker ausschalten.

tread.h

```
#ifndef TREAD_H
#define TREAD_H

#include <unistd.h>

ssize_t timed_read(int fd, void* buf, size_t nbytes, unsigned seconds);

#endif
```

- Mit Hilfe des Weckers lässt sich der Systemaufruf *read* zu *timed_read* erweitern, das ein Zeitlimit berücksichtigt.
- Falls das Zeitlimit erreicht wird, ist kein Fehler, sondern es wird ganz schlicht 0 zurückzugeben.
- Wie bereits beim vorherigen Beispiel wird hier ausgenutzt, dass nicht nur normale Programmabläufe, sondern auch einige Systemaufrufe wie etwa *read* unterbrechbar sind.

tread.c

```
#include <signal.h>
#include <unistd.h>
#include "tread.h"

static volatile sig_atomic_t time_exceeded = 0;

static void alarm_handler(int signal) {
   time_exceeded = 1;
}
```

- Der Signalbehandler für *SIGALRM* arbeitet wie gehabt. Allerdings wird im Unterschied zu zuvor die Variable und der Behandler **static** deklariert, damit diese Deklarationen privat bleiben und nicht in Konflikt zu anderen Deklarationen stehen.

Lese-Operationen mit Zeitlimit

tread.c

```
ssize_t timed_read(int fd, void* buf, size_t nbytes, unsigned seconds) {
   if (seconds == 0) return 0;
   /* setup signal handler and alarm clock but
       remember the previous settings */
   void (*previous_handler)(int) = signal(SIGALRM, alarm_handler);
   if (previous_handler == SIG_ERR) return -1;
   time_exceeded = 0;
   int remaining_seconds = alarm(seconds);
   if (remaining_seconds > 0) {
      if (remaining_seconds <= seconds) {
         remaining_seconds = 1;
      } else {
         remaining_seconds -= seconds;
      }
   }

   ssize_t bytes_read = read(fd, buf, nbytes);

   /* restore previous settings */
   if (!time_exceeded) alarm(0);
   signal(SIGALRM, previous_handler);
   if (remaining_seconds) alarm(remaining_seconds);

   if (bytes_read < 0 && time_exceeded) return 0;
   return bytes_read;
}
```

tread.c

```
void (*previous_handler)(int) = signal(SIGALRM, alarm_handler);
```

- Aus der Sicht einer Bibliotheksfunktion muss damit gerechnet werden, dass auch noch andere Parteien einen Wecker benötigen und deswegen *alarm* aufrufen.
- Deswegen ist es sinnvoll, die eigene Nutzung so zu gestalten, dass die Weckfunktion für die anderen nicht sabotiert wird.
- Dies ist prinzipiell möglich, weil *signal* den gerade eingesetzten Signalbehandler im Erfolgsfalle zurückliefert. Dieser wird hier der Variablen *previous_handler* zugewiesen.

tread.c

```
time_exceeded = 0;
int remaining_seconds = alarm(seconds);
if (remaining_seconds > 0) {
   if (remaining_seconds <= seconds) {
      remaining_seconds = 1;
   } else {
      remaining_seconds -= seconds;
   }
}
```

- Die gleiche Rücksichtnahme erfolgt bei dem Aufruf von *alarm*.
- Im Erfolgsfalle liefert *alarm* den Wert 0, falls zuvor der Wecker ruhte oder einen positiven Wert, der die zuvor noch verbliebenen Sekunden bis zum Signal spezifiziert.
- Die Variable *remaining_seconds* wird auf den Wert gesetzt, den wir abschließend verwenden, um den Wecker neu zu stellen, nachdem er in dieser Funktion nicht mehr benötigt wird.

- *read* hat in diesem Szenario verschiedene Möglichkeiten, zurückzukommen. Erstens kann *read* ganz normal etwas einlesen (positiver Rückgabewert), es kann ein Eingabeende vorliegen (Rückgabewert gleich 0) oder es kann ein Fehler eintreten (negativer Rückgabewert).
- Im Falle einer Unterbrechung durch ein Signal bricht der Systemaufruf mit einem Fehler ab, d.h. es wird -1 zurückgeliefert. Die Variable *errno* hat dann den Wert *EINTR*.
- Wenn *read* unterbrochen wird und mit -1 endet, wurde nichts weggelesen. Ein unterbrochener *write*-Systemaufruf, der -1 liefert, hat nichts geschrieben. Wenn *read* bzw. *write* bereits gelesen bzw. geschrieben haben, wenn sie unterbrochen werden, dann liefern sie nicht -1, sondern die Zahl der bereits gelesenen bzw. geschriebenen Bytes zurück.
- In diesem Beispiel wird jedoch nicht *errno* überprüft, sondern die Variable *time_exceeded* untersucht.

Fenster schließen bei Signalen

tread.c

```
ssize_t bytes_read = read(fd, buf, nbytes);

/* restore previous settings */
if (!time_exceeded) alarm(0);
signal(SIGALRM, previous_handler);
if (remaining_seconds) alarm(remaining_seconds);

if (bytes_read < 0 && time_exceeded) return 0;
return bytes_read;
```

- Bevor *alarm* erneut aufgesetzt wird, muss zuvor der alte Signalbehandler restauriert werden.
- Wenn dies in umgekehrter Reihenfolge geschehen würde, dann gibt es ein kleines Zeitfenster, in dem das Signal *SIGALRM* eintreffen könnte, noch bevor es zum Aufruf von *signal* kam.
- In diesem Falle würde der andere Signalbehandler nicht wie geplant aufgerufen werden.
- Daher wird hier zuerst der alte Signalbehandler eingesetzt, bevor *alarm* aufgerufen wird. Auf diese Weise wird das Fenster geschlossen.

In Erweiterung zu *alarm* stehen die Funktionen der POSIX-Realtime-Erweiterung optional zur Verfügung. Hierzu gehören insbesondere *timer_create* und *timer_set*:

- Die Timer sind dann nicht nur sekundengenau, sondern können mit der maximal auf der Plattform erreichbaren Genauigkeit spezifiziert werden.
- Beliebig viele Timer können parallel laufen.
- Periodisch sich wiederholende Signale können konfiguriert werden.

- Grundsätzlich kann ein Prozess einem anderen Prozess (einschliesslich sich selbst) ein Signal senden.
- Voraussetzung ist dabei unter UNIX, dass der andere Prozess dem gleichen Benutzer gehört oder der das Signal versendende Prozess mit Superuser-Privilegien arbeitet.
- Der ISO-Standard für C sieht zum Signalversand nur eine Funktion *raise*() vor, die es erlaubt, ein Signal an den eigenen Prozess zu versenden.
- Im POSIX-Standard kommt der Systemaufruf *kill*() hinzu, der es erlaubt, ein Signal an einen anderen Prozess zu verschicken, sofern die dafür notwendigen Privilegien vorliegen.

killparent.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void sigterm_handler(int signo) {
   const char msg[] = "Goodbye, cruel world!\n";
   write(1, msg, sizeof msg - 1);
   _Exit(1);
}

int main() {
   if (signal(SIGTERM, sigterm_handler) == SIG_ERR) {
      perror("signal"); exit(1);
   }

   pid_t child = fork();
   if (child == 0) {
      kill(getppid(), SIGTERM);
      exit(0);
   }
   int wstat;
   wait(&wstat);
   exit(0);
}
```

killparent.c

```
kill(getppid(), SIGTERM);
```

- Der Systemaufruf *kill* benötigt zwei Parameter, wobei der erste die Prozess-ID des Signalempfängers und der zweite Parameter das zu versendende Signal nennt.
- Das Versenden von *SIGTERM* gilt per Konvention als „freundliche“ Bitte, den Prozess zu terminieren.
- Der Empfänger erhält so die Gelegenheit, Aufräumarbeiten vorzunehmen, bevor er abschließt.
- Alternativ zu *SIGTERM* gibt es auch *SIGKILL*, das sich nicht behandeln lässt, d.h. dass der Empfänger unter keinen Umständen mehr zum Zuge kommt.

killparent.c

```
void sigterm_handler(int signo) {
   const char msg[] = "Goodbye, cruel world!\n";
   write(1, msg, sizeof msg - 1);
   _Exit(1);
}
```

- Hier ist vorgesehen, dass der Signalbehandler im Falle von *SIGTERM* noch eine Meldung ausgibt, bevor der Prozess terminiert wird.
- Da die Verwendung von Funktionen der *stdio* wie etwa *puts* innerhalb von Signalbehandlern tabu ist, wird hier der Systemaufruf *write* verwendet.
- Ebenfalls tabu ist *exit*, da dabei Funktionen der *stdio* zur Leerung aller Puffer aufgerufen werden.
- Alternativ kann die Funktion *_Exit* aufgerufen werden, die mit dem ISO-Standard 9899-1999 eingeführt wurde. Diese umgeht sämtliche Aufräumarbeiten und terminiert unmittelbar den aufrufenden Prozess.

- Der Systemaufruf *kill*() erfüllt aber auch noch einen weiteren Zweck. Bei einer Signalnummer von 0 wird nur die Zulässigkeit des Signalversendens überprüft.
- Dies kann dazu ausgenutzt werden, um die Existenz eines Prozesses zu überprüfen.
- Mit folgenden Fehler-Codes ist dabei zu rechnen:
  - *ESRCH*: Die genannte Prozess-ID ist zur Zeit nicht vergeben.
  - *EPERM*: Die genannte Prozess-ID existiert, aber es fehlen die Privilegien, dem Prozess ein Signal zu senden.

waitfor.c

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s pid\n", cmdname);
      exit(1);
   }

   /* convert first argument to pid */
   char* endptr = 0;
   pid_t pid = strtol(argv[0], &endptr, 0);
   if (*endptr || pid < 0) {
      fprintf(stderr, "%s: unsigned integer expected as argument: %s\n",
         cmdname, argv[0]);
      exit(1);
   }

   while (kill(pid, 0) == 0) sleep(1);

   if (errno == ESRCH) exit(0);
   perror(cmdname); exit(1);
}
```

- Gelegentlich kommt es vor, dass Prozesse nur auf das Eintreffen eines Signals warten möchten und sonst nichts zu tun haben.
- Theoretisch könnte ein Prozess dann in eine Dauerschleife mit leerem Inhalt treten (auch als *busy loop* bezeichnet).
- Dies wäre jedoch nicht sehr fair auf einem System mit mehreren Prozessen, da dadurch Rechenzeit vergeudet würde.
- Abhilfe schafft hier der Systemaufruf *pause*(), der einen Prozess schlafen legt, bis ein Signal eintrifft.

Auf eintreffende Signale warten

pingpong.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static volatile sig_atomic_t ball_count = 0;

void sighandler(int sig) {
   ++ball_count;
   if (signal(sig, sighandler) == SIG_ERR) _Exit(1);
}

int main() {
   /* this signal setting is inherited to our child */
   if (signal(SIGUSR1, sighandler) == SIG_ERR) {
      perror("signal SIGUSR1"); exit(1);
   }

   pid_t parent = getpid();
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      ball_count = 1; /* give the ball to the child... */
      playwith(parent);
   } else {
      playwith(child);
   }
}
```

pingpong.c

```
static void playwith(pid_t partner) {
   for(int i = 0; i < 10; ++i) {
      if (!ball_count) pause();
      printf("[%d] send signal to %d\n",
         (int) getpid(), (int) partner);
      if (kill(partner, SIGUSR1) < 0) {
         printf("[%d] %d is no longer alive\n",
            (int) getpid(), (int) partner);
         break;
      }
      --ball_count;
   }
   printf("[%d] finishes playing\n", (int) getpid());
}
```

- Mit *pause* wartet der aufrufende Prozess bis zum Eintreffen eines Signals. Wenn dieser Systemaufruf beendet wird, ist das Resultat immer negativ und *errno* ist auf *EINTR* gesetzt.

Auf eintreffende Signale warten

pingpong.c

```
static volatile sig_atomic_t ball_count = 0;
void sighandler(int sig) {
   ++ball_count;
   if (signal(sig, sighandler) == SIG_ERR) _Exit(1);
}

   /* ... */
   if (signal(SIGUSR1, sighandler) == SIG_ERR) {
      perror("signal SIGUSR1"); exit(1);
   }
   /* ... */
```

- *SIGUSR1* gehört zusammen mit *SIGUSR2* zu den Signalen ohne Sonderbedeutung, die problemlos für Zwecke der Prozesskommunikation verwendet werden können.
- Wenn *sighandler* noch vor *fork* als Signalbehandler installiert wird, dann erbt auch der neu erzeugte Prozess diese Einstellung.
- *sighandler* installiert sich selbst erneut, da der ISO-Standard 9899-2011 offen lässt, ob der Signalbehandler nach dem Eintreffen des Signals installiert bleibt oder nicht.

Die vorangegangenen Beispiele werfen die Frage auf, wie UNIX bei der Zustellung von Signalen vorgeht, wenn

- der Prozess zur Zeit nicht aktiv ist,
- gerade ein Systemaufruf für den Prozess abgearbeitet wird oder
- gerade ein Signalbehandler bereits aktiv ist.

Vom ISO-Standard 9899-2011 für C wird in dieser Beziehung nichts festgelegt.

Der POSIX-Standard geht jedoch genauer darauf ein:

- Wenn ein Prozess ein Signal erhält, wird dieses Signal zunächst in den zugehörigen Verwaltungsstrukturen des Betriebssystems vermerkt. Signale, die für einen Prozess vermerkt sind, jedoch noch nicht zugestellt worden sind, werden als *anhängige* Signale bezeichnet.
- Wenn mehrere Signale mit der gleichen Nummer anhängig sind, ist nicht festgelegt, ob eine Mehrfachzustellung erfolgt. Es können also Signale wegfallen.
- Nur aktiv laufende Prozesse können Signale empfangen. Prozesse werden normalerweise durch die Existenz eines anhängigen Signals aktiv — aber dieses kann auch längere Zeit in Anspruch nehmen, wenn dem zwischenzeitlich mangelnde Ressourcen entgegenstehen.
- Für jeden Prozess gibt es eine Menge blockierter Signale, die im Augenblick nicht zugestellt werden sollen. Dies hat nichts mit dem Ignorieren von Signalen zu tun, da blockierte Signale anhängig bleiben, bis die Blockierung aufgehoben wird.

- Der POSIX-Standard legt nicht fest, was mit der Signalbehandlung geschieht, wenn ein Signalbehandler aufgerufen wird.
- Möglich ist das Zurückfallen auf *SIG_DFL* (Voreinstellung mit Prozessterminierung) oder die temporäre automatische Blockierung des Signals bis zur Beendigung des Signalbehandlers.
- Alle modernen UNIX-Systeme wählen die zweite Variante.
- Dies lässt sich aber gemäß dem POSIX-Standard auch erzwingen, indem die umfangreichere Schnittstelle *sigaction*() anstelle von *signal*() verwendet wird. Allerdings ist *sigaction*() nicht mehr Bestandteil des ISO-Standards für C.

- UNIX unterscheidet zwischen unterbrechbaren und unterbrechungsfreien Systemaufrufen. Zur ersteren Kategorie gehören weitgehend alle Systemaufrufe, die zu einer längeren Blockierung eines Prozesses führen können.
- Ist ein nicht blockiertes Signal anhängig, kann ein unterbrechbarer Systemaufruf aufgrund des Signals mit einer Fehlerindikation beendet werden. *errno* wird dann auf *EINTR* gesetzt.
- Dabei ist zu beachten, dass der unterbrochene Systemaufruf nach Beendigung der Signalbehandlung normalerweise *nicht* fortgesetzt wird, sondern manuell erneut gestartet werden muss.
- Dies kann leider zu unerwarteten Überraschungseffekten führen, weil insbesondere auch die *stdio*-Bibliothek keinerlei Vorkehrungen trifft, Systemaufrufe automatisch erneut aufzusetzen, falls es zu einer Unterbrechung kam.

Quote from The Rise of "Worse is Better" by Richard Gabriel:

> *Two famous people, one from MIT and another from Berkeley (but working on Unix) once met to discuss operating system issues. The person from MIT was knowledgeable about ITS (the MIT AI Lab operating system) and had been reading the Unix sources. He was interested in how Unix solved the PC loser-ing problem. The PC loser-ing problem occurs when a user program invokes a system routine to perform a lengthy operation that might have significant state, such as IO buffers. If an interrupt occurs during the operation, the state of the user program must be saved. Because the invocation of the system routine is usually a single instruction, the PC of the user program does not adequately capture the state of the process. The system routine must either back out or press forward. The right thing is to back out and restore the user program PC to the instruction that invoked the system routine so that resumption of the user program after the interrupt, for example, re-enters the system routine.*

It is called "PC loser-ing" because the PC is being coerced into "loser mode," where "loser" is the affectionate name for "user" at MIT.

The MIT guy did not see any code that handled this case and asked the New Jersey guy how the problem was handled. The New Jersey guy said that the Unix folks were aware of the problem, but the solution was for the system routine to always finish, but sometimes an error code would be returned that signaled that the system routine had failed to complete its action. A correct user program, then, had to check the error code to determine whether to simply try the system routine again. The MIT guy did not like this solution because it was not the right thing.

Die beiden Ansätze im Vergleich:

- *PC loser-ing*: Bei der Unterbrechung eines Systemaufrufs wird dieser wie bei einer abgebrochenen Transaktion zum Zeitpunkt des Aufrufs zurückgerollt, d.h. der Zustand vor dem Aufruf wird wiederhergestellt. Der Kontext des Benutzerprozesses wird auf die Instruktion gesetzt, die zum Systemaufruf führte.
- Ansatz von UNIX: Der Systemaufruf wird bei einer Unterbrechung immer beendet. Teilweise mit der Rückgabe eines Fehlercodes (*EINTR*) oder durch eine teilweise Umsetzung (etwa bei *read* oder *write*). Dies zwingt den Programmierer, ggf. bei *EINTR* den Systemaufruf manuell neu zu starten.

Wie behandelt UNIX dies heute:

- Nach wie vor ist der historische UNIX-Ansatz die Voreinstellung.
- Die *sigaction*-Schnittstelle ermöglicht optional (Flag *SA_RESTART*) den automatisierten Neustart eines Systemaufrufs, wo sonst *EINTR* zurückgeliefert wird.

Einerseits wird *EINTR* durchaus benötigt, da ein Abbruch durchaus ein erwünschtes Verhalten sein kann. Andererseits treffen mit *EINTR* unterbrochene Systemaufrufe möglicherweise auch Bibliotheken, die nicht dafür vorbereitet sind. Nicht immer lässt sich das leicht lösen. Die *stdio* bietet beispielsweise keine Vorkehrungen dazu.

- Für die genauere Regulierung der Signalbehandlung bietet POSIX (jedoch nicht ISO-C) den Systemaufruf *sigaction* an. Während bei *signal* zur Spezifikation der Signalbehandlung nur ein Funktionszeiger genügte, kommen bei der **struct** *sigaction*, die *sigaction*() verwendet, die in der folgenden Tabelle genannten Felder zum Einsatz:

Datentyp	Feldname	Beschreibung
void(∗)(**int**)	*sa_handler*	Funktionszeiger (wie bisher)
void(∗)(**int**, *siginfo_t*∗, **void**∗)	*sa_sigaction*	alternativer Zeiger auf einen Signalbehandler, der mehr Informationen zum Signal erhält
sigset_t	*sa_mask*	Menge von Signalen, die während der Signalbehandlung dieses Signals zu blockieren sind
int	*sa_flags*	Menge von Boolean-wertigen Optionen

Folgende Optionen werden im Rahmen des aktuellen POSIX-Standards unterstützt:

SA_NOCLDSTOP	*SIGCHLD* ist nicht zu generieren, wenn Kindprozesse stoppen oder Kindprozesse fortgesetzt werden
SA_ONSTACK	wenn ein alternativer Stack mit *sigaltstack* für Signalbehandlungen spezifiziert wurde, ist dieser zu verwenden
SA_RESETHAND	nach der Signalbehandlung wird die Einstellung für das Signal auf *SIG_DFL* zurückgesetzt
SA_RESTART	unterbrochene Systemaufrufe geben nicht *EINTR* zurück und werden stattdessen neu gestartet
SA_SIGINFO	dem Signalbehandler werden weitere Infos zur Verfügung gestellt
SA_NOCLDWAIT	bei *SIGCHLD* bedeutet dies, dass der Exit-Status der Kindprozesse sofort entsorgt wird
SA_NODEFER	während des Aufrufs des Signalbehandlers wird das eingetroffene Signal blockiert

Die *sigaction*-Schnittstelle: *SA_RESTART*

restart.c

```
volatile sig_atomic_t gotit = 0;
void sighandler(int sig) {
   char msg[] = "I am still waiting for input...\n";
   if (!gotit) {
      write(1, msg, sizeof msg - 1);
      alarm(1);
   }
}

int main() {
   struct sigaction sigact = {
      .sa_handler = sighandler,
      .sa_flags = SA_RESTART,
   };
   if (sigaction(SIGALRM, &sigact, 0) < 0) {
      exit(1);
   }
   char msg[] = "Please type in something.\n";
   write(1, msg, sizeof msg - 1);
   alarm(1);
   char inbuf[32];
   ssize_t nbytes = read(0, inbuf, sizeof inbuf);
   gotit = 1;
   if (nbytes > 0) {
      char thanks[] = "Thanks!\n";
      write(1, thanks, sizeof thanks - 1);
   } else {
      char ooh[] = "Ooh :-(\n";
      write(1, ooh, sizeof ooh - 1);
   }
}
```

strikeback.c

```
volatile int signo = 0;
volatile pid_t pid = 0;

void sighandler(int sig, siginfo_t* siginfo, void* context) {
   signo = sig;
   pid = siginfo->si_pid;
   if (pid) { /* strike back */
      kill(pid, sig);
   }
}

int main() {
   int signals[] = {SIGHUP, SIGINT, SIGTERM, SIGUSR1, SIGUSR2};
   struct sigaction sigact = {
      .sa_sigaction = sighandler,
      .sa_flags = SA_SIGINFO,
   };
   for (int index = 0; index < sizeof(signals)/sizeof(int); ++index) {
      signo = signals[index];
      if (sigaction(signo, &sigact, 0) < 0) {
         perror("sigaction"); exit(1);
      }
   }
   for(;;) {
      pause();
      if (signo) {
         printf("got signal %d from %d\n", signo, (int) pid); fflush(stdout);
      }
   }
}
```

Die *sigaction*-Schnittstelle: *SA_SIGINFO*

strikeback.c

```
volatile int signo = 0;
volatile pid_t pid = 0;

void sighandler(int sig, siginfo_t* siginfo, void* context) {
   signo = sig;
   pid = siginfo->si_pid;
   if (pid) {
      /* strike back */
      kill(pid, sig);
   }
}
```

- Wenn die *SA_SIGINFO*-Option aktiviert wird, dann erhält der Signalbehandler zwei weitere Parameter:

siginfo_t siginfo*	standardisierte **struct** mit einer Reihe von Zusatzinfos (signalabhängig)
void* *context*	eine implementierungsabhängige Repräsentierung des gesicherten Kontexts

Im POSIX-Standard genannte Komponenten der *siginfo_t*-Datenstruktur:

int *si_signo*	Signalnummer (redundant)
int *si_code*	Nummer, die die Ursache identifiziert, bei *SI_NOINFO* keine weiteren Infos
int *si_errno*	*errno* in Verbindung mit dem Signal, falls ungleich 0
int *si_pid*	Prozess, der das Signal gesendet hat
int *si_uid*	Reale Benutzer-ID des sendenden Prozesses
void* *si_addr*	Adresse bei *SIGSEGV* oder *SIGBUS*
int *si_status*	Status bei *SIGCHLD*

- Bei der *sigaction*-Schnittstelle ist es möglich, die Zustellung einiger Signale aufzuhalten während einer Signalbehandlung.
- Dies betrifft implizit das gerade empfangene Signal und auch mögliche weitere Signale. Letzteres wird über das Feld *sa_mask* spezifiziert.
- Blockierte Signale sind dann zunächst anhängig und warten dann darauf, dass der Block aufgehoben wird.
- Wenn mehrfach das gleiche blockierte Signal eintrifft, dann ist nicht definiert, ob dies auch mehrfach zugestellt wird, sobald der Block aufgehoben wird.
- Es kann somit zum Verlust an Signalen kommen.

sigfire.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static const int NOF_SIGNALS = 1000;
static volatile sig_atomic_t received_signals = 0;
static volatile sig_atomic_t terminated = 0;

static void count_signals(int sig) {
   ++received_signals;
}

void termination_handler(int sig) {
   terminated = 1;
}
```

- Dieses Beispiel soll den potentiellen Verlust von Signalen demonstrieren, indem gezählt wird, wieviel von insgesamt 1000 verschickten Signalen ankommen.

Verlust von Signalen

sigfire.c

```
int main() {
   sighold(SIGUSR1); sighold(SIGTERM);
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      struct sigaction action = {
         .sa_handler = count_signals,
      };
      if (sigaction(SIGUSR1, &action, 0) != 0) {
         perror("sigaction"); exit(1);
      }
      action.sa_handler = termination_handler;
      if (sigaction(SIGTERM, &action, 0) != 0) {
         perror("sigaction"); exit(1);
      }
      sigrelse(SIGUSR1); sigrelse(SIGTERM);
      while (!terminated) pause();
      printf("[%d] received %d signals\n", (int) getpid(), received_signals);
      exit(0);
   }

   sigrelse(SIGUSR1); sigrelse(SIGTERM);
   for (int i = 0; i < NOF_SIGNALS; ++i) {
      kill(child, SIGUSR1);
   }
   printf("[%d] sent %d signals\n", (int) getpid(), NOF_SIGNALS);
   kill(child, SIGTERM); wait(0);
}
```

sigfire.c

```
sighold(SIGUSR1); sighold(SIGTERM);
/* ... */
sigrelse(SIGUSR1); sigrelse(SIGTERM);
```

- Mit der Funktion *sighold* kann ein Signal auch außerhalb eines Signalbehandlers explizit geblockt werden.
- Mit *sigrelse* kann dies wieder rückgängig gemacht werden.
- Auf diese Weise können kritische Bereiche geschützt werden.

- Mit Hilfe der Funktionen *wait*() oder *waitpid*() wird die Terminierung erzeugter Prozesse *synchron* abgewickelt.
- Gelegentlich ist es auch sinnvoll, sich die Terminierung über Signale *asynchron* mitteilen zu lassen. Dies geht mit dem Signal *SIGCHLD*, das an den Erzeuger versendet wird, sobald eine der von ihm erzeugten Prozesse terminiert.
- Per Voreinstellung wird dieses Signal ignoriert.

sigchld.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "processlist.h"

static processlist alive, dead;

void child_term_handler(int sig) {
   pid_t pid; int wstat;
   while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
      if (pl_move(&alive, &dead, pid)) {
         pl_modify(&dead, pid, wstat);
      }
   }
}
```

- In diesem Beispiel werden zahlreiche Prozesse erzeugt, deren Exit-Status zeitnah in einer Datenstruktur verwaltet wird.

sigchld.c

```
int main() {
   struct sigaction action = {
      .sa_handler = child_term_handler,
   };
   if (sigaction(SIGCHLD, &action, 0) != 0) {
      perror("sigaction"); exit(1);
   }
   pl_alloc(&alive, 4); pl_alloc(&dead, 4);
   sighold(SIGCHLD);
   for (int i = 0; i < 10; ++i) {
      fflush(0); pid_t child = fork();
      if (child < 0) {
         perror("fork"); exit(1);
      }
      if (child == 0) {
         srand(getpid()); sleep(rand() % 5); exit((char) rand());
      }
      pl_add(&alive, child, 0);
   }
   sigrelse(SIGCHLD);
   while (pl_length(&alive) > 0 || pl_length(&dead) > 0) {
      if (pl_length(&dead) == 0) pause();
      while (pl_length(&dead) > 0) {
         sighold(SIGCHLD);
         int wstat; pid_t pid = pl_pick(&dead, &wstat);
         sigrelse(SIGCHLD);
         printf("[%d] %d\n", (int) pid, WEXITSTATUS(wstat));
      }
   }
}
```

```
theon$ tinysh
% cat >OUT
Some input...
^C
theon$
```

- Die zuvor vorgestellte Shell *tinysh* kümmerte sich nicht um die Signalbehandlung.
- Entsprechend führt ein *SIGINT* auf dem kontrollierenden Terminal nicht nur zum Abbruch des aufgerufenen Kommandos, sondern auch unerfreulicherweise zum abrupten Ende von *tinysh*.

Wie muss also die Signalbehandlung einer Shell aussehen?

- Wenn ein Kommando *im Vordergrund* läuft, muss die Shell die Signale *SIGINT* und *SIGQUIT* ignorieren.
- Wenn ein Kommando **im Hintergrund** läuft, müssen für diesen Prozess *SIGINT* und *SIGQUIT* ignoriert werden.
- Wenn die Shell ein Kommando einliest, sollten *SIGINT* und *SIGQUIT* die Neu-Eingabe des Kommandos ermöglichen.
- Bezüglich *SIGHUP* muss nichts unternommen werden.

Signalbehandlung in einer Shell

tinysh2.c

```
static volatile sig_atomic_t interrupted = 0;

void interrupt_handler(int sig) {
   interrupted = 1;
}

int main() {
   struct sigaction action = {
      .sa_handler = interrupt_handler,
   };
   if (sigaction(SIGINT, &action, 0) != 0 ||
         sigaction(SIGQUIT, &action, 0) != 0) {
      perror("sigaction");
   }

   stralloc line = {0}; strlist tokens = {0}; command cmd = {0};
   while (getline(&line)) {
      stralloc_0(&line); /* required by tokenizer() */
      tokens.len = 0;
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      if (!scan_command(&tokens, &cmd)) continue;

      sighold(SIGINT); sighold(SIGQUIT);
      // ... fork & (exec | wait) ...
      sigrelse(SIGINT); sigrelse(SIGQUIT);
   }
}
```

tinysh2.c

```
sighold(SIGINT); sighold(SIGQUIT);
pid_t child = fork();
if (child == -1) {
   print_error("fork"); continue;
}
if (child == 0) {
   sigrelse(SIGINT); sigrelse(SIGQUIT);
   if (cmd.background) {
      sigignore(SIGINT); sigignore(SIGQUIT);
   }
   exec_command(&cmd);
   print_error(cmd.cmdname);
   exit(255);
}

if (cmd.background) {
   outbuf_printf(&out, "%d\n", (int) child);
} else {
   int wstat;
   pid_t pid = waitpid(child, &wstat, 0);
   if (pid < 0) {
      print_error("wait");
   } else if (!WIFEXITED(wstat) || WEXITSTATUS(wstat)) {
      print_child_status(pid, wstat);
   }
}
sigrelse(SIGINT); sigrelse(SIGQUIT);
```

tinysh2.c

```
bool getline(stralloc* line) {
   bool first = true;
   interrupted = 0;
   for(;;) {
      if (interrupted) {
         interrupted = 0;
         outbuf_printf(&out, "\n");
         first = true;
      }
      if (first) {
         status_report();
         outbuf_printf(&out, "tinysh2> ");
         first = false;
      }
      errno = 0;
      outbuf_flush(&out);
      if (inbuf_sareadline(&in, line)) return true;
      if (errno != EINTR) return false;
   }
}
```

tinysh2.c

```
void print_child_status(pid_t pid, int wstat) {
   outbuf_printf(&out, "[%d] ", (int) pid);
   if (WIFEXITED(wstat)) {
      outbuf_printf(&out, "exit %d", WEXITSTATUS(wstat));
   } else if (WIFSIGNALED(wstat)) {
      outbuf_printf(&out, "terminated with signal %d", WTERMSIG(wstat));
      if (WCOREDUMP(wstat)) outbuf_printf(&out, " (core dump)");
   } else if (WIFSTOPPED(wstat)) {
      outbuf_printf(&out, "stopped with signal %d", WSTOPSIG(wstat));
   } else if (WIFCONTINUED(wstat)) {
      outbuf_printf(&out, "continued");
   } else {
      outbuf_printf(&out, "???");
   }
   outbuf_printf(&out, "\n");
}

void status_report(void) {
   pid_t pid; int wstat;
   while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
      print_child_status(pid, wstat);
   }
}
```

tinysh2.c

```
pid_t pid; int wstat;
while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
   print_child_status(pid, wstat);
}
```

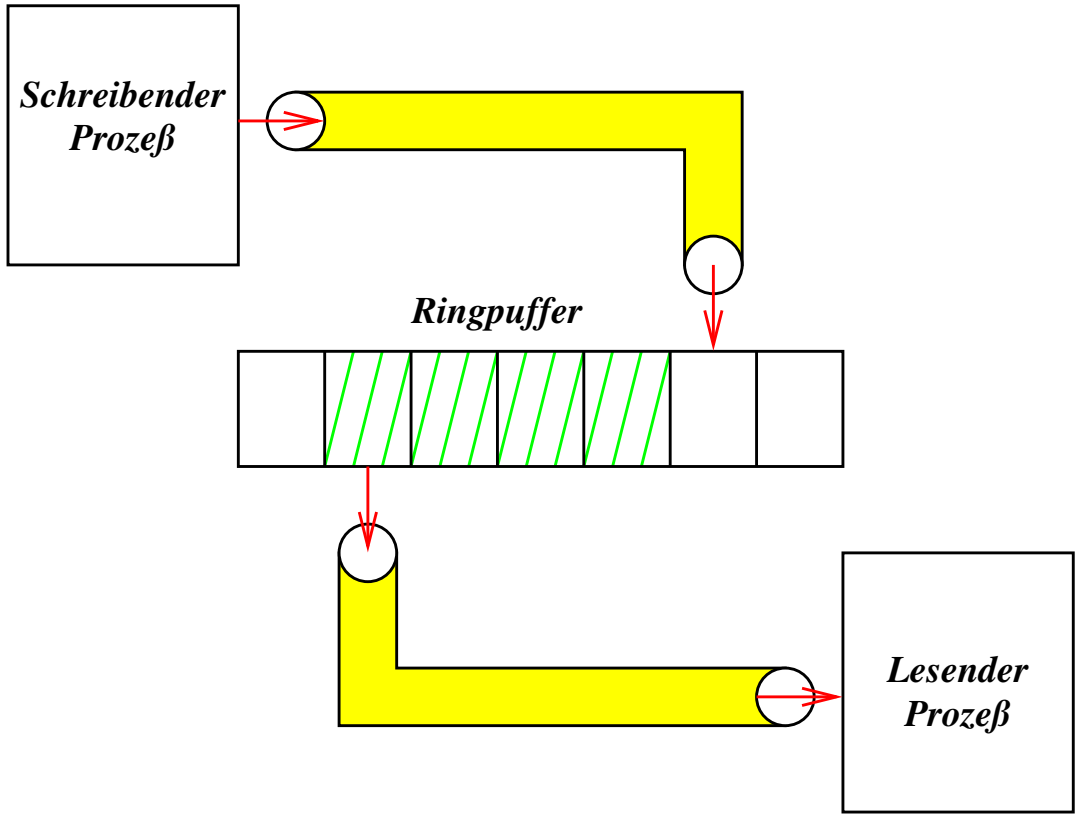
- Die Funktion *waitpid* wartet auf einen gegebenen Kindprozess.
- Wenn (*pid_t*)−1 angegeben wird, dann werden alle Kinder akzeptiert.
- Mit der Option *WNOHANG* blockiert *waitpid* nicht und liefert 0 zurück, falls momentan noch kein Exit-Code für einer der Kind-Prozesse zur Verfügung steht.

command.h

```
#ifndef COMMAND_H
#define COMMAND_H

#include <fcntl.h>
#include <stdbool.h>
#include <afblib/strlist.h>

typedef struct fd_assignment {
   char* path;
   int oflags;
   mode_t mode;
} fd_assignment;

typedef struct command {
   char* cmdname;
   strlist argv;
   int background;
   /* for file descriptors 0 and 1 */
   fd_assignment assignments[2];
} command;

/* convert list of tokens into a command record;
   return true in case of success */
bool scan_command(strlist* tokens, command* cmd);

/* open input and output files, if required, and
   exec to the given command */
void exec_command(command* cmd);

#endif
```

```
theon$ ypcat passwd | iconv -f latin1 | cut -d: -f5 |
> sed 's/ .*//' | sort | uniq -c | sort -rn | head
  54 Michael
  48 Daniel
  46 Alexander
  44 Tobias
  37 Florian
  35 Christian
  33 Matthias
  32 Johannes
  30 Markus
  30 Lukas
theon$
```

- Welches sind die 10 häufigsten Vornamen unserer Benutzer?
- Dank Pipelines und dem Unix-Werkzeugkasten lässt sich diese Frage schnell beantworten.
- Die Notation und die zugehörige Art der Interprozesskommunikation wurde von Douglas McIlroy, einem der Mitautoren der ersten Unix-Shell, in den 70er-Jahren entwickelt und hat sehr zur Popularität von Unix beigetragen.



- Pipelines sind unidirektionale Kommunikationskanäle. Die beiden Enden einer Pipeline werden über verschiedene Dateiverbindungen angesprochen.
- Sie werden innerhalb des Unix-Betriebssystems mit Hilfe eines festdimensionierten Ringpuffers implementiert.

- Typische Größen des Ringbuffers sind 64 Kilobyte (Linux, OS X) oder 20 Kilobyte (Solaris 10).
- Wenn der Puffer vollständig gefüllt ist, wird ein Prozess, der ihn weiter zu füllen versucht, blockiert, bis wieder genügend Platz zur Verfügung steht.
- Wenn der Puffer leer ist, wird ein lesender Prozess blockiert, bis der Puffer sich zumindest partiell füllt.
- Dies ist vergleichbar mit der Datenstruktur einer FIFO-Queue (*first in, first out*) mit explizit begrenzter Kapazität.
- Der POSIX-Standard unterstützt sowohl benannte Pipelines als auch solche, die mit Hilfe des Systemaufrufs *pipe*() erzeugt werden. Die benannten Pipelines sind aber kaum noch in Gebrauch, da die bidirektionalen UNIX-Domain-Sockets (mehr dazu später) normalerweise bevorzugt werden.

Konstruktion einer Pipeline

pipehello.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
enum {PIPE_READ = 0, PIPE_WRITE = 1};
int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      close(pipefds[PIPE_WRITE]);
      char buf[32];
      ssize_t nbytes;
      while ((nbytes = read(pipefds[PIPE_READ],
            buf, sizeof buf)) > 0) {
         if (write(1, buf, nbytes) < nbytes) exit(1);
      }
      exit(0);
   }
   close(pipefds[PIPE_READ]);
   const char message[] = "Hello!\n";
   write(pipefds[PIPE_WRITE], message, sizeof message - 1);
   close(pipefds[PIPE_WRITE]);
   wait(0);
}
```

pipehello.c

```
enum {PIPE_READ = 0, PIPE_WRITE = 1};
int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   /* ... */
}
```

- Mit dem Systemaufruf *pipe* wird eine Pipeline erzeugt. Zurückgegeben wird dabei ein Array mit zwei Dateiverbindungen, die auf das lesende (Index 0) und das schreibende (Index 1) Ende verweisen.
- Normalerweise wird eine Interprozesskommunikation auf Basis von *pipe* nur über *fork* aufgebaut, indem das entsprechende andere Ende der Pipeline an einen neu erzeugten Prozess vererbt wird.
- (Theoretisch ist es auch möglich, Dateideskriptoren (und damit auch eine Seite einer Pipeline) über UNIX-Domain-Sockets zu übermitteln.)

Schreibende Seite einer Pipeline

pipehello.c

```
pid_t child = fork();
if (child < 0) {
   perror("fork"); exit(1);
}
if (child == 0) {
   /* ... */
}
close(pipefds[PIPE_READ]);
const char message[] = "Hello!\n";
write(pipefds[PIPE_WRITE], message, sizeof message - 1);
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Der in eine Pipeline schreibende Prozess sollte das nicht genutzte Ende der Pipeline (hier das lesende) schließen. (Mehr dazu später.)
- Danach kann auf das schreibende Ende ganz normal mit *write* (oder auch darauf aufbauend der *stdio*) geschrieben werden.
- Sobald dies abgeschlossen ist, sollte das schreibende Ende geschlossen werden, damit ein Eingabe-Ende auf der anderen Seite der Pipeline erkannt werden kann.

Lesende Seite einer Pipeline

pipehello.c

```
if (child == 0) {
   close(pipefds[PIPE_WRITE]);
   char buf[32];
   ssize_t nbytes;
   while ((nbytes = read(pipefds[PIPE_READ],
         buf, sizeof buf)) > 0) {
      if (write(1, buf, nbytes) < nbytes) exit(1);
   }
   exit(0);
}
```

- Der von einer Pipeline lesende Prozess sollte das nicht genutzte Ende der Pipeline (hier das schreibende) schließen. (Mehr dazu später.)
- Danach kann auf das lesende Ende ganz normal mit *read* (oder auch darauf aufbauend der *stdio*) gelesen werden.
- Die Schleife kopiert einfach alle Eingaben aus der Pipeline zur Dateiverbindung 1 (Standard-Ausgabe).
- Sobald der Ringpuffer geleert ist und alle schreibenden Enden geschlossen sind, wird ein Eingabe-Ende erkannt.

- Nach *pipe* und *fork* haben zwei Prozesse jeweils beide Enden der Pipeline.
- Ein Eingabe-Ende auf der lesenden Seite wird genau dann (und nur dann!) erkannt, wenn **alle** schreibenden Enden geschlossen sind.
- Wenn also die lesende Seite es versäumt, die schreibende Seite zu schließen, wird sie kein Eingabe-Ende erkennen, wenn der andere Prozess seine schreibende Seite schließt.
- Stattdessen käme es zu einem endlosen Hänger.

- Genau dann (und nur dann!) wenn es kein Ende der Pipeline zum Lesen mehr gibt, führt das Schreiben auf das Ende zum Schreiben zur Zustellung des *SIGPIPE*-Signals bzw. dem Fehler *EPIPE*.
- Wenn die schreibende Seite es versäumt, ihr Ende zum Lesen zu schließen und der lesende Prozess aus irgendwelchen Gründen terminiert, ohne die Pipeline auslesen zu können, dann füllt sich zunächst der Ringpuffer und danach wird die schreibende Seite endlos blockiert.
- Entsprechend gäbe es wieder einen endlosen Hänger.
- Deswegen ist es von kritischer Bedeutung, dass die nicht benötigten Enden nach *fork* bei beiden Prozessen sofort geschlossen werden, um diese Probleme zu vermeiden.

Behandlung von *SIGPIPE*

sigpipe.c

```
int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      close(pipefds[PIPE_WRITE]);
      char buf[32];
      ssize_t nbytes = read(pipefds[PIPE_READ],
            buf, sizeof buf);
      if (nbytes > 0) {
         if (write(1, buf, nbytes) < nbytes) exit(1);
      }
      exit(0);
   }
   close(pipefds[PIPE_READ]);
   struct sigaction action = {0}; action.sa_handler = sigpipe_handler;
   if (sigaction(SIGPIPE, &action, 0) < 0) {
      perror("sigaction"); exit(1);
   }
   while (!sigpipe_received) {
      const char message[] = "Hello!\n";
      write(pipefds[PIPE_WRITE], message, sizeof message - 1);
   }
   close(pipefds[PIPE_WRITE]); wait(0);
}
```

sigpipe.c

```
volatile sig_atomic_t sigpipe_received = 0;

void sigpipe_handler(int sig) {
   sigpipe_received = 1;
}
```

- Der Signalbehandler für *SIGPIPE* setzt hier nur eine globale Variable, so dass entsprechend getestet werden kann.
- Alternativ könnte als Signalbehandler auch *SIG_IGN* eingetragen werden. Das würde keine Funktion benötigt werden und es müsste dann explizit jede *write*-Operation überprüft werden. Wenn niemand mehr das andere Ende lesen kann, würde *errno* auf *EPIPE* gesetzt werden.

sigpipe.c

```
if (child == 0) {
   close(pipefds[PIPE_WRITE]);
   char buf[32];
   ssize_t nbytes = read(pipefds[PIPE_READ],
         buf, sizeof buf);
   if (nbytes > 0) {
      if (write(1, buf, nbytes) < nbytes) exit(1);
   }
   exit(0);
}
```

- Anders als zuvor ruft der neu erzeugte Prozess *read* nur ein einziges Mal auf und endet dann.
- Sobald sich dieser Prozess mit *exit* verabschiedet, bleibt kein lesendes Ende der Pipeline mehr offen, so dass damit dann die schreibende Seite das Signal *SIGPIPE* erhält, sobald sie in die Pipeline weiterhin schreibt.

sigpipe.c

```
close(pipefds[PIPE_READ]);
struct sigaction action = {0};
action.sa_handler = sigpipe_handler;
if (sigaction(SIGPIPE, &action, 0) < 0) {
   perror("sigaction"); exit(1);
}
while (!sigpipe_received) {
   const char message[] = "Hello!\n";
   write(pipefds[PIPE_WRITE], message, sizeof message - 1);
}
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Beim übergeordneten Prozess wird zunächst der Signalbehandler für *SIGPIPE* eingesetzt.
- Danach wird solange in die Pipeline geschrieben, bis das Signal endlich eintrifft.

sigpipe2.c

```
close(pipefds[PIPE_READ]);
sigignore(SIGPIPE);
ssize_t nbytes;
do {
   const char message[] = "Hello!\n";
   nbytes = write(pipefds[PIPE_WRITE],
      message, sizeof message - 1);
} while (nbytes > 0);
if (errno != EPIPE) perror("write");
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Alternativ könnte *SIGPIPE* ignoriert werden.
- Dann ist die Überprüfung der *write*-Operationen zwingend notwendig.

- Pipelines werden sehr gerne eingesetzt, um die Ausgabe eines Kommandos auszulesen und/oder die zugehörige Eingabe zu generieren.
- POSIX bietet für diese Funktionalität auf Basis der *stdio* die Funktionen *popen*() und *pclose*() an.
- Da *popen* in jedem Falle das erste Argument mitsamt Sonderzeichen an die Shell weiterreicht, ist dies nicht ohne Sicherheitsrisiken, die sich bei dieser Schnittstelle leider nicht vermeiden lassen.
- Das Sicherheitsrisiko ist beispielsweise gegeben, wenn Teile des ersten Arguments durch Benutzereingaben beeinflussbar sind.
- Deswegen ist von dieser Schnittstelle abzuraten.
- Besser ist es, direkt mit *pipe*, *fork* und *execvp* zu arbeiten, so dass keine Gefahr besteht, dass Kommandozeilenargumente als Programmieranweisung in der Shell missverstanden werden.

pconnect.h

```
#include <stdbool.h>
#include <unistd.h>

enum {PIPE_READ = 0, PIPE_WRITE = 1};

typedef struct pipe_end {
   int fd;
   pid_t pid; /* pid of the forked-off child */
   int wstat; /* result of wait returned by phangup */
} pipe_end;

/* create a pipeline to the given command;
   mode should be either PIPE_READ or PIPE_WRITE;
   return a filled pipe_end structure and true on success
   and false in case of failures */
bool pconnect(const char* path, char* const* argv,
      int mode, pipe_end* pipe_con);

/* like pconnect() but connect fd to the standard input
   or output file descriptor that is not connected to the pipe */
bool pconnect2(const char* path, char* const* argv,
      int mode, int fd, pipe_end* pipe_con);

/* close pipeline and wait for the forked-off process to exit;
   the wait status is returned in pipe->wstat;
   true is returned if successful, false otherwise */
bool phangup(pipe_end* pipe_end);
```

pconnect.h

```
typedef struct pipe_end {
   int fd;
   pid_t pid; /* pid of the forked-off child */
   int wstat; /* result of wait returned by phangup */
} pipe_end;
```

- In der Verwaltungsstruktur wird von *pconnect* die Prozess-ID des neu erzeugten Prozesses und der Dateideskriptor zur Pipeline notiert.
- Wenn *phangup* aufgerufen wird, kann auf das Ende dieser Prozess-ID mit *waitpid* gewartet werden.
- Der zurückgelieferte Status wird dann in *wstat* abgelegt.

pconnect.c

```
/* like pconnect() but connect fd to the standard input
   or output file descriptor that is not connected to the pipe */
bool pconnect2(const char* path, char* const* argv,
      int mode, int fd, pipe_end* pipe_con) {
   int pipefds[2];
   if (pipe(pipefds) < 0) return false;
   int parent_side = mode; int child_side = 1 - mode;
   pid_t child = fork();
   if (child < 0) {
      close(pipefds[0]); close(pipefds[1]); return false;
   }
   if (child == 0) {
      close(pipefds[parent_side]);
      dup2(pipefds[child_side], child_side); close(pipefds[child_side]);
      if (fd != parent_side) {
         dup2(fd, parent_side); close(fd);
      }
      execvp(path, argv); exit(255);
   }
   close(pipefds[child_side]);
   /* make sure that our side is closed for forked-off childs */
   if (!add_fd(pipefds[parent_side])) return false;
   /* make sure that our side is closed when we exec */
   int flags = fcntl(pipefds[parent_side], F_GETFD);
   flags |= FD_CLOEXEC;
   fcntl(pipefds[parent_side], F_SETFD, flags);
   pipe_con->pid = child;
   pipe_con->fd = pipefds[parent_side];
   pipe_con->wstat = 0;
   return true;
}
```

pconnect.c

```
bool pconnect2(const char* path, char* const* argv,
      int mode, int fd, pipe_end* pipe_con) {
   int pipefds[2];
   if (pipe(pipefds) < 0) return false;
   int parent_side = mode; int child_side = 1 - mode;
   pid_t child = fork();
   if (child < 0) {
      close(pipefds[0]); close(pipefds[1]);
      return false;
   }
   /* ... */
}
```

- Der Index *myside* wird auf zu benutzende Ende des übergeordneten Prozesses gesetzt, *otherside* auf das Ende des neu erzeugten Prozesses.
- *parent_side* und *child_side* dienen als Index für *pipefds*.
- *pipefds*[0] ist die lesende Seite, *pipefds*[1] die schreibende.
- Passend dazu ist *PIPE_READ* 0 und *PIPE_WRITE* 1.

pconnect.c

```
if (child == 0) {
   close(pipefds[parent_side]);
   dup2(pipefds[child_side], child_side);
   close(pipefds[child_side]);
   if (fd != parent_side) {
      dup2(fd, parent_side); close(fd);
   }
   execvp(path, argv); exit(255);
}
```

- Beim Kindprozess wird zunächst das nicht benötigte Ende der Pipeline geschlossen. Dann wird mit *dup2* das verbliebene Ende als Standardeingabe bzw. -ausgabe zur Verfügung gestellt. Nach dem *dup2*-Aufruf kann die dann überflüssig gewordene Dateiverbindung geschlossen werden.
- Die Standard-Aus- und Eingabe sind beide zu setzen. Der Dateideskriptor *fd* ist deswegen bei Bedarf mit dem jeweils noch nicht festgelegten Standard-Verbindung zu verknüpfen.

pconnect.c

```
close(pipefds[child_side]);
/* make sure that our side is closed for forked-off childs */
if (!add_fd(pipefds[parent_side])) return false;
/* make sure that our side is closed when we exec */
int flags = fcntl(pipefds[parent_side], F_GETFD);
flags |= FD_CLOEXEC;
fcntl(pipefds[parent_side], F_SETFD, flags);
pipe_con->pid = child;
pipe_con->fd = pipefds[parent_side];
pipe_con->wstat = 0;
return true;
```

- Die Option *FD_CLOEXEC* sorgt dafür, dass diese Dateiverbindung automatisch beim Aufruf einer der *exec*-Varianten geschlossen wird. Dies ist wichtig, falls mehrere Pipelines parallel genutzt werden.
- Mit *add_fd* werden die bei einem späteren *fork* zu schließenden Dateideskriptoren gesammelt.

pconnect.c

```
static bool initialized = false;
static fd_set pipes;

static void child_after_fork_handler() {
   /* close all pipes that were opened by pconnect/pconnect2 */
   for (int fd = 0; fd < FD_SETSIZE; ++fd) {
      if (FD_ISSET(fd, &pipes)) {
         close(fd);
      }
   }
   FD_ZERO(&pipes);
}

static bool add_fd(int fd) {
   if (!initialized) {
      FD_ZERO(&pipes);
      if (pthread_atfork(0, 0, child_after_fork_handler) < 0) {
         return false;
      }
      initialized = true;
   }
   FD_SET(fd, &pipes);
   return true;
}

static void remove_fd(int fd) {
   FD_CLR(fd, &pipes);
}
```

pconnect.c

```
static fd_set pipes;
```

- Der Datentyp *fd_set* repräsentiert eine Menge von Dateideskriptoren (als festdimensionierter Bitset).
- Die Datenstruktur und die zugehörigen Makros steht über **#include** <sys/select.h> zur Verfügung:

void *FD_CLR*(**int** *fd*, *fd_set** *fdsetp*);	$fdset \leftarrow fdset \setminus \{fd\}$
int *FD_ISSET*(**int** *fd*, *fd_set** *fdsetp*);	$fd \in fdset$
void *FD_SET*(**int** *fd*, *fd_set** *fdsetp*);	$fdset \leftarrow fdset \cup \{fd\}$
void *FD_ZERO*(*fd_set** *fdsetp*);	$fdset \leftarrow \{\}$

pconnect.c

```
static bool add_fd(int fd) {
   if (!initialized) {
      FD_ZERO(&pipes);
      if (pthread_atfork(0, 0, child_after_fork_handler) < 0) {
         return false;
      }
      initialized = true;
   }
   FD_SET(fd, &pipes);
   return true;
}
```

- Es ist wichtig, dass die Pipe-Enden unseres Prozesses nicht an mit *fork* erzeugte neue Prozesse weitervererbt werden.
- Mit *pthread_atfork* wird hier *child_after_fork_handler* konfiguriert als Handler, der unmittelbar nach *fork* auf der Seite des Kindprozesses aufzurufen ist.

pconnect.c

```
static void child_after_fork_handler() {
   /* close all pipes that were opened by pconnect/pconnect2 */
   for (int fd = 0; fd < FD_SETSIZE; ++fd) {
      if (FD_ISSET(fd, &pipes)) {
         close(fd);
      }
   }
   FD_ZERO(&pipes);
}
```

- Wann immer *fork* aufgerufen wird, sind beim Kindprozess alle Pipe-Enden zu schließen.

pconnect.c

```
bool phangup(pipe_end* pipe) {
   remove_fd(pipe->fd);
   if (close(pipe->fd) < 0) return false;
   if (waitpid(pipe->pid, &pipe->wstat, 0) < 0) return false;
   return true;
}
```

- *phangup* schließt die Verbindung zur Pipeline und wartet darauf, dass der entsprechende Kindprozess terminiert.

- Die messbare Größe des Pipe-Buffers lässt sich definieren als die maximale Zahl an Bytes, die blockierungsfrei mit *write* in eine Pipe geschrieben werden kann, ohne dass die Gegenseite liest.
- Insbesondere unter Solaris ist die Größe nicht einfach zu ermitteln. Wenn hier *O_NONBLOCK* gesetzt wird und *write* bei einer Zahl von Bytes, die über dem Limit liegt, einen kleineren Wert tatsächlich geschriebener Bytes zurückgibt, dann liegt dieser Wert unter dem theoretischen Maximum.
- Konkret unter Solaris 11:
  - *write*(*fd*, *buf*, 25600) liefert 20480.
  - *write*(*fd*, *buf*, 25599) liefert jedoch 25599.

Wie groß ist der Pipe-Buffer?

measure-pipe.c

```
static int pipe_and_fork(int i, size_t nbytes) {
   int fds[2];
   if (pipe(fds) < 0) die("pipe");
   pid_t pid = fork(); if (pid < 0) die("fork");
   if (pid == 0) {
      close(fds[0]);
      char* buf = malloc(nbytes);
      int fd = fds[1];
      int flags = fcntl(fd, F_GETFL) | O_NONBLOCK;
      fcntl(fd, F_SETFL, flags);
      ssize_t written = write(fd, buf, nbytes);
      if (written < nbytes) exit(255);
      exit(i);
   }
   close(fds[1]);
   return fds[0];
}
```

- Für jeden einzelnen Test wird ein Prozess und eine Pipeline erzeugt. Mit *O_NONBLOCK* wird sichergestellt, dass *write* nicht blockiert.

measure-pipe.c

```
static size_t suck_pipe(int fd, size_t expected) {
   char* buf = malloc(expected); if (!buf) die("malloc");
   size_t bytes_read = 0;
   while (bytes_read < expected) {
      ssize_t nbytes = read(fd, buf, expected - bytes_read);
      if (nbytes < 0) die("read from pipe");
      if (nbytes == 0) break;
      bytes_read += nbytes;
   }
   close(fd);
   free(buf);
   return bytes_read;
}
```

- Mit *suck_pipe* wird überprüft, wieviel Bytes sich aus der Pipeline auslesen lassen, nachdem der Unterprozess terminiert ist und alle schreibenden Enden geschlossen sind.

Wie groß ist der Pipe-Buffer?

measure-pipe.c

```
static size_t run_tests(size_t nbytes[], size_t tests) {
   int pipes[tests];
   for (int i = 0; i < tests; ++i) {
      pipes[i] = pipe_and_fork(i, nbytes[i]);
   }
   // wait for all the forked processes
   int success[tests];
   for (int i = 0; i < tests; ++i) {
      success[i] = 0;
   }
   int wstat;
   pid_t pid;
   while ((pid = wait(&wstat)) > 0) {
      if (WIFEXITED(wstat)) {
         int status = WEXITSTATUS(wstat);
         if (status != 255 && status < MAXPROCESSES) {
            success[status] = 1;
         }
      }
   }
   // evaluate and confirm results
   size_t confirmed_len = 0;
   for (int i = 0; i < tests; ++i) {
      if (success[i]) {
         confirmed_len = suck_pipe(pipes[i], nbytes[i]);
         continue;
      }
      break;
   }
   return confirmed_len;
}
```

Wie groß ist der Pipe-Buffer?

measure-pipe.c

```
// check for buf sizes that are powers of two
size_t test1() {
   size_t nbytes[MAXSIZE];
   for (int i = 0; i < MAXSIZE; ++i) {
      nbytes[i] = (1 << i);
   }
   return run_tests(nbytes, MAXSIZE);
}

// check for arbitrary buf sizes
size_t test2(size_t buflen,
      size_t increment, size_t tests) {
   size_t nbytes[tests];
   for (size_t i = 0; i < tests; ++i) {
      nbytes[i] = buflen + increment * i;
   }
   return run_tests(nbytes, tests);
}
```

- Zuerst wird die größte Zweierpotenz ermittelt, die blockierungsfrei geschrieben werden kann. Später wird das sukzessive verfeinert.

Wie groß ist der Pipe-Buffer?

measure-pipe.c

```
int main() {
   size_t buflen = test1();
   if (!buflen) {
      printf("pipe buffer size is beyond %zu\n",
         (size_t)1 << (MAXSIZE-1)); exit(1);
   }
   size_t increment = buflen / MAXPROCESSES;
   size_t lastlen = 0; size_t tests = MAXPROCESSES;
   while (increment > 0) {
      size_t len = test2(buflen, increment, tests);
      if (!len) {
         printf("pipe buffer len is possibly %zu "
            "but that did not get confirmed\n", buflen); exit(1);
      }
      lastlen = len; buflen = len;
      if (increment > MAXPROCESSES) {
         tests = MAXPROCESSES;
         increment /= MAXPROCESSES;
      } else if (increment > 1) {
         tests = increment; increment = 1;
      } else {
         increment = 0;
      }
   }
   if (lastlen) {
      printf("pipe buffer size = %zu\n", lastlen);
   } else {
      printf("pipe buffer size = %zu\n", buflen);
   }
}
```

Ergebnisse:

- Solaris 8: 10240
- Solaris 9 und 10: 20480
- Solaris 11: 25599
- Linux: 65536
- OS X: 65536

Quellen: `https://github.com/afborchert/pipebuf`

- Eine unidirektionale Kommunikation is ausreichend, da alle Eingabedaten über *fork()* vererbt werden können.
- Spannend ist die Frage, wieviele Kommunikationskanäle benötigt werden: Ist für jeden Unterprozess eine Pipeline zu erzeugen oder kann eine Pipeline für alle Unterprozesse gemeinsam verwendet werden?
- Bei letzterem stellt sich die Frage, ob sich die Ausgaben verschiedener Unterprozesse in die gleiche Pipeline vermischen können. Hier stellt der POSIX-Standard sicher, dass dies nicht geschieht, sofern die bei *write* angegebene Quantität nicht mehr als *PIPE_BUF* beträgt.
- Konkrete Werte:
  - Solaris 8, 9, 10, 11: 5120
  - Linux: 4096
  - OS X: 512

Wie werden Pipelines von der Shell aufgesetzt?

Hier gibt es eine von der alten Bourne-Shell geprägte traditionelle Vorgehensweise und eine modernere Variante, die insbesondere von der Korn-Shell und der bash praktiziert werden.

Um den Unterschied zu sehen, hilft ein kleines Testprogramm.

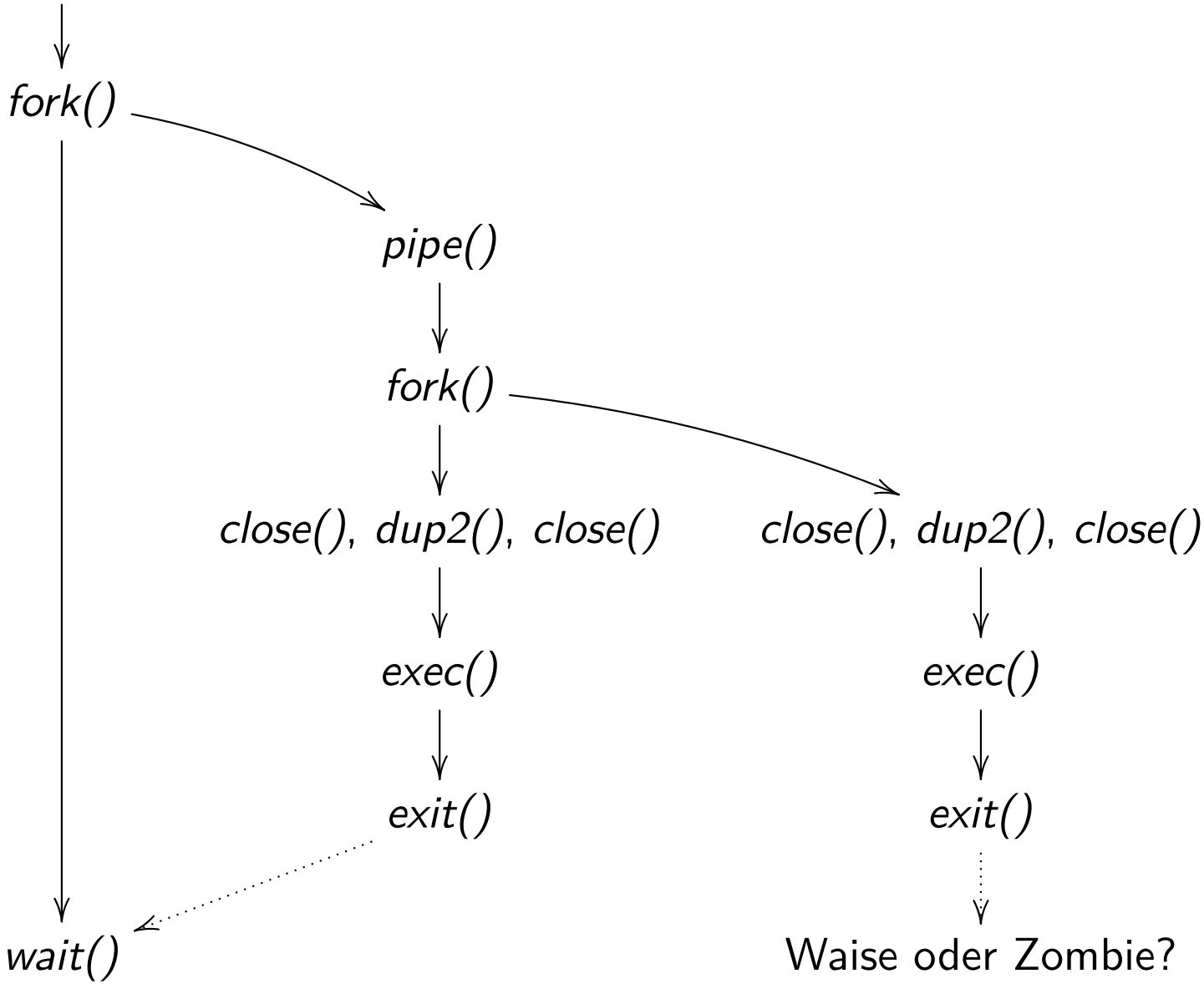
Testprogramm für die Shell

pinfo.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv) {
   /* who are we? */
   pid_t pid = getpid();
   pid_t parent = getppid();
   /* copy standard input to standard output if it is not a terminal */
   if (!isatty(0)) {
      char buf[BUFSIZ];
      ssize_t nbytes;
      while ((nbytes = read(0, buf, sizeof buf)) > 0) {
         size_t written = 0;
         while (written < nbytes) {
            ssize_t count = write(1, buf + written, nbytes - written);
            if (count <= 0) break;
            written += count;
         }
      }
   }
   /* and add info about own process to it */
   printf("pid = %d, parent = %d\n", (int) pid, (int) parent);
}
```

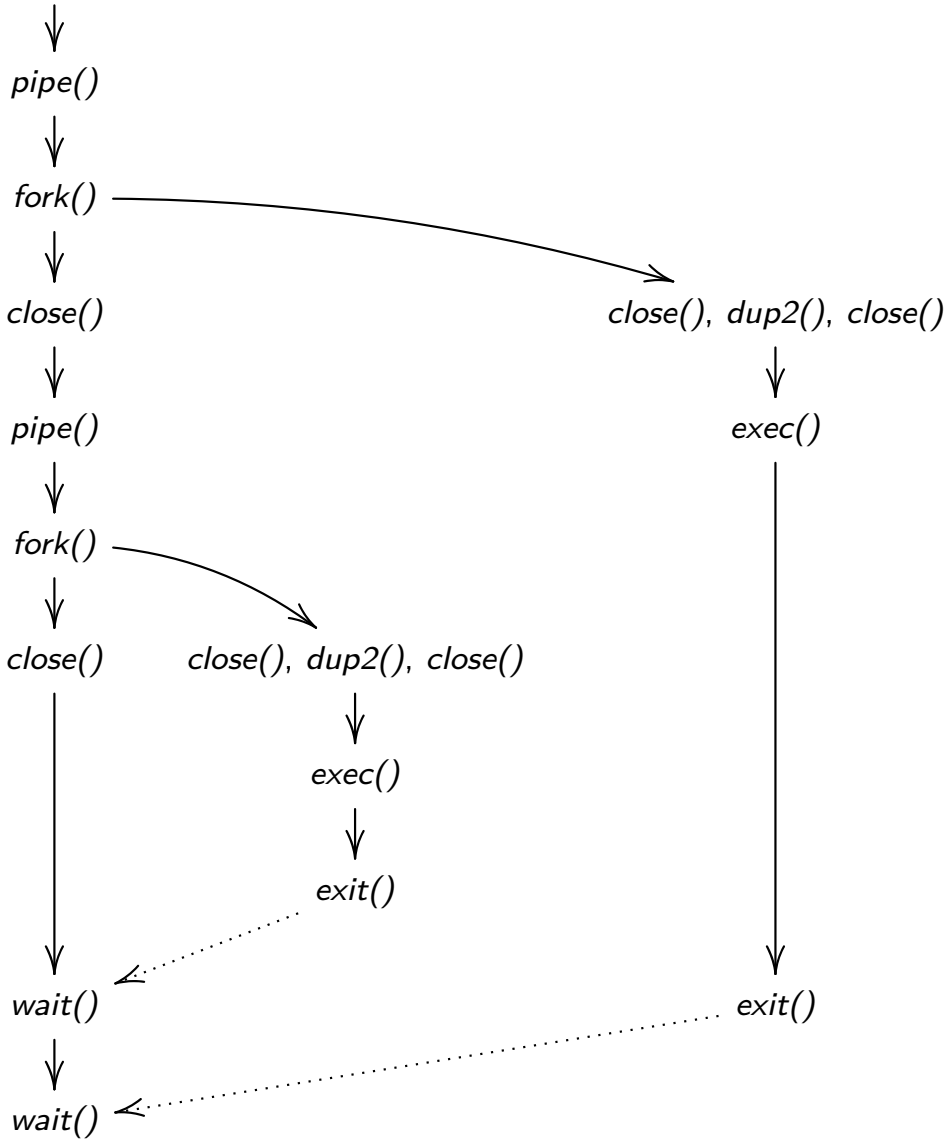
```
theseus$ /bin/sh -c 'echo $$; pinfo | pinfo | pinfo | pinfo'
25877
pid = 25879, parent = 25878
pid = 25880, parent = 25878
pid = 25881, parent = 25878
pid = 25878, parent = 25877
theseus$
```

- Der Shell-Prozess (25877) hat nur einen Kindprozess, der die gesamte Kommandozeile umsetzt (25878).
- Der Kindprozess legt die drei Pipelines an, ruft *fork* für alle Kommandos in der Pipeline mit Ausnahme des letzten, um dann schließlich selbst mit *exec* die Ausführung des letzten Kommandos in der Pipeline zu übernehmen.
- Der Shell-Prozess wartet nur auf den letzten Prozess in der Pipeline und kann nur dessen Exit-Code in Betracht ziehen.


fork()
pipe()
fork()
close(), dup2(), close()
close(), dup2(), close()
exec()
exec()
exit()
exit()
wait()
Waise oder Zombie?

```
theon$ /bin/ksh -c 'echo $$; pinfo | pinfo | pinfo | pinfo'
29704
pid = 29705, parent = 29704
pid = 29706, parent = 29704
pid = 29707, parent = 29704
pid = 29708, parent = 29704
theon$
```

- Der Shell-Prozess (29704) kontrolliert alle Kindprozesse.
- Vorteil: Es bleiben keine Waisen oder Zombies zurück, die Shell kann sich um alle von ihr erzeugten Kommandos kümmern.


pipe()
fork()
close()
pipe()
fork()
close()
close(), dup2(), close()
exec()
exit()
wait()
wait()
close(), dup2(), close()
exec()

Auf wen wird unmittelbar gewartet? 177

```
theon$ date; /bin/sh -c 'sleep 3 | true'; date
Mon May 27 13:24:29 CEST 2019
Mon May 27 13:24:29 CEST 2019
theon$ date; /bin/ksh -c 'sleep 3 | true'; date
Mon May 27 13:24:35 CEST 2019
Mon May 27 13:24:35 CEST 2019
theon$ date; /bin/bash -c 'sleep 3 | true'; date
Mon May 27 13:24:38 CEST 2019
Mon May 27 13:24:41 CEST 2019
theon$
```

- Die Bourne-Shell wartet nur auf das letzte Glied der Pipeline (auf die anderen Prozesse kann ohnehin nicht mehr gewartet werden).
- Die Korn-Shell wartet ebenfalls nur auf das letzte Glied der Pipeline – sie verhindert nur, dass die anderen Glieder der Pipeline verwaisen oder zu Zombies werden.
- Die *bash* wartet, bis alle Glieder der Pipeline fertig sind.

```
theon$ exit 1 | exit 2 | exit 3
theon$ echo ${PIPESTATUS[0]} ${PIPESTATUS[1]} ${PIPESTATUS[2]} $?
1 2 3 3
theon$
```

- In der Shell steht „$?“ für den Exit-Code des letzten Kommandos.
- Bei einer Pipeline bezieht sich das auf das letzte Glied der Kette.
- Die *bash* unterstützt über das *PIPESTATUS*-Array auch den Abruf der anderen Exit-Codes.

- Ein Netzwerkdienst ist ein Prozess, der unter einer Netzwerkadresse einen Dienst anbietet.
- Ein Klient, der die Netzwerkadresse kennt, kann einen bidirektionalen Kommunikationskanal zu dem Netzwerkdienst eröffnen und über diesen mit dem Dienst kommunizieren.
- Die Kommunikation wird durch ein Protokoll strukturiert, bei dem typischerweise Anfragen oder Kommandos auf dem Hinweg übermittelt werden und auf dem Rückweg des Kommunikationskanals die zugehörigen Antworten kommen.
- Wenn erst die Antwort gelesen werden muss, bevor die nächste Anfrage gestellt werden darf, wird von einem *synchronen* Protokoll gesprochen.
- Wenn mehrere Anfragen unmittelbar hintereinander gestellt werden dürfen, ohne dass erst die Antworten abgewartet werden, wird von *Pipelining* gesprochen. (Das hat nichts mit den Pipes aus dem vorherigen Kapitel zu tun.)

- Die beiden Kommunikationspartner müssen nicht miteinander verwandt sein.
- Sie müssen nicht einmal auf dem gleichen Rechner laufen.
- Da der Kommunikationskanal bidirektional ist, wird ein echter Dialog zwischen den beiden Prozessen möglich.
- Der Aufbau einer Verbindung ist jedoch schwieriger, da zunächst die Netzwerkadresse des gewünschten Partners ermittelt werden muss.

Wenn Dienste über das Netzwerk angeboten und in Anspruch genommen werden, ergeben sich viele Vorteile:

- Der Dienst kann allen offenstehen, und ein direkter Zugang zu dem Rechner, auf dem der Dienst angeboten wird, ist nicht notwendig.
- Viele Parteien können in kooperativer Weise einen Dienst gleichzeitig nutzen.
- Der Dienste-Anbieter hat weniger Last, da die Benutzerschnittstelle auf anderen Rechnern laufen kann.

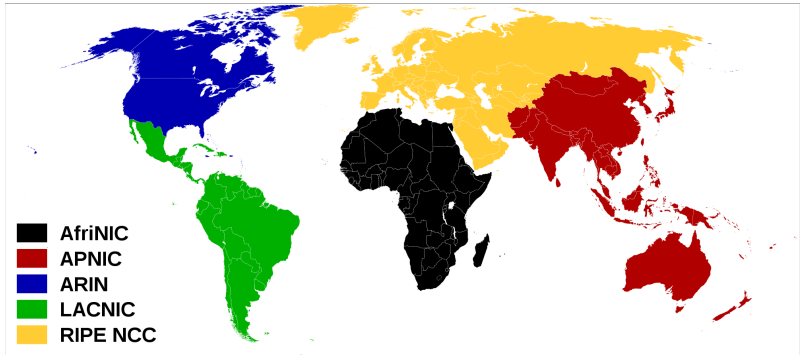
- Der Kreis derjenigen, die auf einen Netzwerkdienst zugreifen können, ist möglicherweise ziemlich umfangreich (normalerweise das gesamte Internet).
- Somit muss jeder Netzwerkdienst Zugriffsberechtigungen einführen und überprüfen und kann sich dabei nicht wie traditionelle Applikationen auf die des Betriebssystems verlassen.
- Dienste, die gleichzeitig von vielen genutzt werden können, haben vielerlei zusätzliche Konsistenz- und Synchronisierungsprobleme, für die nicht jede Art von Datenhaltung geeignet ist.
- Netzwerke bringen neue Arten von Ausfällen mit sich, wenn eine Netzwerkverbindung zusammenbricht oder es zu längeren „Hängern“ kommt.

- Im Rahmen dieser Vorlesung beschäftigen wir uns vorwiegend mit TCP/IP, also dem verbindungsorientierten Protokoll des Internets. (Mehr zur Semantik später.)
- Im Internet gibt es zwei etablierte Räume für Netzwerkadressen: IPv4 und IPv6.
- IPv4 arbeitet mit 32-Bit-Adressen und ist seit dem 1. Januar 1983 in Benutzung.
- Da der Adressraum bei IPv4 seit 2011 weitgehend ausgeschöpft ist, gibt es als Alternative IPv6, das mit 128-Bit-Adressen arbeitet.
- Im Rahmen dieser Vorlesung beschäftigen wir uns nur mit IPv4.
- Eine IPv4-Adresse (das gilt auch für IPv6) adressiert nur den Rechner, auf dem der Dienst läuft. Der Dienst selbst wird über eine Portnummer (16 Bit) ausgewählt.
- Ein Netzwerkdienst wird also z.B. über eine IPv4-Adresse und eine Port-Nummer adressiert.

```
clonard$ telnet 134.60.54.12 13
Trying 134.60.54.12...
Connected to 134.60.54.12.
Escape character is '^]'.
Mon Jun 14 11:03:16 2010
Connection to 134.60.54.12 closed by foreign host.
clonard$
```

- 134.60.54.12 ist eine IPv4-Adresse in der sogenannten *dotted-decimal*-Notation, bei der durch Punkte getrennt jedes der vier Bytes der Adresse einzeln dezimal spezifiziert wird.
- 134.60.54.12 ist also eine lesbarere Form für 2252092940.
- 13 ist die Port-Nummer des *daytime*-Dienstes.
- Die Port-Nummer ist nicht zufällig. Die 13 ist explizit von der IANA (*Internet Assigned Numbers Authority*) dem *daytime*-Dienst zugewiesen worden.

Aufteilung des Netzwerkraums



Karte von Dork und Sémhur auf Wikimedia Commons, CC-BY-SA 3.0

- Die IANA teilt den globalen IPv4-Adressraum auf einzelne lokale Institutionen, den sogenannten *Regional Internet Registries*.
- ARIN ist zuständig für Amerika, RIPE für Europa, den Mittleren Osten und Zentralasien, APNIC für Asien, Australien und Ozeanien, AfriNIC für Afrika und LACNIC für Lateinamerika einschließlich Teile der Karibik.
- Die Universität Ulm hat seit 1989 den Adressbereich *134.60.0.0/16*.

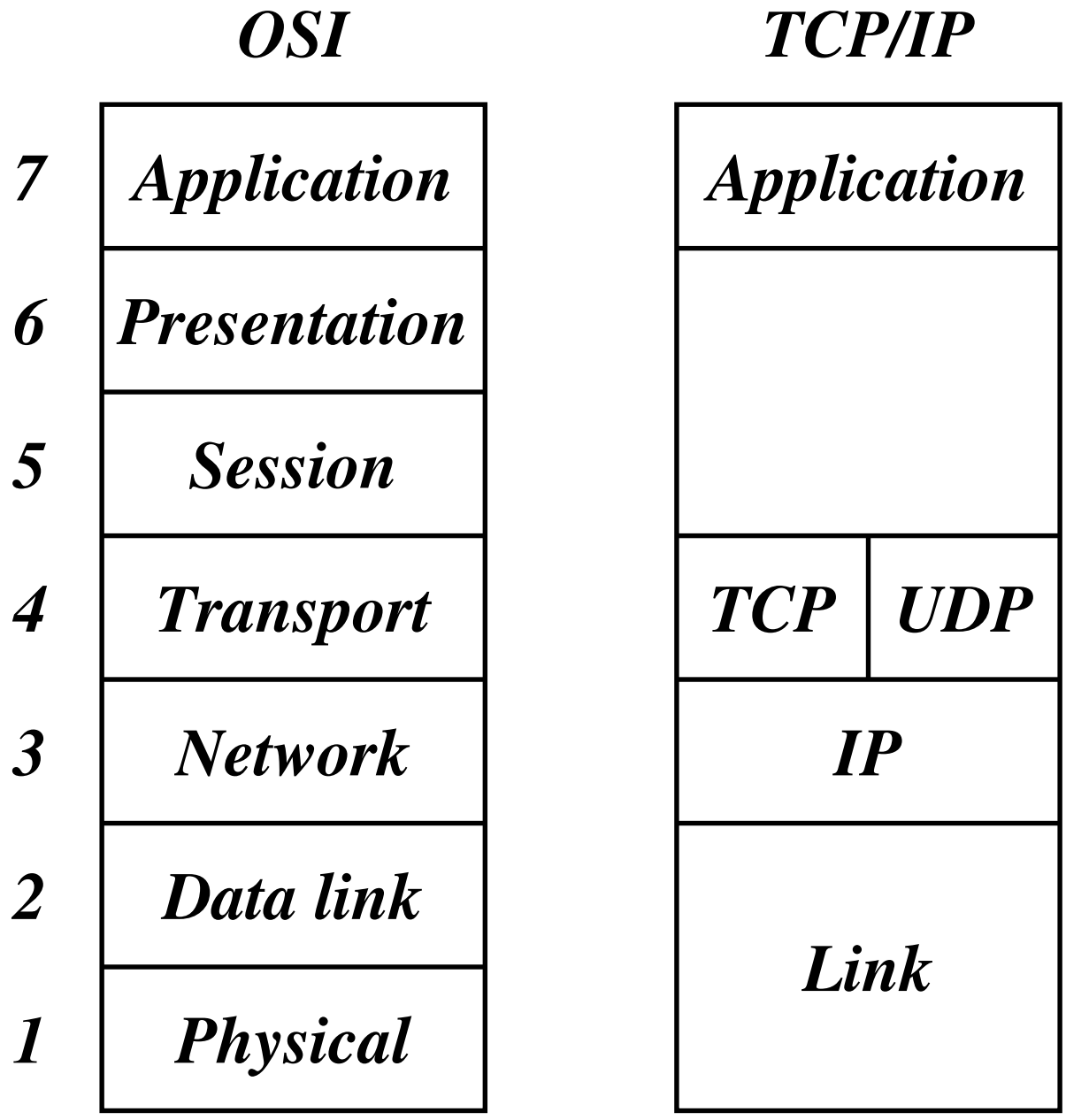
Wer ist zuständig für die 134.60.54.12? 186

```
theon$ curl -s http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.txt |
> sed 's/  */ /g' | grep '^ 134'
 134/8 Administered by ARIN 1993-05 whois.arin.net https://rdap.arin.net/registry LEGACY
theon$ whois -h whois.arin.net 134.60.66.7
[...]
NetRange:       134.58.0.0 - 134.61.255.255
CIDR:           134.60.0.0/15, 134.58.0.0/15
NetName:        RIPE-ERX-134-58-0-0
NetHandle:      NET-134-58-0-0-1
Parent:         NET134 (NET-134-0-0-0-0)
NetType:        Early Registrations, Transferred to RIPE NCC
OriginAS:
Organization:   RIPE Network Coordination Centre (RIPE)
RegDate:        2003-11-26
Updated:        2003-11-26
Comment:        These addresses have been further assigned to users in
Comment:        the RIPE NCC region.  Contact information can be found in
Comment:        the RIPE database at http://www.ripe.net/whois
[...]
theon$ whois -h whois.ripe.net 134.60.66.7
[...]
inetnum:        134.60.0.0 - 134.60.255.255
netname:        UNI-ULM
descr:          Ulm, Germany
country:        DE
[...]
route:          134.60.0.0/16
descr:          UNI-ULM
origin:         AS553
[...]
```

- Für Rechnernamen wie *theon.mathematik.uni-ulm.de* können über hierarchisierte Domain-Server die zugehörigen IP-Adressen abgefragt werden.
- Die Abfrage beginnt zuerst bei einem der 13 sogenannten Root-Server, die weltweit verteilt sind und deren IP-Adressen jedem Domain-Server bekannt sind.
- Einer davon ist *198.41.0.4*. Dieser verrät, welche Nameserver für die Top-Level-Domain *de* zuständig ist.
- Einer davon ist *194.0.0.53*. Dieser verrät, welche Nameserver für *uni-ulm.de* zuständig sind.
- Einer davon ist *134.60.1.111*, der sogleich in der Lage ist, diesen Namen vollständig aufzulösen und die *134.60.66.7* zurückzuliefern.

- IP-Adressen wie *134.60.66.7* werden nur auf einer abstrakten Ebene zur Verfügung gestellt.
- IP-Adressen werden auf der darunterliegenen physischen Ebene und denen damit verbundenen Protokollen nicht verstanden.
- So wird beispielsweise beim Ethernet, das bei uns weitgehend an der Universität zum Einsatz kommt, mit 6-Byte-Adressen gearbeitet.
- Die Theon hat beispielsweise die Ethernet-Adresse *90:1b:0e:ae:12:84* (Bytes werden hier in Form von Hexzahlen angegeben). Diese Adressen sind jedoch nur lokal auf einem Ethernet-Segment von Bedeutung.

- Aufbauend auf der Schicht mit IP-Adressen (IP-Protokoll) gibt es alternative Transport-Schichten, über die Pakete versendet werden können.
- Mittels UDP (*User Datagram Protocol*) können einzelne Pakete sehr effizient, aber unzuverlässig versendet werden.
- Im Gegensatz dazu gewährleistet TCP (*Transmission Control Protocol*) eine sichere Verbindung, die jedoch mehr Verwaltungsaufwand mit sich bringt.
- Parallel zu TCP/IP entstand 1983 das OSI-Referenz-Modell (*Open Systems Interconnection*), das eine feinere Schichtung vorsieht. Die Präsentations- oder Sitzungsebene fand jedoch nie ihren Weg in die Protokollhierarchie von TCP/IP.


OSI
TCP/IP
7
Application
6
Presentation
5
Session
4
Transport
3
Network
2
Data link
1
Physical
Application
TCP
UDP
IP
Link

- Für TCP/IP gibt es zwei Schnittstellen, die beide zum POSIX-Standard gehören:
- Die Berkeley Sockets wurden 1983 im Rahmen von BSD 4.2 eingeführt. Dies war die erste TCP/IP-Implementierung.
- Im Jahr 1987 kam durch UNIX System V Release 3.0 noch TLI (*Transport Layer Interface*) hinzu, die auf Streams basiert (einer anderen System-V-spezifischen Abstraktion).
- Die Berkeley-Socket-Schnittstelle hat sich weitgehend durchgesetzt. Wir werden uns daher nur mit dieser beschäftigen.

Die Entwickler der Berkeley-Sockets setzten sich folgende Ziele:

- **Transparenz**: Die Kommunikation zwischen zwei Prozessen soll nicht davon abhängen, ob sie auf dem gleichen Rechner laufen oder nicht.
- **Effizienz**: Zu Zeiten von BSD 4.2 (also 1983) war dies ein außerordentlich wichtiges Kriterium wegen der damals noch sehr geringen Rechenleistung. Aus diesem Grund werden insbesondere keine weiteren System-Prozesse zur Kommunikation eingesetzt, obwohl dies zu mehr Flexibilität und Modularität hätte führen können.
- **Kompatibilität**: Viele bestehende Applikationen und Bibliotheken wissen nichts von Netzwerken und sollen dennoch in einem verteilten Umfeld eingesetzt werden können. Dies wurde dadurch erreicht, dass nach einem erfolgten Verbindungsaufbau (der z.B. von einem anderen Prozess durchgeführt werden kann) Ein- und Ausgabe in gewohnter Weise (wie bei Dateien, Pipelines oder Terminal-Verbindungen) erfolgen können.

Die Semantik einer Kommunikation umschließt bei jeder Verbindung eine Teilmenge der folgenden Punkte:

1. Daten werden in der Reihenfolge empfangen, in der sie abgeschickt worden sind.
2. Daten kommen nicht doppelt an.
3. Daten werden zuverlässig übermittelt.
4. Einzelne Pakete kommen in der originalen Form an (d.h. sie werden weder zerstückelt noch mit anderen Paketen kombiniert).
5. Nachrichten außerhalb des normalen Kommunikationsstromes (*out-of-band messages*) werden unterstützt.
6. Die Kommunikation erfolgt verbindungs-orientiert, womit die Notwendigkeit entfällt, sich bei jedem Paket identifizieren zu müssen.

Die folgende Tabelle zeigt die Varianten, die von der Berkeley-Socket-Schnittstelle unterstützt werden:

Name	1	2	3	4	5	6
SOCK_STREAM	⋆	⋆	⋆		⋆	⋆
SOCK_DGRAM				⋆		
SOCK_SEQPACKET	⋆	⋆	⋆	⋆	⋆	⋆
SOCK_RDM	⋆	⋆	⋆	⋆		

(1: Reihenfolge korrekt; 2: nicht doppelt; 3: zuverlässige Übermittlung; 4: keine Stückelung; 5: *out-of-band*; 6: verbindungsorientiert.)

- *SOCK_STREAM* lässt sich ziemlich direkt auf TCP abbilden.
- *SOCK_STREAM* kommt den Pipelines am nächsten, wenn davon abgesehen wird, dass die Verbindungen bei Pipelines nur unidirektional sind.
- UDP wird ziemlich genau durch *SOCK_DGRAM* widergespiegelt.
- Die Varianten *SOCK_SEQPACKET* (TCP-basiert) und *SOCK_RDM* (UDP-basiert) fügen hier noch weitere Funktionalitäten hinzu. Allerdings fand *SOCK_RDM* nicht den Weg in den POSIX-Standard und wird auch von einigen Implementierungen nicht angeboten.
- Im weiteren Verlauf dieser Vorlesung werden wir uns nur mit *SOCK_STREAM*-Sockets beschäftigen.

```
int sfd = socket(domain, type, protocol);
```

- Bis zu einem gewissen Grad ist eine Betrachtung, die sich an unserem Telefonsystem orientiert, hilfreich.
- Bevor Sie Telefonanrufe entgegennehmen oder selbst anrufen können, benötigen Sie einen Telefonanschluss.
- Dieser Anschluss wird mit dem Systemaufruf *socket* erzeugt.
- Bei *domain* wird hier normalerweise *PF_INET* angegeben, um das IPv4-Protokoll auszuwählen. (Alternativ wäre etwa *PF_INET6* für IPv6 denkbar.)
- *PF* steht dabei für *protocol family*. Bei *type* kann eine der unterstützten Semantiken ausgewählt werden, also beispielsweise *SOCK_STREAM*.
- Der dritte Parameter *protocol* erlaubt in einigen Fällen eine weitere Selektion. Normalerweise wird hier schlicht 0 angegeben.

- Nachdem der Anschluss existiert, fehlt noch eine zugeordnete Telefonnummer. Um bei der Analogie zu bleiben, haben wir eine Vorwahl (IP-Adresse) und eine Durchwahl (Port-Nummer).
- Auf einem Rechner können mehrere IP-Adressen zur Verfügung stehen.
- Es ist dabei möglich, nur eine dieser IP-Adressen zu verwenden oder alle gleichzeitig, die zur Verfügung stehen.
- Bei den Port-Nummern ist eine automatische Zuteilung durch das Betriebssystem möglich.
- Alternativ ist es auch möglich, sich selbst eine Port-Nummer auszuwählen. Diese darf aber noch nicht vergeben sein und muss bei nicht-privilegierten Prozessen eine Nummer jenseits des Bereiches der wohldefinierten Port-Nummern sein, also typischerweise mindestens 1024 betragen.
- Die Verknüpfung eines Anschlusses mit einer vollständigen Adresse erfolgt mit dem Systemaufruf *bind*...

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(port);
bind(sfd, (struct sockaddr *) &address, sizeof address);
```

- Die Datenstruktur **struct** *sockaddr_in* repräsentiert Adressen für IPv4, die aus einer IP-Adresse und einer Port-Nummer bestehen.
- Das Feld *sin_family* legt den Adressraum fest. Hier gibt es passend zur Protokollfamilie *PF_INET* nur *AF_INET* (*AF* steht hier für *address family*).
- Bei dem Feld *sin_addr.s_addr* lässt sich die IP-Adresse angeben. Mit *INADDR_ANY* übernehmen wir alle IP-Adressen, die zum eigenen Rechner gehören.
- Das Feld *sin_port* spezifiziert die Port-Nummer.

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(port);
bind(sfd, (struct sockaddr *) &address, sizeof address);
```

- Da Netzwerkadressen grundsätzlich nicht von der Byte-Anordnung eines Rechners abhängen dürfen, wird mit *htonl* (*host to network long*) der 32-Bit-Wert der IP-Adresse in die standardisierte Form konvertiert. Analog konvertiert *htons*() (*host to network short*) den 16-Bit-Wert *port* in die standardisierte Byte-Reihenfolge.
- Wenn die Port-Nummer vom Betriebssystem zugeteilt werden soll, kann bei *sin_port* auch einfach 0 angegeben werden.

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(port);
bind(sfd, (struct sockaddr *) &address, sizeof address);
```

- Der Datentyp **struct** *sockaddr_in* ist eine spezielle Variante des Datentyps **struct** *sockaddr*. Letzterer sieht nur ein Feld *sin_family* vor und ein generelles Datenfeld *sa_data*, das umfangreich genug ist, um alle unterstützten Adressen unterzubringen.
- Bei *bind*() wird der von *socket*() erhaltene Deskriptor angegeben (hier *sfd*), ein Zeiger, der auf eine Adresse vom Typ **struct** *sockaddr* verweist, und die tatsächliche Länge der Adresse, die normalerweise kürzer ist als die des Typs **struct** *sockaddr*.
- Schön sind diese Konstruktionen nicht, aber C bietet eben keine objekt-orientierten Konzepte, wenngleich die Berkeley-Socket-Schnittstelle sehr wohl polymorph und damit objekt-orientiert ist.

```
listen(sfd, SOMAXCONN);
```

- Damit eingehende Verbindungen (oder Anrufe in unserer Telefon-Analogie) entgegengenommen werden können, muss *listen*() aufgerufen werden.
- Nach *listen*() kann der Anschluss „klingeln“, aber noch sind keine Vorbereitungen getroffen, das Klingeln zu hören oder den Hörer abzunehmen.
- Der zweite Parameter bei *listen*() gibt an, wieviele Kommunikationspartner es gleichzeitig klingeln lassen dürfen.
- *SOMAXCONN* ist hier das Maximum, das die jeweilige Implementierung erlaubt.

Anschlussdaten ermitteln

newsocket.c

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

void print_ip_addr(in_addr_t ipaddr) {
   if (ipaddr == INADDR_ANY) {
      printf("INADDR_ANY");
   } else {
      uint32_t addr = ntohl(ipaddr);
      printf("%u.%u.%u.%u",
         addr>>24, (addr>>16)&0xff,
         (addr>>8)&0xff, addr&0xff);
   }
}

int main() {
   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   if (sfd < 0) exit(1);
   if (listen(sfd, SOMAXCONN) < 0) exit(2);
   struct sockaddr address;
   socklen_t len = sizeof address;
   if (getsockname(sfd, &address, &len) < 0) exit(3);
   struct sockaddr_in * inaddr = (struct sockaddr_in *) &address;
   printf("This is the address of my new socket:\n");
   printf("IP address:  "); print_ip_addr(inaddr->sin_addr.s_addr);
   printf("\n");
   printf("Port number: %hu\n", ntohs(inaddr->sin_port));
}
```

```
struct sockaddr client_addr;
socklen_t client_addr_len = sizeof client_addr;
int fd = accept(sfd, &client_addr, &client_addr_len);
```

- Liegt noch kein Anruf vor, blockiert *accept*() bis zum nächsten Anruf.
- Wenn mit *accept*() ein Anruf eingeht, wird ein Dateideskriptor auf den bidirektionalen Verbindungskanal zurückgeliefert.
- Normalerweise speichert *accept*() die Adresse des Klienten beim angegebenen Zeiger ab. Wenn als Zeiger 0 angegeben wird, entfällt dies.

Ein einfacher Zeitdienst

timeserver.c

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#define PORT 11011
int main () {
   struct sockaddr_in address = {0};
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = htonl(INADDR_ANY);
   address.sin_port = htons(PORT);
   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &address,
             sizeof address) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      perror("socket"); exit(1);
   }
   int fd;
   while ((fd = accept(sfd, 0, 0)) >= 0) {
      char timebuf[32]; time_t clock; time(&clock);
      ctime_r(&clock, timebuf);
      write(fd, timebuf, strlen(timebuf)); close(fd);
   }
}
```

timeserver.c

```
if (sfd < 0 ||
     setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                &optval, sizeof optval) < 0 ||
     bind(sfd, (struct sockaddr *) &address,
          sizeof address) < 0 ||
     listen(sfd, SOMAXCONN) < 0) {
   perror("socket"); exit(1);
}
```

- Hier wird zusätzlich noch *setsockopt* aufgerufen, um die Option *SO_REUSEADDR* einzuschalten.
- Dies empfiehlt sich immer, wenn eine feste Port-Nummer verwendet wird.
- Fehlt diese Option, kann es passieren, dass bei einem Neustart des Dienstes die Port-Nummer nicht sofort wieder zur Verfügung steht, da noch alte Verbindungen nicht vollständig abgewickelt worden sind.

Ein Klient für den Zeitdienst

timeclient.c

```
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 11011
int main (int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s host\n", cmdname); exit(1);
   }
   char* hostname = *argv; struct hostent* hp;
   if ((hp = gethostbyname(hostname)) == 0) {
      fprintf(stderr, "unknown host: %s\n", hostname); exit(1);
   }
   char* hostaddr = hp->h_addr_list[0];
   struct sockaddr_in addr = {0}; addr.sin_family = AF_INET;
   memcpy((void *) &addr.sin_addr, (void *) hostaddr, hp->h_length);
   addr.sin_port = htons(PORT);
   int fd;
   if ((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket"); exit(1);
   }
   if (connect(fd, (struct sockaddr *) &addr, sizeof addr) < 0) {
      perror("connect"); exit(1);
   }
   char buffer[BUFSIZ]; ssize_t nbytes;
   while((nbytes = read(fd, buffer, sizeof buffer)) > 0 &&
         write(1, buffer, nbytes) == nbytes);
}
```

timeclient.c

```
char* hostname = *argv;
struct hostent* hp;
if ((hp = gethostbyname(hostname)) == 0) {
   fprintf(stderr, "unknown host: %s\n", hostname);
   exit(1);
}
char* hostaddr = hp->h_addr_list[0];
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;
memmove((void *) &addr.sin_addr, (void *) hostaddr, hp->h_length);
addr.sin_port = htons(PORT);
```

- Der Klient erhält über die Kommandozeile den Namen des Rechners, auf dem der Zeitdienst zur Verfügung steht.
- Für die Abbildung eines Rechnernamens in eine IP-Adresse wird die Funktion *gethostbyname*() benötigt, die im Erfolgsfalle eine oder mehrere IP-Adressen liefert, unter denen sich der Rechner erreichen lässt.
- Hier wird die erste IP-Adresse ausgewählt.

Für die Kombination aus Rechnernamen (oder alternativ einer IP-Adresse) und einer Portnummer gibt es mit RFC 2396 und RFC 2373 einen Standard:

$$
\begin{aligned}
\langle \text{hostport} \rangle &\longrightarrow \langle \text{host} \rangle \ [\ \text{„:“}\ \langle \text{port} \rangle\ ] \\
\langle \text{host} \rangle &\longrightarrow \langle \text{hostname} \rangle \\
&\longrightarrow \langle \text{IPv4address} \rangle \\
&\longrightarrow \langle \text{IPv6reference} \rangle \\
\langle \text{hostname} \rangle &\longrightarrow \{\ \langle \text{domainlabel} \rangle\ \text{„.“}\ \}\ \langle \text{toplabel} \rangle\ [\ \text{„.“}\ ] \\
\langle \text{domainlabel} \rangle &\longrightarrow \langle \text{alphanum} \rangle \\
&\longrightarrow \langle \text{alphanum} \rangle\ \{\ \langle \text{alphanum} \rangle\ \mid\ \text{„-“}\ \}\ \langle \text{alphanum} \rangle \\
\langle \text{toplabel} \rangle &\longrightarrow \langle \text{alpha} \rangle \\
&\longrightarrow \langle \text{alpha} \rangle\ \{\ \langle \text{alphanum} \rangle\ \mid\ \text{„-“}\ \}\ \langle \text{alphanum} \rangle \\
\langle \text{IPv4address} \rangle &\longrightarrow \{\ \langle \text{digit} \rangle\ \}\ \text{„.“}\ \{\ \langle \text{digit} \rangle\ \} \\
&\qquad \text{„.“}\ \{\ \langle \text{digit} \rangle\ \}\ \text{„.“}\ \{\ \langle \text{digit} \rangle\ \}
\end{aligned}
$$

$$
\begin{aligned}
\langle \text{IPv6reference} \rangle &\longrightarrow \text{„[“ } \langle \text{IPv6address} \rangle \text{ „]“} \\
\langle \text{IPv6address} \rangle &\longrightarrow \langle \text{hexpart} \rangle \text{ [„:“ } \langle \text{IPv4address} \rangle \text{]} \\
\langle \text{hexpart} \rangle &\longrightarrow \langle \text{hexseq} \rangle \\
&\longrightarrow \langle \text{hexseq} \rangle \text{ „::“ [} \langle \text{hexseq} \rangle \text{]} \\
&\longrightarrow \text{„::“ [} \langle \text{hexseq} \rangle \text{]} \\
\langle \text{hexseq} \rangle &\longrightarrow \{ \langle \text{hexdigit} \rangle \} \\
&\longrightarrow \langle \text{hexseq} \rangle \text{ „:“ } \{ \langle \text{hexdigit} \rangle \}
\end{aligned}
$$

Beispiele:

- 127.0.0.1:12345
- [::1]:12345
- 0:12345
- *theon.mathematik.uni−ulm.de*:12345

Datenstruktur für einen *hostport*

hostport.h

```
typedef struct hostport {
   /* parameters for socket() */
   int domain;
   int protocol;
   /* parameters for bind() or connect() */
   struct sockaddr_storage addr;
   socklen_t namelen;
} hostport;

bool parse_hostport(char* input, hostport* hp,
   in_port_t defaultport);
```

- In der Vorlesungsbibliothek gibt es eine Funkton *parse_hostport*, die eine Zeichenkette entsprechend der Syntax des RFC 2396 analysiert und in einer **struct** *hostport* ablegt.
- In der Datenstruktur *hostport* liegen alle Parameter, die für die Systemaufrufe *socket()*, *bind()* oder *connect()* benötigt werden.
- Mit so einer Schnittstelle lässt sich auch die Festlegung auf IPv4 oder IPv6 vermeiden.

timeclient2.c

```
hostport hp;
if (!parse_hostport(*argv, &hp, PORT)) {
   fprintf(stderr, "unknown hostport: %s\n", *argv); exit(1);
}
int fd;
if ((fd = socket(hp.domain, SOCK_STREAM, hp.protocol)) < 0) {
   perror("socket"); exit(1);
}
if (connect(fd, (struct sockaddr *) &hp.addr, hp.namelen) < 0) {
   perror("connect"); exit(1);
}
```

- Der im *hostport* verwendete Datentyp **struct** *sockaddr_storage* ist unabhängig von der Wahl eines bestimmten Netzwerks bzw. des zugehörigen Adressraums.
- Deswegen kann nicht mehr **sizeof** verwendet werden, da dieser jetzt eine Maximalgröße aufweist für alle denkbaren Varianten. Die zu verwendende Größe steht über *hp.namelen* zur Verfügung.

timeserver2.c

```
hostport hp;
if (!parse_hostport(*argv, &hp, PORT)) {
   fprintf(stderr, "unknown hostport: %s\n", *argv); exit(1);
}
int sfd;
if ((sfd = socket(hp.domain, SOCK_STREAM, hp.protocol)) < 0) {
   perror("socket"); exit(1);
}

int optval = 1;
if (setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                &optval, sizeof optval) < 0 ||
     bind(sfd, (struct sockaddr *) &hp.addr, hp.namelen) < 0 ||
     listen(sfd, SOMAXCONN) < 0) {
   perror("socket"); exit(1);
}
```

- Analog kann *parse_hostport* auch in Verbindung mit *bind* verwendet werden.

Fragmentierung der Pakete bei Netzwerkverbindungen 213

- Die Ein- und Ausgabe über Netzwerkverbindungen bringt in Vergleich zur Behandlungen von Dateien und interaktiven Benutzern einige Veränderungen mit sich.
- Wenn eine Verbindung des Typs *SOCK_STREAM* zum Einsatz gelangt, so kommen die Daten zwar in der korrekten Reihenfolge an, jedoch nicht in der ursprünglichen Paketisierung.
- Als ursprüngliche Pakete werden hier die Daten betrachtet, die mit Hilfe eines einzigen Aufrufs von *write*() geschrieben werden:

```
const char greeting[] = "Hi, how are you?\r\n";
ssize_t nbytes = write(sfd, greeting, sizeof greeting);
```

Fragmentierung der Pakete bei Netzwerkverbindungen 214

- Wenn beispielsweise bei einer Netzwerkverbindung immer vollständige Zeilen mit *write*() geschrieben werden, so ist es möglich, dass die korrespondierende *read*()-Operation nur einen Teil einer Zeile zurückliefert oder auch ein Fragment, das sich über mehr als eine Zeile erstreckt.
- Diese Problematik legt es nahe, nur zeichenweise einzulesen, wenn genau eine einzelne Zeile eingelesen werden soll:

```
char ch;
stralloc line = {0};
while (read(fd, &ch, sizeof ch) == 1 && ch != '\n') {
   stralloc_append(&line, &ch);
}
```

Fragmentierung der Pakete bei Netzwerkverbindungen 215

- Diese Vorgehensweise ist jedoch außerordentlich ineffizient, weil Systemaufrufe wie *read*() zu einem Kontextwechsel zwischen dem aufrufenden Prozess und dem Betriebssystem führen.
- Wenn ein Kontextwechsel für jedes einzulesende Byte initiiert wird, dann ist der betroffene Rechner mehr mit Kontextwechseln als mit sinnvollen Tätigkeiten beschäftigt.
- Wenn jedoch in größeren Einheiten eingelesen wird, ist möglicherweise mehr als nur die gewünschte Zeile in *buf* zu finden. Oder auch nur ein Teil der Zeile:

```
char buf[512];
ssize_t nbytes = read(fd, buf, sizeof buf);
```

Entsprechend ist eine gepufferte Eingabe notwendig, bei der die Eingabe-Operationen aus einem Puffer versorgt werden, der, wenn er leer wird, mit Hilfe einer *read*()-Operation aufzufüllen ist.

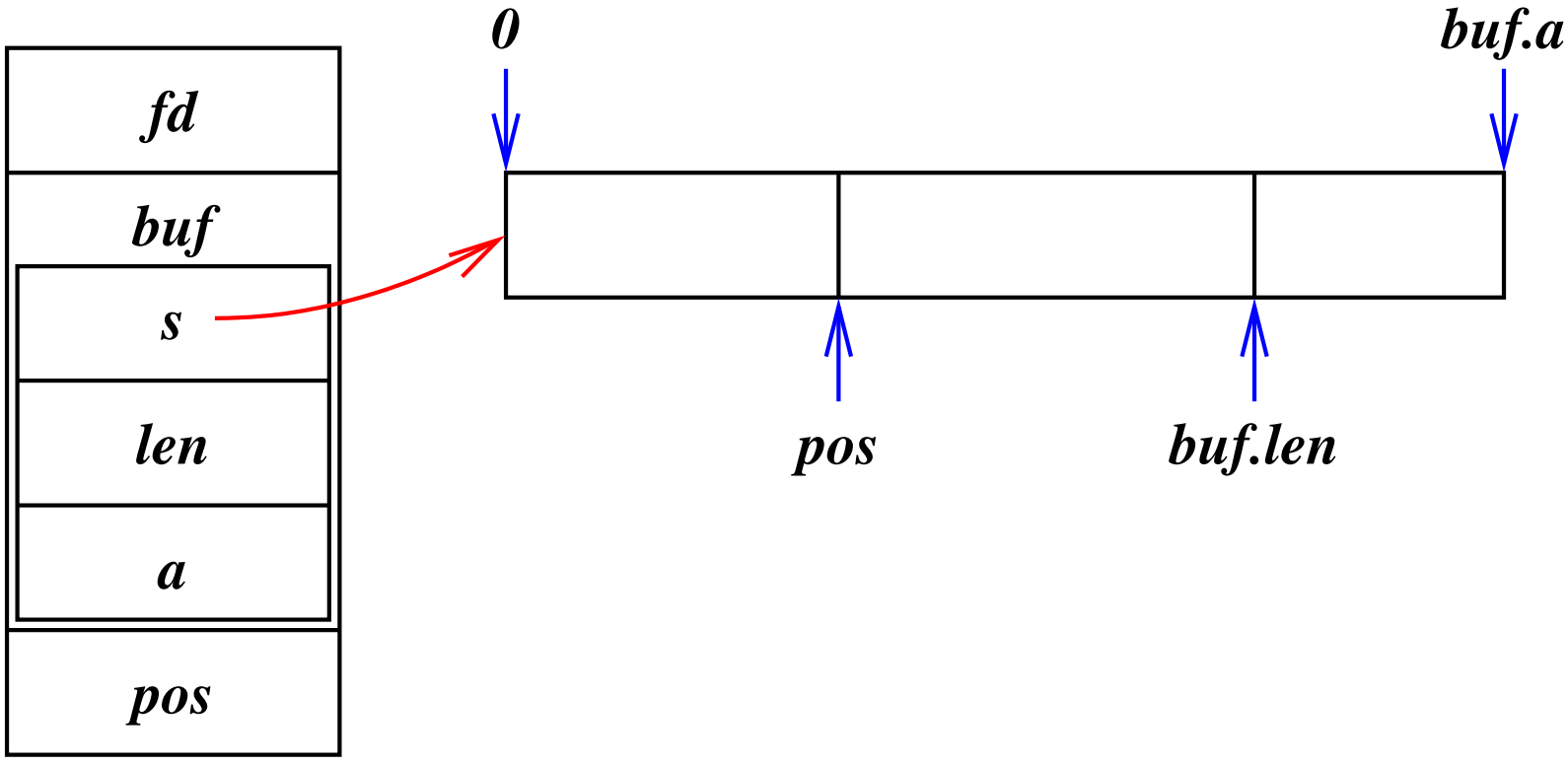
Diese Funktion kann die *stdio* jedoch nicht erfüllen:

- Es gibt keine Lesefunktion, die sich wie *read* auf den vorhandenen Buffer-Inhalt beschränkt. Dies ist jedoch für die Vermeidung ineffizienter Lese-Operationen unerlässlich.
- Systemaufrufe wie *poll* oder *select*, die für Netzwerkverbindungen häufig benötigt werden, funktionieren nur für Dateideskriptoren, nicht für *FILE*. Sie können nicht angepasst werden, da es keine Funktionen gibt, die den Füllungsstand des Buffers angibt.

- Entsprechend wird eine Alternative zur *stdio* benötigt, die für Netzwerkverbindungen geeignet ist.
- Die Datenstruktur für einen Eingabe-Puffer benötigt entsprechend einen Dateideskriptor, einen Puffer und einen Positionszeiger innerhalb des Puffers:

inbuf.h

```
typedef struct inbuf {
   int fd;
   stralloc buf;
   unsigned int pos;
} inbuf;
```


fd
buf
s
len
a
pos
0
buf.a
pos
buf.len

Schnittstelle für eine gepufferte Eingabe

inbuf.h

```
#ifndef AFBLIB_INBUF_H
#define AFBLIB_INBUF_H

#include <stdbool.h>
#include <stddef.h>
#include <stralloc.h>
#include <unistd.h>

typedef struct inbuf {
   int fd;
   stralloc buf;
   size_t pos;
} inbuf;

/* set size of input buffer */
bool inbuf_alloc(inbuf* ibuf, size_t size);

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size);

/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf);

/* move backward one position */
bool inbuf_back(inbuf* ibuf);

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf);

#endif
```

Implementierung für eine gepufferte Eingabe

inbuf.c

```
/* set size of input buffer */
bool inbuf_alloc(inbuf* ibuf, size_t size) {
   return stralloc_ready(&ibuf->buf, size);
}

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (ibuf->pos >= ibuf->buf.len) {
      if (ibuf->buf.a == 0 && !inbuf_alloc(ibuf, 512)) return -1;
      /* fill input buffer */
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = read(ibuf->fd, ibuf->buf.s, ibuf->buf.a);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return nbytes;
      ibuf->buf.len = nbytes;
      ibuf->pos = 0;
   }
   ssize_t nbytes = ibuf->buf.len - ibuf->pos;
   if (size < nbytes) nbytes = size;
   memcpy(buf, ibuf->buf.s + ibuf->pos, nbytes);
   ibuf->pos += nbytes;
   return nbytes;
}
```

inbuf.c

```
/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf) {
   char ch;
   ssize_t nbytes = inbuf_read(ibuf, &ch, sizeof ch);
   if (nbytes <= 0) return -1;
   return ch;
}

/* move backward one position */
bool inbuf_back(inbuf* ibuf) {
   if (ibuf->pos == 0) return false;
   ibuf->pos--;
   return true;
}

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf) {
   stralloc_free(&ibuf->buf);
}
```

- Die Ausgabe sollte ebenfalls gepuffert erfolgen, um die Zahl der Systemaufrufe zu minimieren.
- Ein Positionszeiger ist nicht erforderlich, wenn Puffer grundsätzlich vollständig an *write*() übergeben werden.
- Hier ist das einzige Problem, dass die *write*()-Operation unter Umständen nicht den gesamten gewünschten Umfang akzeptiert und nur einen Teil der zu schreibenden Bytes akzeptiert und entsprechend eine geringere Quantität als Wert zurückgibt.

outbuf.h

```
typedef struct outbuf {
   int fd;
   stralloc buf;
} outbuf;
```

outbuf.h

```
#ifndef AFBLIB_OUTBUF_H
#define AFBLIB_OUTBUF_H

#include <stdbool.h>
#include <stddef.h>
#include <stralloc.h>
#include <unistd.h>

typedef struct outbuf {
   int fd;
   stralloc buf;
} outbuf;

/* works like write(2) but to obuf */
ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size);

/* works like fputc but to obuf */
int outbuf_putchar(outbuf* obuf, char ch);

/* write contents of obuf to the associated fd */
bool outbuf_flush(outbuf* obuf);

/* release storage associated with obuf */
void outbuf_free(outbuf* obuf);

#endif
```

outbuf.c

```
/* works like write(2) but to obuf */
ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (!stralloc_readyplus(&obuf->buf, size)) return -1;
   memcpy(obuf->buf.s + obuf->buf.len, buf, size);
   obuf->buf.len += size;
   return size;
}

/* works like fputc but to obuf */
int outbuf_putchar(outbuf* obuf, char ch) {
   if (outbuf_write(obuf, &ch, sizeof ch) <= 0) return -1;
   return ch;
}
```

outbuf.c

```
/* write contents of obuf to the associated fd */
bool outbuf_flush(outbuf* obuf) {
   ssize_t left = obuf->buf.len; ssize_t written = 0;
   while (left > 0) {
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = write(obuf->fd, obuf->buf.s + written, left);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return false;
      left -= nbytes; written += nbytes;
   }
   obuf->buf.len = 0;
   return true;
}

/* release storage associated with obuf */
void outbuf_free(outbuf* obuf) {
   stralloc_free(&obuf->buf);
}
```

- Zwischen Dienste-Anbietern und ihren Klienten auf dem Netzwerk besteht häufig ein ähnliches Verhältnis wie zwischen einer Shell und dem zugehörigen Benutzer.
- Der Klient gibt ein Kommando, das typischerweise mit dem Zeilentrenner CR LF, beendet wird, und der Dienste-Anbieter sendet darauf eine Antwort zurück,
  - die zum Ausdruck bringt, ob das Kommando erfolgreich verlief oder fehlschlug, und
  - einen Antworttext über eine oder mehrere Zeilen bringt.
- Es gibt keine zwingende Notwendigkeit, bei einem Protokoll Zeilentrenner zu verwenden. Alternativ wäre es auch denkbar,
  - die Länge eines Pakets zu Beginn explizit zu deklarieren oder
  - Pakete fester Länge zu wählen.

```
theon$ telnet mail.rz.uni-ulm.de smtp
Trying 134.60.1.11...
Connected to mail.uni-ulm.de.
Escape character is '^]'.
220 mail.uni-ulm.de ESMTP Sendmail 8.15.2/8.15.2; Tue, 11 Jun 2019 13:24:51 +0200 (CEST)
help
214-2.0.0 This is sendmail version 8.15.2
214-2.0.0 Topics:
214-2.0.0       HELO    EHLO    MAIL    RCPT    DATA
214-2.0.0       RSET    NOOP    QUIT    HELP    VRFY
214-2.0.0       EXPN    VERB    ETRN    DSN     AUTH
214-2.0.0       STARTTLS
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation see
214-2.0.0       http://www.sendmail.org/email-addresses.html
214-2.0.0 For local information send email to Postmaster at your site.
214 2.0.0 End of HELP info
huhu
500 5.5.1 Command unrecognized: "huhu"
helo theon.mathematik.uni-ulm.de
250 mail.uni-ulm.de Hello theon.mathematik.uni-ulm.de [134.60.66.7], pleased to meet you
quit
221 2.0.0 mail.uni-ulm.de closing connection
Connection to mail.uni-ulm.de closed by foreign host.
theon$
```

```
theon$ telnet mail.rz.uni-ulm.de smtp
Trying 134.60.1.11...
Connected to mail.uni-ulm.de.
Escape character is '^]'.
220 mail.uni-ulm.de ESMTP Sendmail 8.15.2/8.15.2; Tue, 11 Jun 2019 13:24:51 +0
```

- Beim SMTP-Protokoll erfolgt zunächst eine Begrüßung des Dienste-Anbieters.
- Die Begrüßung oder auch eine andere Antwort des Anbieters besteht aus einer dreistelligen Nummer, einem Leerzeichen oder einem Minus und beliebigem Text, der durch CR LF abgeschlossen wird.
- Die erste Ziffer der dreistelligen Nummer legt hier fest, ob ein Erfolg oder ein Problem vorliegt. Die beiden weiteren Ziffern werden zur feineren Unterscheidung der Rückmeldung verwendet.
- Eine führende 2 bedeutet Erfolg, eine 4 signalisiert ein temporäres Problem und eine 5 signalisiert einen permanenten Fehler.

```
help
214-2.0.0 This is sendmail version 8.15.2
214-2.0.0 Topics:
214-2.0.0       HELO    EHLO    MAIL    RCPT    DATA
214-2.0.0       RSET    NOOP    QUIT    HELP    VRFY
214-2.0.0       EXPN    VERB    ETRN    DSN     AUTH
214-2.0.0       STARTTLS
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation see
214-2.0.0       http://www.sendmail.org/email-addresses.html
214-2.0.0 For local information send email to Postmaster at your site.
214 2.0.0 End of HELP info
```

- In der Beispielsitzung ist das erste Kommando ein „help“, gefolgt von CR LF.
- Da die Antwort sich über mehrere Zeilen erstreckt, werden alle Zeilen, hinter der noch mindestens eine folgt, mit einem Minuszeichen hinter der dreistelligen Zahl gekennzeichnet.

```
huhu
500 5.5.1 Command unrecognized: "huhu"
helo theon.mathematik.uni-ulm.de
250 mail.uni-ulm.de Hello theon.mathematik.uni-
ulm.de [134.60.66.7], pleased to meet you
quit
221 2.0.0 mail.uni-ulm.de closing connection
Connection to mail.uni-ulm.de closed by foreign host.
theon$
```

- Das unbekannte Kommando „huhu“ provoziert hier eine Fehlermeldung provoziert, die durch den Code 500 als solche kenntlich gemacht wird.
- Das SMTP-Protokoll erlaubt auch eine Fortsetzung des Dialogs nach Fehlern, so dass dann noch ein „helo“-Kommando akzeptiert wurde.
- Die Verbindung wurde mit dem „quit“-Befehl beendet.

- 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
```

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.

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);
   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.

- 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.

Verpackung von *poll* in eine Bibliothek

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, size_t len);
ssize_t read_from_link(connection* link, char* buf, size_t 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, size_t len);
ssize_t read_from_link(connection* link, char* buf, size_t 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

multiplexor.h

```
typedef struct connection {
   int fd;
   void* handle; /* may be freely used by the application */
   void* mpx_handle; /* corresponding parameter from run_multiplexor */
   /* private fields */
   void* mpx; /* internal handle */
   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;
   size_t len;
   size_t pos;
   struct output_queue_member* next;
} output_queue_member;
// ...
bool write_to_link(connection* link, char* buf, size_t 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 */
   size_t count; /* number of connections */
   struct pollfd* pollfds; /* parameter for poll() */
   size_t 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 size_t setup_polls(multiplexor* mpx) {
   size_t 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 size_t setup_polls(multiplexor* mpx) {
   /* ... */

   size_t 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.

Schreiben auf eine Verbindung

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 = true;
   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) {
   /* ignore SIGPIPE as we might receive this signal
      on writing to connections which were already
      closed by our client */
   struct sigaction sigact = {.sa_handler = SIG_IGN};
   struct sigaction old_sigact = {0};
   if (sigaction(SIGPIPE, &sigact, &old_sigact) < 0) return;

   /* ... */

   /* restore previous SIGPIPE handler */
   sigaction(SIGPIPE, &old_sigact, 0);
}
```

- *SIGPIPE* kann uns unerwartet treffen, wenn wir in eine Netzwerkverbindung schreiben, die von der Gegenseite bereits geschlossen ist.
- Entsprechend müssen wir uns dagegen wappnen und Verbindungen bei Schreibfehlern umgehend fallen lassen.

multiplexor.c

```
multiplexor mpx = {
   .socket = socket, .ohandler = open_handler,
   .ihandler = input_handler, .chandler = close_handler,
   .mpx_handle = mpx_handle, .socketok = true,
};
size_t count;
while ((count = setup_polls(&mpx)) > 0) {
   if (poll(mpx.pollfds, count, -1) <= 0) return;
   for (size_t 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*

- Bei *socket* lässt sich *SOCK_DGRAM* als zweiter Parameter angeben.
- Der Netzwerkdienst kann dann wie gewohnt *setsockopt* und *bind* aufrufen. Der Systemaufruf *listen* fällt weg, da dieser nur bei verbindungsorientierten Sockets Anwendung findet.
- Eingehende Pakete können dann mit *recvfrom* empfangen werden, das (in Ergänzung zu *read*) auch die Absenderadresse mitliefert. Mit *sendto* ist eine Antwort an eine gegebene Adresse möglich.
- Der Klient verwendet wie gewohnt *connect* und kann dann *read* und *write* verwenden, wobei hier (falls die Buffergröße groß genug ist) vollständige Pakete gelesen und verschickt werden.

timeserver.c

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(PORT);
int sfd = socket(PF_INET, SOCK_DGRAM, 0);
int optval = 1;
if (sfd < 0 ||
     setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                &optval, sizeof optval) < 0 ||
     bind(sfd, (struct sockaddr *) &address,
          sizeof address) < 0) {
   perror("socket"); exit(1);
}
ssize_t nbytes; char buf[BUFSIZ];
struct sockaddr_in sender; socklen_t sender_len = sizeof(sender);
while ((nbytes = recvfrom(sfd, buf, sizeof buf, 0,
          (struct sockaddr*) &sender, &sender_len)) >= 0) {
   char timebuf[32]; time_t clock; time(&clock);
   ctime_r(&clock, timebuf, sizeof timebuf);
   sendto(sfd, timebuf, strlen(timebuf), 0,
      (struct sockaddr*) &sender, sender_len);
}
```

timeserver.c

```
ssize_t nbytes; char buf[BUFSIZ];
struct sockaddr_in sender; socklen_t sender_len = sizeof(sender);
while ((nbytes = recvfrom(sfd, buf, sizeof buf, 0,
          (struct sockaddr*) &sender, &sender_len)) >= 0) {
   /* ... */
}
```

Im Vergleich zu *read* erwartet *recvfrom* drei weitere Parameter:

- **int** *flags*
  normalerweise 0, der Standard nennt *MSG_PEEK* (Nachricht nicht verkonsumieren), *MSG_OOB* (*out of band*) und *MSG_WAITALL* (Nachricht muss vollständig vorliegen)
- **struct** *sockaddr∗ restrict address*
  Zeiger auf den Puffer für die Absenderadresse, darf 0 sein
- *socklen_t∗ restrict address_len*
  Zeiger auf eine Variable mit der Länge von *address*, die aktualisiert wird mit der tatsächlichen Länge der Absenderadresse.

timeclient.c

```
int fd;
if ((fd = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
   perror("socket"); exit(1);
}
if (connect(fd, (struct sockaddr *) &addr, sizeof addr) < 0) {
   perror("connect"); exit(1);
}
char buffer[BUFSIZ]; ssize_t nbytes;
/* send an empty packet */
if (write(fd, buffer, 0) < 0) {
   perror("write"); exit(1);
}
/* receive response */
if ((nbytes = read(fd, buffer, sizeof buffer)) > 0) {
   write(1, buffer, nbytes);
} else {
   perror("read"); exit(1);
}
```

- Bei TCP bzw. *SOCK_STREAM* erfolgte implizit bereits ein Austausch von Paketen durch die Systemaufrufe *listen* und *connect*.
- Bei UDP bzw. *SOCK_DGRAM* fällt dies weg. Der Systemaufruf *connect* hat hier nur die Funktion, dass eine Socket fest mit einer Adresse verbunden wird, d.h. es kann anschließend wie gewohnt *read* und *write* verwendet werden ohne eine weitere Spezifikation der Adresse.
- Wenn *connect* wegfällt, muss die Adresse beim Paketversand immer angegeben werden.
- Da *connect* bei *SOCK_DGRAM* noch nicht zum Austausch von Paketen führt, kann zu diesem Zeitpunkt noch nicht festgestellt werden, ob die Gegenseite den Dienst überhaupt anbietet. Das stellt sich erst (mit etwas Glück) bei einem anschließenden *write* heraus.
- Anders als bei TCP bzw. *SOCK_STREAM* kann das Versenden leerer Pakete sinnvoll sein.

timeclient2.c

```
/* receive response */
struct pollfd pollfds[1] = { {.fd = fd, .events = POLLIN} };
unsigned int attempts = 0;
while (attempts < 10 &&
       poll(pollfds, 1, 100 /* milliseconds */) == 0) {
   ++attempts;
   /* resend package */
   if (send(fd, buffer, 0, 0) < 0) {
      perror("send"); exit(1);
   }
}
if (attempts < 10) {
   if ((nbytes = recv(fd, buffer, sizeof buffer, 0)) > 0) {
      write(1, buffer, nbytes);
   } else {
      perror("recv");
   }
}
```

- UDP-Pakete können verloren gehen. Entsprechend sollte damit gerechnet werden, dass keine Antwort auf eine Anfrage eingeht. Entsprechend ist es sinnvoll, mehrere Versuche einzuplanen.

- Neben *AF_INET* und *AF_INET6* wird durch den POSIX-Standard auch noch *AF_UNIX* genannt.
- *PF_UNIX* bzw. *AF_UNIX* stehen für UNIX-Domain-Sockets.
- Ähnlich zu den Pipes bieten sie eine Interprozess-Kommunikation innerhalb eines Rechners auf Basis der BSD-Socket-Schnittstelle an.
- Anders als Pipes sind sie bidirektional.
- Als Adresse wird ein Dateiname verwendet. Eine Socket-Datei wird durch einen entsprechenden *bind*-Systemaufruf implizit erzeugt. Die Datei wird aber nicht automatisch entfernt, wenn der Dienst endet.
- Das Dateisystem kann hier den Zugriffsschutz übernehmen.

- Für UNIX-Domain-Sockets gibt es aus **#include** <sys/un.h> die Datenstruktur **struct** *sockaddr_un* mit folgenden Komponenten:

sa_family_t sun_family	Adressfamilie, hier immer *AF_UNIX*
char *sun_path*[]	Pfadname der Socket-Datei

- Die maximale Länge des Pfadnamens ist sehr begrenzt, typischerweise liegt das Limit bei ca. 100 Bytes.
- Bei *bind* kann ein Zeiger auf eine entsprechende Datenstruktur übergeben werden.
- Noch einfacher ist es, die Funktion *parse_hostport* entsprechend zu erweitern, so dass auch Pfadnamen unterstützt werden.

UNIX-Domain-Sockets als Alternative zu Pipes 300

UNIX-Domain-Sockets können alternativ zu Pipes verwendet werden:

- Statt *pipe* ist *socketpair* zu verwenden, das analog zwei miteinander verbundene Sockets mit einem Systemaufruf erzeugt:

```
int sfds[2];
if (socketpair(PF_UNIX, SOCK_SEQPACKET, 0, sfds) < 0) {
   /* failure ... */
}
```

- Typischerweise unterstützt *socketpair* ausschließlich UNIX-Domain-Sockets. Statt *SOCK_SEQPACKET* kann natürlich auch *SOCK_STREAM* oder *SOCK_DGRAM* verwendet werden, wobei letzteres keine Vorteile bietet.
- Wie bei Pipes sollte jede Seite nur ein Ende verwenden und das andere schließen. Anders als bei Pipes sind beide Enden voll bidirektional.
- Über UNIX-Domain-Sockets können auch geöffnete Dateideskriptoren versandt werden...

Grundsätzlich können bei Sockets statt *write* und *read* die allgemeineren Systemaufrufe *sendmsg* und *recvmsg* verwendet werden:

- *ssize_t sendmsg*(**int** *socket*, **const struct** *msghdr* ∗*message*, **int** *flags*);
- Die Datenstruktur **struct** *msghdr* bietet folgende Komponenten:

void∗ *msg_name*	Adresse (optional)
socklen_t msg_namelen	Länge der Adresse
struct *iovec*∗ *msg_iov*	Array mit I/O-Puffern
int *msg_iovlen*	Länge des Arrays
void∗ *msg_control*	Zusatzdaten
socklen_t msg_controllen	Länge der Zusatzdaten
int *msg_flags*	Flags bei empfangenen Nachrichten

Optional können Zusatzdaten beigefügt werden. Diese Daten werden beidseits inhaltlich interpretiert. Die wichtigste (und wohl einzige portable) Anwendung ist für die Übertragung von Dateideskriptoren:

- Zusatzdaten bestehen aus mehreren im Speicher unmittelbar hintereinander liegenden Datenbereichen, bei der jeder Bereich mit der Header-Datenstruktur **struct** *cmsghdr* beginnt.
- Header-Datenstruktur:

socklen_t cmsg_len	Länge der Zusatzdaten einschließlich des Headers
int *cmsg_level*	Protokollebene
int *cmsg_type*	Art der Zusatzdaten

- Für Dateideskriptoren sollte *cmsg_level* auf *SOL_SOCKET* und *cmsg_type* auf *SCM_RIGHTS* gesetzt werden.

Versand eines Dateideskriptors

transmit_fd.c

```
struct fd_cmsg {
   struct cmsghdr cm;
   int fd;
};

ssize_t send_fd_and_message(int sfd, int fd, void* buf, size_t buflen) {
   struct fd_cmsg cmsg = {
      .cm = {
         .cmsg_len = sizeof cmsg,
         .cmsg_level = SOL_SOCKET,
         .cmsg_type = SCM_RIGHTS
      },
      .fd = fd
   };
   struct iovec iovec[1] = {
      {
         .iov_base = buf,
         .iov_len = buflen
      }
   };
   struct msghdr msg = {
      .msg_iov = iovec,
      .msg_iovlen = sizeof(iovec)/sizeof(iovec[0]),
      .msg_control = &cmsg.cm,
      .msg_controllen = sizeof cmsg,
   };
   return sendmsg(sfd, &msg, /* flags = */ 0);
}
```

Empfang eines Dateideskriptors

transmit_fd.c

```
ssize_t recv_fd_and_message(int sfd, int* fd_ptr, void* buf, size_t buflen) {
   struct fd_cmsg cmsg = {{0}};
   struct iovec iovec[1] = {
      {
         .iov_base = buf,
         .iov_len = buflen
      }
   };
   struct msghdr msg = {
      .msg_iov = iovec,
      .msg_iovlen = sizeof(iovec)/sizeof(iovec[0]),
      .msg_control = &cmsg.cm,
      .msg_controllen = sizeof cmsg,
   };
   ssize_t nbytes = recvmsg(sfd, &msg, MSG_WAITALL);
   if (nbytes < 0) return -1;
   if (fd_ptr) *fd_ptr = cmsg.fd;
   return nbytes;
}
```

hostport.c

```
bool parse_hostport(char* input, hostport* hp, in_port_t defaultport) {
   if (input[0] == '/' || input[0] == '.') {
      /* special case: UNIX domain socket */
      hp->domain = PF_UNIX;
      hp->protocol = 0;
      struct sockaddr_un* sp = (struct sockaddr_un*) &hp->addr;
      sp->sun_family = AF_UNIX;
      strncpy(sp->sun_path, input, sizeof sp->sun_path);
      hp->namelen = sizeof(struct sockaddr_un);
      return true;
   }
   // regular hostports ...
}
```

- Wegen der objekt-orientierten Socket-Schnittstelle genügt eine entsprechende Erweiterung der *parse_hostport*-Funktion, um UNIX-Domain-Sockets zu unterstützen.

```
thales$ ls
MXP             lockmanager.o  mxprequest.c   mxpresponse.h  mxpsession.o
Makefile        mutexd         mxprequest.h   mxpresponse.o
lockmanager.c   mutexd.c       mxprequest.o   mxpsession.c
lockmanager.h   mutexd.o       mxpresponse.c  mxpsession.h
thales$ mutexd ./socket &
[1] 3000
thales$ ls -l socket
srwxrwxr-x 1 borchert sai 0 Jul  6 13:42 socket
thales$ cd ../connect
thales$ connect ../mutexd-multiplexed/socket
S
id Andreas
Swelcome
quit
thales$
```

- Für interaktiv nutzbare Netzwerkdienste stand *telnet* zur Verfügung. Dieser lässt sich aber nicht für UNIX-Domain-Sockets verwenden.
- Entsprechend wird ein verallgemeinerter Ansatz namens *connect* benötigt...

connect.c

```
int main(int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s hostport\n", cmdname);
      exit(1);
   }
   char* hostport_string = *argv++; --argc;
   hostport hp;
   if (!parse_hostport(hostport_string, &hp, 0)) {
      fprintf(stderr, "%s: invalid hostport: %s\n", cmdname,
         hostport_string);
      exit(1);
   }
   int sfd = socket(hp.domain, SOCK_STREAM, hp.protocol);
   if (sfd < 0) {
      perror("socket"); exit(1);
   }
   if (connect(sfd, (struct sockaddr*) &hp.addr, hp.namelen) < 0) {
      perror(hostport_string); exit(1);
   }
   // ...
}
```

connect.c

```
struct pollfd fds[] = {
   {sfd, POLLIN, 0}, /* wait for input from the socket */
   {0, POLLIN, 0}, /* wait for input from stdin */
};
char buf[BUFSIZ];
while (poll(fds, sizeof(fds)/sizeof(fds[0]), -1) > 0) {
   for (int index = 0; index < sizeof(fds)/sizeof(fds[0]); ++index) {
      if (fds[index].revents) {
         ssize_t nbytes = read(fds[index].fd, buf, sizeof buf);
         if (nbytes < 0) { perror("read"); exit(1); }
         if (nbytes == 0) exit(0);
         int outfd = (index == 0? 1: sfd);
         size_t written = 0;
         while (written < nbytes) {
            ssize_t outbytes = write(outfd,
               buf + written, nbytes - written);
            if (outbytes < 0) {
               perror("write"); exit(1);
            }
            written += outbytes;
         }
      }
   }
}
```

Es gibt zahlreiche Techniken zur lokalen Kommunikation und Synchronisation:

- *pipe*: unidirektional, Prozesse müssen miteinander verwandt sein.
- Benannte Pipes: über das Dateisystem, die Pipe-Datei muss explizit angelegt werden.
- UNIX-Domain-Sockets: bidirektional, deutlich einfacher im Vergleich zu benannten Pipes.
- Message Queues mit *msgsnd*, *msgrcv* etc. – sehr unhandlich wie alle System-V-IPC-Mechanismen
- Gemeinsame Speicherbereiche mit *mmap* – da fehlt noch die Synchronisierung...

- Speicherbereiche können auch von nicht miteinander verwandten Prozessen gemeinsam genutzt werden.
- Hierzu genügt es, mit *mmap* eine Datei in den eigenen Speicherbereich abzubilden.
- Vorteil: Das Hin- und Herkopieren kann minimiert werden.
- Nachteile:
  - Die Größe des gemeinsamen Speicherbereichs wird zu Beginn festgelegt. Dieser kann später nicht ohne weiteres wachsen.
  - Die Synchronisierung muss auf irgendeine andere Weise erreicht werden.

Zur Synchronisation von Prozessen auf dem gleichen Rechner bieten sich u.a. folgende Techniken an:

- Über das Dateisystem, etwa mit *link* (siehe erstes *mutexd*-Beispiel) oder mit *flock*. Nachteil: Wie warten wir darauf, dass uns der Partner etwas mitgeteilt hat?
- Semaphores aus dem System-V-IPC-Mechanismen (ebenso sehr unhandlich). Nachteil wie oben.
- Andere Kommunikation mit impliziter Synchronisierung
- Mutex- und Bedingungsvariablen der POSIX-Threads-Schnittstelle

Letzteres ist vielleicht überraschend. Interessanterweise können Mutex- und Bedingungsvariablen der POSIX-Threads-Schnittstelle auch von mehreren Prozessen mit Hilfe gemeinsamer Speicherbereiche genutzt werden.

Schnittstelle für Mutex-Variablen

shared_mutex.h

```
#include <stdbool.h>
#include <pthread.h>

typedef pthread_mutex_t shared_mutex;

bool shared_mutex_create(shared_mutex* mutex);
bool shared_mutex_free(shared_mutex* mutex);
bool shared_mutex_lock(shared_mutex* mutex);
bool shared_mutex_unlock(shared_mutex* mutex);
```

- Da besondere Vorkehrungen zu treffen sind, damit POSIX-Mutex-Variablen in gemeinsamen Speicherbereichen funktionieren, lohnt sich eine besondere Schnittstelle.
- Die Funktionen *shared_mutex_create* und *shared_mutex_free* dürfen nur von einem einzigen Prozess aufgerufen werden – typischerweise demjenigen, der den gemeinsamen Speicher vorbereitet, bevor die anderen Prozesse ihn in ihren Adressraum abbilden.

shared_mutex.c

```
bool shared_mutex_create(shared_mutex* mutex) {
   pthread_mutexattr_t mxattr;
   pthread_mutexattr_init(&mxattr);
   bool ok = true;
   if (pthread_mutexattr_setpshared(&mxattr, PTHREAD_PROCESS_SHARED)) {
      ok = false;
   }
   if (ok && pthread_mutex_init(mutex, &mxattr)) {
      ok = false;
   }
   pthread_mutexattr_destroy(&mxattr);
   return ok;
}
```

- Mit Hilfe von Mutex-Variablen können mehrere Parteien sichergehen, dass nur ein Prozess Zugang zu einer Ressource hat.
- Eine Mutex-Variable wird mit *pthread_mutex_init* initialisiert.
- Als einziges Attribut wird hier *PTHREAD_PROCESS_SHARED* gesetzt. Dies muss gesetzt sein, wenn die Mutex-Variable von mehreren Prozessen gemeinsam genutzt wird.

Weitere Operationen für Mutex-Variablen

shared_mutex.c

```
bool shared_mutex_free(shared_mutex* mutex) {
   return pthread_mutex_destroy(mutex) == 0;
}

bool shared_mutex_lock(shared_mutex* mutex) {
   return pthread_mutex_lock(mutex) == 0;
}

bool shared_mutex_unlock(shared_mutex* mutex) {
   return pthread_mutex_unlock(mutex) == 0;
}
```

- Die weiteren Operationen können direkt übernommen werden.
- Wenn alles in Ordnung läuft, wird jeweils 0 zurückgegeben. Sonst handelt es sich um einen Fehlercode.
- Durch den Aufruf von *pthread_mutex_lock* wird der Aufrufer blockiert, bis die Mutex-Variable frei ist.
- Danach ist sie vom Aufrufer belegt, bis sie mit *pthread_mutex_unlock* wieder freigegeben wird.

Wenn ein Prozess auf eine Datenstruktur im gemeinsamen Speicherbereich zugreifen möchte, dann sollte das nur über die in der Datenstruktur integrierte Mutex-Variable erfolgen:

- Zu Beginn ist *shared_mutex_lock* aufzurufen,
- dann kann im sogenannten *kritischen Bereich* ein Zugriff auf die Datenstruktur erfolgen, wonach
- mit *shared_mutex_unlock* der Lock wieder freizugeben ist.

Es muss hier darauf geachtet werden, dass der kritische Bereich nicht versehentlich ohne eine Freigabe des Locks verlassen wird.

shared_cv.h

```
#include <pthread.h>
#include <afblib/shared_mutex.h>

typedef pthread_cond_t shared_cv;

bool shared_cv_create(shared_cv* cv);
bool shared_cv_free(shared_cv* cv);

bool shared_cv_wait(shared_cv* cv, shared_mutex* mutex);
bool shared_cv_notify_one(shared_cv* cv);
bool shared_cv_notify_all(shared_cv* cv);
```

- Bedingungsvariablen erlauben es, auf ein Ereignis zu warten, das mit einem der *notify*-Funktionen signalisiert wird.
- Wie bei Mutex-Variablen darf das Anlegen und Abbauen nur von einem einzigen Prozess vorgenommen werden.

shared-counter.c

```
struct shared_counter {
   shared_mutex mutex;
   unsigned int counter;
};
```

Das folgende triviale Beispiel zeigt, wie

- ein gemeinsamer Speicherbereich angelegt wird für diese Datenstruktur,
- wie dieser über *fork* an Kindprozesse weitervererbt wird,
- die dann konkurrierend darauf zugreifen und über die Mutex-Variable sich jeweils einen exklusiven Zugang sichern.

Beispiel zu Mutex-Variablen

shared-counter.c

```
/* create shared memory region */
void* sm = mmap(0, sizeof(struct shared_counter),
   PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, 0);
if (sm == MAP_FAILED) {
   perror("mmap"); exit(1);
}

/* initialize shared counter */
struct shared_counter* scnt = (struct shared_counter*) sm;
if (!shared_mutex_create(&scnt->mutex)) {
   perror("mutex"); exit(1);
}
scnt->counter = 0;
```

- Mit *MAP_ANON* wird implizit */dev/zero* als abzubildende Datei gewählt. Dies gehört nicht zum Umfang von POSIX, wird aber weitgehend unterstützt (einschließlich Linux und Solaris).

shared-counter.c

```
/* create some processes who inherit the shared memory region */
for (unsigned int i = 0; i < 10; ++i) {
   pid_t pid = fork();
   if (pid < 0) {
      perror("fork"); break;
   }
   if (pid == 0) {
      srand(getpid() ^ time(0));
      for (unsigned int i = 0; i < 5; ++i) {
         shared_mutex_lock(&scnt->mutex);
         unsigned int increment = (unsigned int) (1 + rand() % 10);
         printf("[%5d] counter = %u, incremented by %u\n",
            (int)getpid(), scnt->counter, increment);
         scnt->counter += increment;
         shared_mutex_unlock(&scnt->mutex);
      }
      exit(0);
   }
}
```

shared-counter.c

```
        shared_mutex_lock(&scnt->mutex);
        unsigned int increment = (unsigned int) (1 + rand() % 10);
        printf("[%5d] counter = %u, incremented by %u\n",
           (int)getpid(), scnt->counter, increment);
        scnt->counter += increment;
        shared_mutex_unlock(&scnt->mutex);
```

- Der „kritische Bereich“ zwischen *shared_mutex_lock* und *shared_mutex_unlock* wird immer nur von maximal einem der Prozesse betreten.
- Weitere Prozesse werden ggf. in eine Warteschlange eingereiht, bis der Vorgänger *shared_mutex_unlock* aufruft.

shared-counter.c

```
/* wait for all childs to finish */
int wstat;
while (wait(&wstat) > 0);

printf("final value of the counter: %u\n", scnt->counter);

/* finish everything */
shared_mutex_free(&scnt->mutex);
munmap(sm, sizeof(struct shared_counter));
```

- Mit *wait*-Schleife synchronisieren wir uns mit dem Abschluss aller Kindprozesse.
- Danach wird die Mutex-Variable abgebaut und der gemeinsame Speicherbereich freigegeben.

Anlegen einer POSIX-Bedingungsvariablen

shared_cv.c

```
bool shared_cv_create(shared_cv* cv) {
   pthread_condattr_t condattr;
   pthread_condattr_init(&condattr);
   bool ok = true;
   if (pthread_condattr_setpshared(&condattr,
         PTHREAD_PROCESS_SHARED)) {
      ok = false;
   }
   if (ok && pthread_cond_init(cv, &condattr)) {
      ok = false;
   }
   pthread_condattr_destroy(&condattr);
   return ok;
}
```

- Wie bei den Mutex-Variablen müssen Bedingungsvariablen, die von mehreren Prozessen gemeinsam genutzt werden, mit dem Attribut *PTHREAD_PROCESS_SHARED* angelegt werden.

shared_cv.c

```
bool shared_cv_wait(shared_cv* cv, shared_mutex* mutex) {
   return pthread_cond_wait(cv, mutex) == 0;
}
```

- Die Operation *pthread_cond_wait* erfolgt immer in Verbindung mit einer Mutex-Variablen, die bereits mit *pthread_mutex_lock* reserviert sein muss.
- In einer atomaren Operation wird dann die Mutex-Variable freigegeben und der aufrufende Prozess (bzw. Thread) in die zugehörige Warteschlange eingereiht.

Weitere Operationen für POSIX-Bedingungsvariablen

shared_cv.c

```
bool shared_cv_free(shared_cv* cv) {
   return pthread_cond_destroy(cv) == 0;
}

bool shared_cv_notify_one(shared_cv* cv) {
   return pthread_cond_signal(cv) == 0;
}

bool shared_cv_notify_all(shared_cv* cv) {
   return pthread_cond_broadcast(cv) == 0;
}
```

- Bei *pthread_cond_signal* wird genau ein Prozess bzw. Thread aufgeweckt und aus der Warteschlange entfernt.
- Bei *pthread_cond_broadcast* wird die gesamte Warteschlange geleert und alle wartenden Prozesse bzw. Threads aufgeweckt.

Bei Ringpuffern handelt es sich um eine klassische Datenstruktur, die den konkurrierenden Lese- und Schreibzugriff erlaubt. Die Datenstruktur besteht aus einem festdimensionierten Array von Nachrichten und den folgenden Variablen:

- *read_index*: hier ist die nächste Nachricht zu lesen
- *write_index*: hier ist das nächste Nachricht zu schreiben
- *filled*: der aktuelle Füllgrad

Hierbei gilt, dass

- beim Schreiben darauf gewartet werden muss, bis der Füllgrad unter dem Maximum liegt und
- bei Lesen darauf zu warten ist, dass der Füllgrad positiv ist.

shared-ringbuffer.c

```
#define RINGBUF_SIZE (4)

struct shared_ringbuffer {
   shared_mutex mutex;
   shared_cv ready_for_reading;
   shared_cv ready_for_writing;
   struct message ringbuf[RINGBUF_SIZE];
   unsigned int filled;
   unsigned int write_index;
   unsigned int read_index;
};
```

- *mutex* sichert den exklusiven Zugang auf den Ringpuffer.
- Mit der Bedingungsvariablen *ready_for_reading* kann gewartet werden, bis der Füllgrad positiv ist und
- mit *ready_for_writing* kann darauf gewartet werden, dass der Füllgrad unter dem Maximum liegt.

shared-ringbuffer.c

```
static bool init_ringbuffer(struct shared_ringbuffer* rb) {
   if (!shared_mutex_create(&rb->mutex) ||
         !shared_cv_create(&rb->ready_for_reading) ||
         !shared_cv_create(&rb->ready_for_writing)) {
      return false;
   }
   rb->filled = rb->write_index = rb->read_index = 0;
   return true;
}
```

- Wie zuvor darf nur ein Prozess, die Mutex- und die Bedingungsvariablen anlegen.

Versenden einer Nachricht

shared-ringbuffer.c

```
static void send_message(struct shared_ringbuffer* rb,
      struct message* mp) {
   shared_mutex_lock(&rb->mutex);
   while (rb->filled == RINGBUF_SIZE) {
      shared_cv_wait(&rb->ready_for_writing, &rb->mutex);
   }
   rb->ringbuf[rb->write_index] = *mp;
   rb->write_index = (rb->write_index + 1) % RINGBUF_SIZE;
   ++rb->filled;
   shared_mutex_unlock(&rb->mutex);
   shared_cv_notify_one(&rb->ready_for_reading);
}
```

- Alle Operationen finden nur mit exklusivem Zugang statt. Einzige Ausnahme sind die *notify*-Funktionen bei Bedingungsvariablen, die ohne Vorkehrungen konkurrierend genutzt werden können.
- Solange der Ringpuffer voll ist, warten wir darauf, dass jemand eine Nachricht daraus empfängt.
- Sobald wir die Nachricht abgelegt haben, wecken wir mit *shared_cv_notify_one* maximal einen Prozess auf, der darauf wartete, etwas zu empfangen.

shared-ringbuffer.c

```
static void receive_message(struct shared_ringbuffer* rb,
      struct message* mp) {
   shared_mutex_lock(&rb->mutex);
   while (rb->filled == 0) {
      shared_cv_wait(&rb->ready_for_reading, &rb->mutex);
   }
   *mp = rb->ringbuf[rb->read_index];
   rb->read_index = (rb->read_index + 1) % RINGBUF_SIZE;
   --rb->filled;
   shared_mutex_unlock(&rb->mutex);
   shared_cv_notify_one(&rb->ready_for_writing);
}
```

- Solange der Ringpuffer leer ist, warten wir darauf, dass ein anderer Prozess eine Nachricht sendet.
- Sobald wir eine Nachricht entnommen haben, wecken wir mit *shared_cv_notify_one* maximal einen Prozess auf, der darauf wartete, etwas zu senden.

Die POSIX-Funktion *pthread_cond_wait* umfasst drei Operationen:

1. Freigabe der angegebenen Mutex-Variablen.
2. Warten bis auf das Eintreffen einer Notifikation über die Bedingungsvariablen mit *pthread_cond_signal* oder *pthread_cond_broadcast*.
3. Warten, bis die Mutex-Variable wieder gelockt ist.

Hierbei sind die beiden ersten Operationen atomar, d.h. der Mutex wird erst freigegeben, wenn der wartende Prozess in der Warteschlange eingetragen ist.

Denkbares Szenario, wenn *pthread_cond_wait* diese Atomizität nicht zusichern würde. Es sind die Prozesse P_1 und P_2 beteiligt und der Ringpuffer sei zu Beginn im initialen Zustand:

P_1	P_2
Aufruf von *receive_message* und Sichern des exklusiven Zugangs. **while** (*rb*−>*filled* == 0){	
	Aufruf von *send_message* und Warten auf den exklusiven Zugang.
shared_cv_wait(&*rb*−>*ready_for_reading*, &*rb*−>*mutex*); *rb*−>*mutex* wird freigegeben	
	wacht auf und hat exklusiven Zugang, fügt ein Element hinzu, erhöht *rb*−>*filled* um 1, gibt den Lock wieder frei *shared_cv_notify_one*(&*rb*−>*ready_for_reading*); (dies verpufft wirkungslos, da die Warteschlange noch leer ist)
Eintrag in die Warteschlange und langes Warten, obwohl der Ringpuffer nicht mehr leer ist.	

Gelegentlich wird **if** statt **while** verwendet. Warum jedoch **while** notwendig ist, zeigt folgendes Szenario mit den Prozessen P_1, P_2 und P_3:

P_1	P_2	P_3
Aufruf von *receive_message*, Gewinnung des exklusiven Zugangs. Feststellung, dass *filled* gleich 0 ist. Aufruf von *shared_cv_wait*, wobei der Lock freigegeben wird und P_1 in der Warteschlange ist.		
	Aufruf von *send_message*, Gewinnung des exklusiven Zugangs, Eintrag einer Nachricht, Freigabe des Locks.	
		Aufruf von *receive_message*, Gewinnung des exklusiven Zugangs, Feststellung, dass *filled* positiv ist. Entnahme der Nachricht und Freigabe des Locks.
	Aufruf von *shared_cv_notify_one*	
Aufwachen und Feststellung, dass *filled* immer noch 0 ist.		

Die Vorlesung fokussierte auf den Abstraktionen des POSIX-Standards. Diese haben aber durchaus einige Defizite und Probleme, die von diversen UNIX-Varianten bei ihrer Weiterentwicklung unterschiedlich gelöst worden sind und bislang nicht Teil des POSIX-Standards wurden.

Wir fokussieren uns hier insbesondere auf die Schwächen der *select*- und *poll*-Schnittstelle.

Die Schnittstellen *select* und *poll* haben zahlreiche Nachteile:

- Bei einer großen Zahl von n Dateideskriptoren, gibt es bei jedem Aufruf einen Aufwand von $O(n)$, um die Datenstruktur im Kern aufzubauen und kurz danach wieder abzubauen, obwohl bei vielen Anwendungen die Menge der Dateideskriptoren und der für uns relevanten Ereignisse sich nicht sehr rasch ändert. Auch die Anwendung hat einen Aufwand von $O(n)$ um herauszufinden, welches Ereignis eingetreten ist.
- Sie sind beschränkt auf I/O-Ereignisse. Ereignisse im Kontext der Prozesse (z.B. Terminierung eines Kindprozesses) oder im Dateisystem sind nicht integriert.
- Probleme bei der Parallelisierung zur Skalierung: *C10k problem* in Verbindung mit dem *thundering herd problem*.

Einige Begriffsdefinitionen für die folgenden Diskussionen:

- **Ereignis**: Beliebige Zustandsveränderungen im Kernel, die erfasst werden können, z. B. Eintreffen eines Netzwerkpakets, Eröffnen einer Verbindung, Terminierung eines Prozesses, Zustellung eines Signals usw.
- **Filter**: Spezifikation einer Klasse von Ereignissen, z.B. ein Dateideskriptor und das Eintreffen einer Eingabe.
- **Filterliste**: Menge von beliebig vielen Filtern, die beschreiben, an welchen Ereignissen Interesse besteht.
- **Aggregation**: Mehrere Ereignisse der gleichen Art (z.B. aufeinanderfolgende Pakete bei der gleichen Netzwerkverbindung) können u.U. zu einem Ereignis aggregiert werden.

- Die Filter bestehen jeweils nur aus Dateideskriptoren und den zugehörigen Ereignismasken (z.B. *POLLIN* und *POLLOUT*).
- Die möglicherweise sehr umfangreiche Filterliste wird bei jedem Aufruf von *poll* und *select* übergeben.
- Nach dem Aufruf beginnt die umfangreiche Suche nach den Ereignissen, die stattgefunden haben.
- Sowohl beim Kernel als auch bei der Anwendung haben wir einen Aufwand von $O(n)$, wenn n der Umfang der Filterliste ist.

Idee: Wir verlagern die Verwaltung der Filterliste in den Kernel.
Vorgehensweise:

- Zunächst muss eine Filterliste im Kernel angelegt werden.
- Es gibt Systemaufrufe, die das Hinzufügen und Löschen von Filtern in der Filterliste erlauben.
- Es ist möglich, auf das Eintreffen eines von der Filterliste erfassten Ereignisses zu warten.

poll und *select* sind zustandslos (*stateless*), d.h. die Filterlisten sind nur während eines Aufrufs von *poll* oder *select* relevant.

Wenn jetzt eine im Kernel existierende Filterliste unabhängig von einem aktuell wartenden Systemaufruf ein Ereignis erfasst, was soll dann geschehen?

- Soll das Ereignis aufgezeichnet werden?
- Können Ereignisse, die den gleichen Filter betreffen, aggregiert werden?
- Was passiert, wenn mehrere Filterlisten sich für das gleiche Ereignis interessieren?
- Was passiert bei *fork*?

Aus der Elektrotechnik werden hier gerne die Begriffe pegelgesteuert (*level-triggered*) und flankengesteuert (*edge-triggered*) verwendet.

Angenommen, wir haben einen Eingangssignal, bei dem entweder Spannung anliegt oder nicht. Was soll dann als Ereignis betrachtet werden?

- Pegelgesteuert (*level-triggered*): Das Ereignis ist immer dann zu erfassen, wenn wir nachsehen und die Spannung zu diesem Zeitpunkt anliegt.
- Flankengesteuert (*edge-triggered*): Nur der Vorgang des Hochgehens der Spannung wird als Ereignis betrachtet. Das Ereignis wird somit nur ein einziges Mal erfasst.

- Eine flankengesteuerte Ereignisbehandlung setzt einen entsprechenden Zustand im Kernel voraus.
- Da sowohl *poll* als auch *select* zustandslos sind, können sie nur pegelgesteuert erfolgen:
  - Beim Aufruf sehen wir nach, ob bei einem der Filter der „Pegel anliegt“ (z.B. vorhandene Eingabe).
  - Falls ja, enden *poll* bzw. *select* sofort und melden dies als Ereignisse.
  - Falls nein, warten wir darauf, dass bei einem der Filter der Pegel hochgeht.

Wenn die Filterliste in den Kernel wandert, haben wir folgende Möglichkeiten:

- Wir können die Filterliste vollkommen ignorieren, wenn wir uns nicht in einem Systemaufruf befinden, der auf das Eintreffen eines Ereignisses dieser Filterliste wartet.
- Oder wir können von einer im Kernel existierenden Filterliste erfasste Ereignisse laufend aufzeichnen und in eine Warteschlange einreihen, auf die zugegriffen wird, sobald die nächste Abfrage- oder Warte-Operation stattfindet.

Es ergeben sich daraus unterschiedliche Ansätze in der Programmierung:

- pegelgesteuert: Nachdem wir auf das Eintreffen der Ereignisse gewartet haben, müssen wir diese verkonsumieren (d.h. den Pegel wegnehmen), bevor wir erneut warten. Sonst erhalten wir die gleichen Ereignisse erneut.
- flankengesteuert: Wenn wir gewartet haben, müssen wir darauf achten, alle eingetroffenen Ereignisse vollständig abzuarbeiten, weil diese sonst verlorengehen können. Eine Warteoperation mit einem Filter, der auf Eingabe wartet, kann dann auch trotz existierender Eingabe blockieren, weil es schon eine Gelegenheit gab, diese abzugreifen.

Bei konkurrierenden Filterlisten im Kernel gibt es je nach Ansatz unterschiedliche Probleme bzw. Fragestellungen:

- pegelgesteuert: Nach dem Eintreffen eines Ereignisses werden möglicherweise sehr viel mehr Prozesse bzw. Threads mit der Abarbeitung beschäftigt, obwohl einer das hätte alleine behandeln können (*thundering herd problem*).
- flankengesteuert: Wird es von allen Filterlisten erfasst oder nur exklusiv von einer einzigen?

Was passiert, wenn mehrere konkurrierende Filterlisten das gleiche Ereignis erfassen?

- Bei Exklusivität erreicht das Ereignis (im Idealfall) nur einen einzigen.
- Die Entscheidung, wer es erhält, fällt typischerweise beim Warten.
- Implementiert wird es dadurch, dass das Ereignis nach dem ersten Abruf sofort für alle anderen entfernt wird.

Exklusivität ist orthogonal zur Frage, ob es pegel- oder flankengesteuert ist.

Einige der Lösungen bieten eine Oneshot-Variante an.

- Bei dieser Option wird ein Filter aus der Filterliste entfernt, sobald ein entsprechendes Ereignis eingetreten ist.
- Für konkurrierende Situationen ist dies nur relevant, wenn eine Filterliste z.B. von konkurrierenden Threads gemeinsam genutzt wird.

Folgende Erweiterungen gibt es, die allesamt nicht zu POSIX gehören:

- */dev/poll* wurde von Sun für Solaris 7 bzw. 8 um 1999 eingeführt. Ist ausschließlich pegelgesteuert und beschränkt sich auf die Reduktion des Aufwands von $O(n)$ auf $O(1)$ im Vergleich zu *poll* oder *select*.
- *kqueue* wurde für FreeBSD entwickelt und stand ab FreeBSD 4.1 im Juli 2000 zur Verfügung. Das verbreitete sich auf andere BSD-Varianten und auf MacOS. Zahlreiche weitere Ereignisklassen wurden hinzugefügt. Ist überwiegend pegelgesteuert, Exklusivität und Oneshot werden unterstützt. Mehrere Threads können sich eine kqueue teilen, über *fork* kann es nicht vererbt werden.
- *epoll* kam zuerst mit dem Linux-Kernel 2.5.44 im Oktober 2002 und operiert per Voreinstellung pegelgesteuert, kann aber auch flankengesteuert arbeiten. Mehrere Threads oder Prozesse können sich eine Filterliste teilen, Oneshot ist dann möglich und sinnvoll. Exklusivität wird seit kurzem unterstützt.

- Was können optimierende Übersetzer erreichen?
- Wie lassen sich optimierende Übersetzer unterstützen?
- Welche Fallen können sich durch den Einsatz von optimierenden Übersetzer eröffnen?

Was wird optimiert?

Es gibt zwei teilweise gegensätzliche Ziele der Optimierung:

- Minimierung der Länge des erzeugten Maschinencodes.
- Minimierung der Ausführungszeit.

Es ist relativ leicht, sich dem ersten Ziel zu nähern. Die zweite Problemstellung ist in ihrer allgemeinen Form nicht vorbestimmbar (wegen potentiell unterschiedlicher Eingaben) bzw. in seiner allgemeinen Form nicht berechenbar.

- Die Problemstellung ist grundsätzlich für sehr kleine Sequenzen lösbar.
- Bei größeren Sequenzen wird zwar nicht das Minimum erreicht, dennoch sind die Ergebnisse beachtlich, wenn alle bekannten Techniken konsequent eingesetzt werden.

- Der GNU-Superoptimizer generiert sukzessive alle möglichen Instruktionssequenzen, bis eine gefunden wird, die die gewünschte Funktionalität umsetzt.
- Die Überprüfung erfolgt durch umfangreiche Tests, ist aber kein Beweis, dass die gefundene Sequenz äquivalent zur gewünschten Funktion ist. In der Praxis sind jedoch noch keine falschen Lösungen geliefert worden.
- Der Aufwand des GNU-Superoptimizers liegt bei $O((mn)^{2n})$, wobei m die Zahl der zur Verfügung stehenden Instruktionen ist und n die Länge der kürzesten Sequenz.
- Siehe `https://ftp.gnu.org/gnu/superopt/`
  (ist von 1995 und lässt sich leider mit modernen C-Übersetzern nicht mehr übersetzen).

- Problemstellung: Gegeben seien zwei nicht-negative ganze Zahlen in den Registern r_1 und r_2. Gewünscht ist das Minimum der beiden Zahlen in r_1.
- Eine naive Umsetzung erledigt dies analog zu einer **if**-Anweisung mit einem Vergleichstest und einem Sprung.
- Folgendes Beispiel zeigt dies für die SPARC-Architektur und den Registern %l0 und %l1:

```
        subcc   %l0,%l1,%g0
        bleu    endif
        nop
        or      %l1,%g0,%l0
endif:
```

```
        subcc   %l1,%l0,%g1
        subx    %g0,%g0,%g2
        and     %g2,%g1,%l1
        addcc   %l1,%l0,%l0
```

- Der Superoptimizer benötigt (auf Theon) nur eine Sekunde, um 28 Sequenzen mit jeweils 4 Instruktionen vorzuschlagen, die allesamt das Minimum bestimmen, ohne einen Sprung zu benötigen. Dies ist eine der gefundenen Varianten.
- Die Instruktionen entsprechen folgendem Pseudo-Code:

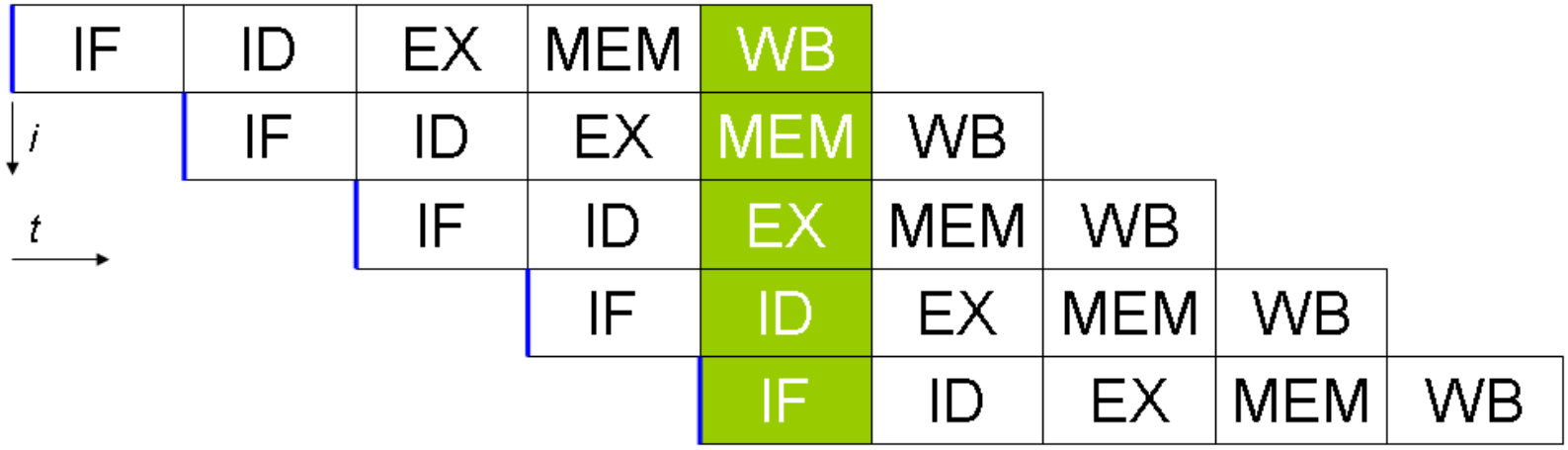
```
%g1 = %l1 - %l0
carry = %l0 > %l1? 1: 0
%g2 = -carry
%l1 = %g2 & %g1
%l0 = %l1 + %l0
```

- Generell ist die Vermeidung bedingter Sprünge ein Gewinn, da diese das Pipelining erschweren.

- Moderne Prozessoren arbeiten nach dem Fließbandprinzip: Über das Fließband kommen laufend neue Instruktionen hinzu und jede Instruktion wird nacheinander von verschiedenen Fließbandarbeitern bearbeitet.
- Dies parallelisiert die Ausführung, da unter günstigen Umständen alle Fließbandarbeiter gleichzeitig etwas tun können.
- Eine der ersten Pipelining-Architekturen war die IBM 7094 aus der Mitte der 60er-Jahre mit zwei Stationen am Fließband. Die UltraSPARC-IV-Architektur hat 14 Stationen, die Broadwell-Mikroarchitektur hat 16.
- Die RISC-Architekturen (RISC = *reduced instruction set computer*) wurden speziell entwickelt, um das Potential für Pipelining zu vergrößern.
- Bei der Pentium-Architektur werden im Rahmen des Pipelinings die Instruktionen zuerst intern in RISC-Instruktionen konvertiert, so dass sie ebenfalls von diesem Potential profitieren kann.

Um zu verstehen, was alles innerhalb einer Pipeline zu erledigen ist, hilft ein Blick auf die möglichen Typen von Instruktionen:

- Operationen, die nur auf Registern angewendet werden und die das Ergebnis in einem Register ablegen (wie etwa subcc in den Beispielen).
- Instruktionen mit Speicherzugriff. Hier wird eine Speicheradresse berechnet und dann erfolgt entweder eine Lese- oder eine Schreiboperation.
- Sprünge.



Eine einfache Aufteilung sieht folgende einzelne Schritte vor:

- Instruktion vom Speicher laden (IF)
- Instruktion dekodieren (ID)
- Instruktion ausführen, beispielsweise eine arithmetische Operation oder die Berechnung einer Speicheradresse (EX)
- Lese- oder Schreibzugriff auf den Speicher (MEM)
- Abspeichern des Ergebnisses in Registern (WB)

Das Diagramm wurde von Stephane Tsacas geklaut.

- Bedingte Sprünge sind ein Problem für das Pipelining, da unklar ist, wie gesprungen wird, bevor es zur Ausführungsphase kommt.
- RISC-Maschinen führen typischerweise die Instruktion unmittelbar nach einem bedingten Sprung immer mit aus, selbst wenn der Sprung genommen wird. Dies mildert etwas den negativen Effekt für die Pipeline.
- Im übrigen gibt es die Technik der *branch prediction*, bei der ein Ergebnis angenommen wird und dann das Fließband auf den Verdacht hin weiterarbeitet, dass die Vorhersage zutrifft. Im Falle eines Misserfolgs muss dann u.U. recht viel rückgängig gemacht werden.
- Das ist machbar, solange nur Register verändert werden. Manche Architekturen verfolgen die Alternativen sogar parallel und haben für jedes abstrakte Register mehrere implementierte Register, die die Werte für die einzelnen Fälle enthalten.
- Die Vorhersage wird vom Übersetzer generiert. Typisch ist beispielsweise, dass bei Schleifen eine Fortsetzung der Schleife vorhergesagt wird.

Es gibt zwei Möglichkeiten, um die Vorhersage bei Branch Prediction zu verbessern:

- Durch Profiling: Die Option „-fprofile-arcs“ beim *gcc* instrumentiert den Code, so dass Statistiken über den Verlauf zur Laufzeit generiert werden in Dateien, die in „.gcda“ enden. Diese Daten können mit der Option „-fbranch-probabilities“ bei einer späteren Neu-Übersetzung genutzt werden, um die Vorhersagen des Übersetzers zu verbessern.
- Durch explizite Vorgaben: Der *gcc* unterstützt die Builtin-Funktion *__builtin_expect*, die vom Übersetzer als Vorhersage übernommen wird. Beispiel:

```
ptr = head;
while (__builtin_expect(ptr != 0, 1)) {
    /* ... */
    ptr = ptr->next;
}
```

Normalerweise empfiehlt sich der erstere Ansatz.

- Bei moderneren Architekturen mit sehr langen Pipelines kommt ein statischer Hinweis viel zu spät, da hierzu die Instruktion bereits dekodiert sein muss.
- Neuere Architekturen verlassen sich daher ausschließlich auf ihre eigenen Vorhersagen auf der Basis früheren Verhaltens.
- Offenbar kommt beginnend mit der 2013 erschienenen Haswell-Architektur von Intel kommt das TAGE-Verfahren zum Einsatz, das 2006 von André Seznec und Pierre Michaud veröffentlicht worden ist: *A case for (partially) TAgged GEometric history length branch prediction*
- Dieses Verfahren zieht die vorangegangene Sprunghistorie und die Adresse der Sprung-Instruktion in Betracht und verwendet in der zugehörigen Hash-Tabelle partielle Tags, um im Falle von Kollisionen falsche Entscheidungen zu vermeiden.
- Statische Hinweise wie der entsprechende x86-Instruktionspräfix werden seit dem Aufkommen dynamischer Verfahren ignoriert.

Typische Techniken zur Minimierung des Maschinencodes

- Lokale Variablen und Parameter soweit wie möglich in Registern halten. Dies spart Lade- und Speicherinstruktionen.
- Vereinfachung von Blattfunktionen. Das sind Funktionen, die keine weitere Funktionen aufrufen.
- Auswerten von konstanten Ausdrücken während der Übersetzzeit (*constant folding*).
- Vermeidung von Sprüngen.
- Vermeidung von Multiplikationen, wenn einer der Operanden konstant ist.
- Elimination mehrfach vorkommender Teilausdrücke.
  Beispiel: `a[i + j] = 3 * a[i + j] + 1;`
- Konvertierung absoluter Adressberechnungen in relative.
  Beispiel: `for (int i = 0; i < 10; ++i) a[i] = 0;`
- Datenflussanalyse und Eliminierung unbenötigten Programmtexts

- Ein Übersetzer kann lokale Variablen nur dann permanent in einem Register unterbringen, wenn zu keinem Zeitpunkt eine Speicheradresse benötigt wird.
- Sobald der Adress-Operator & zum Einsatz kommt, muss diese Variable zwingend im Speicher gehalten werden.
- Das gleiche gilt, wenn Referenzen auf die Variable existieren.
- Zwar kann der Übersetzer den Wert dieser Variablen ggf. in einem Register vorhalten. Jedoch muss in verschiedenen Situationen der Wert neu geladen werden, z.B. wenn ein weiterer Funktionsaufruf erfolgt, bei dem ein Zugriff über den Zeiger bzw. die Referenz erfolgen könnte.

```
int index;
while (scanf("%d", &index) == 1) {
   a[index] = f(a[index]);
   a[index] += g(a[index]);
}
```

- In diesem Beispiel wird ein Zeiger auf *index* an die *scanf*-Funktion übergeben. Der Übersetzer weiß nicht, ob diese Adresse über irgendwelche Datenstrukturen so abgelegt wird, dass die Funktionen *f* und *g* darauf zugreifen. Entsprechend wird der Übersetzer genötigt, immer wieder den Wert von *index* aus dem Speicher zu laden.
- Deswegen ist es ggf. hilfreich, explizit eine weitere lokale Variablen zu verwenden, die eine Kopie des Werts erhält und von der keine Adresse genommen wird:

```
int index;
while (scanf("%d", &index) == 1) {
   int i = index;
   a[i] = f(a[i]);
   a[i] += g(a[i]);
}
```

```
bool find(int* a, int len, int* index) {
   while (*index < len) {
      if (a[*index] == 0) return true;
      ++*index;
   }
   return false;
}
```

- Parameter, die über einen Zeiger übergeben werden, sollten bei häufiger Nutzung in ausschließlich lokal genutzte Variablen kopiert werden, um einen externen Einfluss auszuschließen.

```
bool find(int* a, int len, int* index) {
   for (int i = *index; i < len; ++i) {
      if (a[i] == 0) {
         index = i; return true;
      }
   }
   *index = len;
   return false;
}
```

```
void f(int* i, int* j) {
   *i = 1;
   *j = 2;
   // value of *i?
}
```

- Wenn mehrere Zeiger oder Referenzen gleichzeitig verwendet werden, unterbleiben Optimierungen, wenn nicht ausgeschlossen werden kann, dass mehrere davon auf das gleiche Objekt zeigen.
- Wenn der Wert von `*i` verändert wird, dann ist unklar, ob sich auch `*j` verändert. Sollte anschließend auf `*j` zugegriffen werden, muss der Wert erneut geladen werden.
- In C gibt es die Möglichkeit, mit Hilfe des Schlüsselworts **restrict** Aliasse auszuschließen:

```
void f(int* restrict i, int* restrict j) {
   *i = 1;
   *j = 2;
   // *i == 1 still assumed
}
```

- Mit der Datenflussanalyse werden Abhängigkeitsgraphen erstellt, die feststellen, welche Variablen unter Umständen in Abhängigkeit welcher anderer Variablen verändert werden können.
- Im einfachsten Falle kann dies auch zur Propagation von Konstanten genutzt werden.
  Beispiel: Nach `int a = 7; int b = a;` ist bekannt, dass b den Wert 7 hat.
- Die Datenflussanalyse kann für eine Variable recht umfassend durchgeführt werden, wenn sie lokal ist und ihre Adresse nie weitergegeben wurde.

```
void loopingsleep(int count) {
   for (int i = 0; i < count; ++i)
      ;
}
```

- Mit Hilfe der Datenflussanalyse lässt sich untersuchen, welche Teile einer Funktion Einfluss haben auf den **return**-Wert oder die außerhalb der Funktion sichtbaren Datenstrukturen.
- Anweisungen, die nichts von außen sichtbares verändern, können eliminiert werden.
- Auf diese Weise verschwindet die **for**-Schleife im obigen Beispiel.
- Der erwünschte Verzögerungseffekt lässt sich retten, indem in der Schleife unter Verwendung der Schleifenvariablen eine externe Funktion aufgerufen wird. (Das funktioniert, weil normalerweise keine globale Datenflussanalyse stattfindet.)

- Globale Variablen können ebenso in die Datenflussanalyse einbezogen werden.
- Bei C wird davon ausgegangen, dass globale Variablen sich nicht überraschend ändern, solange keine Alias-Problematik vorliegt und keine unbekannten Funktionen aufgerufen werden.
- Das ist problematisch, wenn Threads oder Signalbehandler unsynchronisiert auf globale Variablen zugreifen.
- In diesem Fällen muss **volatile** verwendet werden.

Optimierende Übersetzer bieten typischerweise Stufen an. Recht typisch ist dabei folgende Aufteilung des gcc:

Stufe	Option	Vorteile
0		schnelle Übersetzung, mehr Transparenz beim Debugging
1	-O1	lokale Peephole-Optimierungen
2	-O2	Minimierung des Umfangs des generierten Codes
3	-O3	Minimierung der Laufzeit mit ggf. umfangreicheren Code
4	-Ofast	Abweichungen von Standard sind möglich – dies betrifft insbesondere mathematische Anwendungen, da u.a. die *order of evaluation* nicht eingehalten wird

Zusätzlich bietet sich noch das Profiling an und ggf. Optionen wie „-funroll-loops“.

Bei der (oder den) höchsten Optimierungsstufe(n) wird teilweise eine erhebliche Expansion des generierten Codes in Kauf genommen, um Laufzeitvorteile zu erreichen. Die wichtigsten Techniken:

- Loop unrolling
- Instruction scheduling
- Function inlining
- Vektorisierungen

```
for (int i = 0; i < 100; ++i) {
   a[i] = i;
}
```

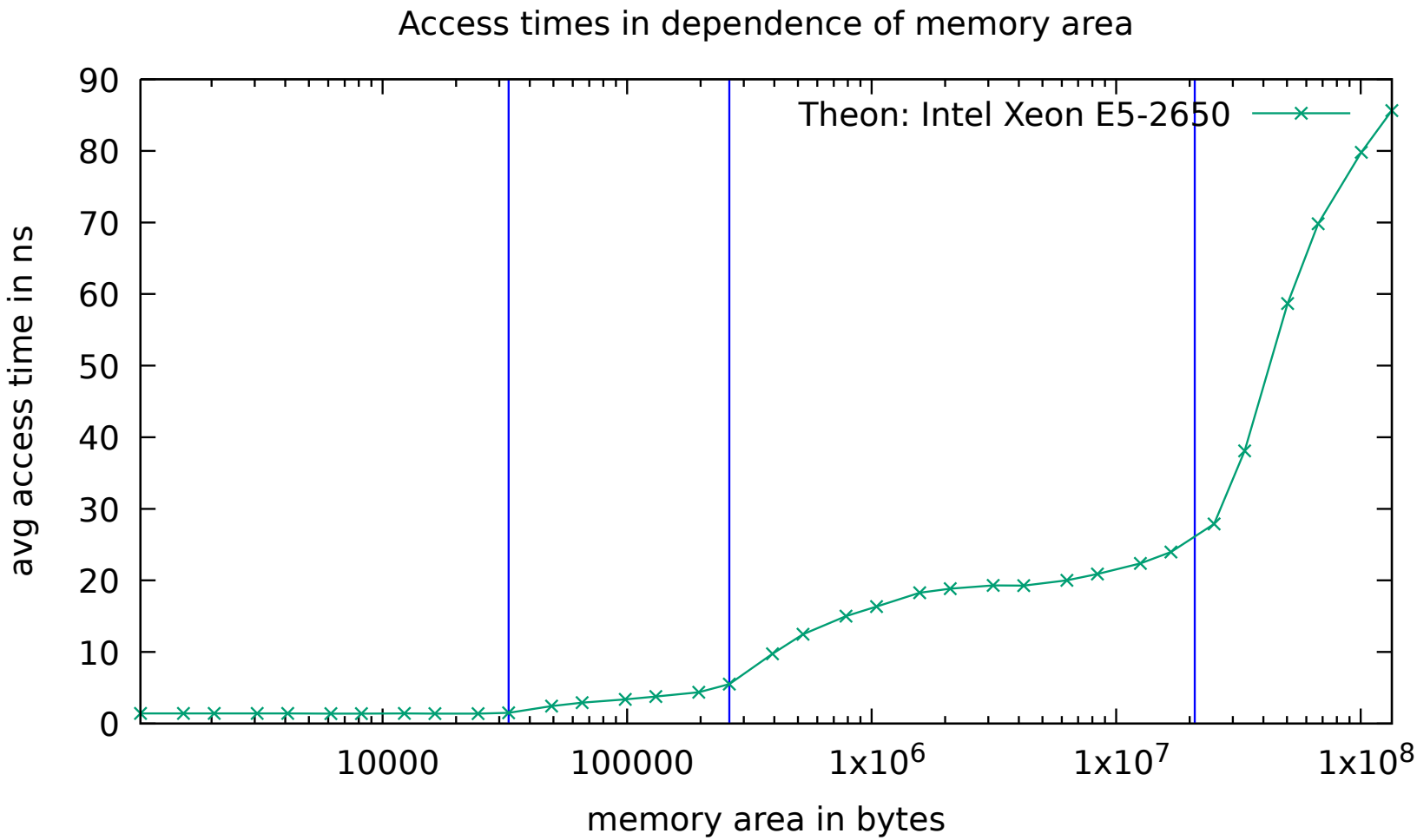
- Diese Technik reduziert deutlich die Zahl der bedingten Sprünge, indem mehrere Schleifendurchläufe in einem Zug erledigt werden.
- Das geht nur, wenn die einzelnen Schleifendurchläufe unabhängig voneinander erfolgen können, d.h. kein Schleifendurchlauf von den Ergebnissen der früheren Durchgänge abhängt.
- Dies wird mit Hilfe der Datenflussanalyse überprüft, wobei sich der Übersetzer auf die Fälle beschränkt, bei denen er sich sicher sein kann. D.h. nicht jede für diese Technik geeignete Schleife wird auch tatsächlich entsprechend optimiert.

```
for (int i = 0; i < 100; i += 4) {
   a[i] = i;
   a[i+1] = i + 1;
   a[i+2] = i + 2;
   a[i+3] = i + 3;
}
```

- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus Komplexitäts-, Produktions- und Kostengründen nicht beliebig ausbauen.
- Deswegen arbeiten moderne Architekturen mit einer Kette hintereinander geschalteter Speicher. Zur Einschätzung der Größenordnung sind hier die Angaben für die Theon, die mit einem E5-2650-v4-Prozessor der Broadwell Mikroarchitektur augestattet ist:

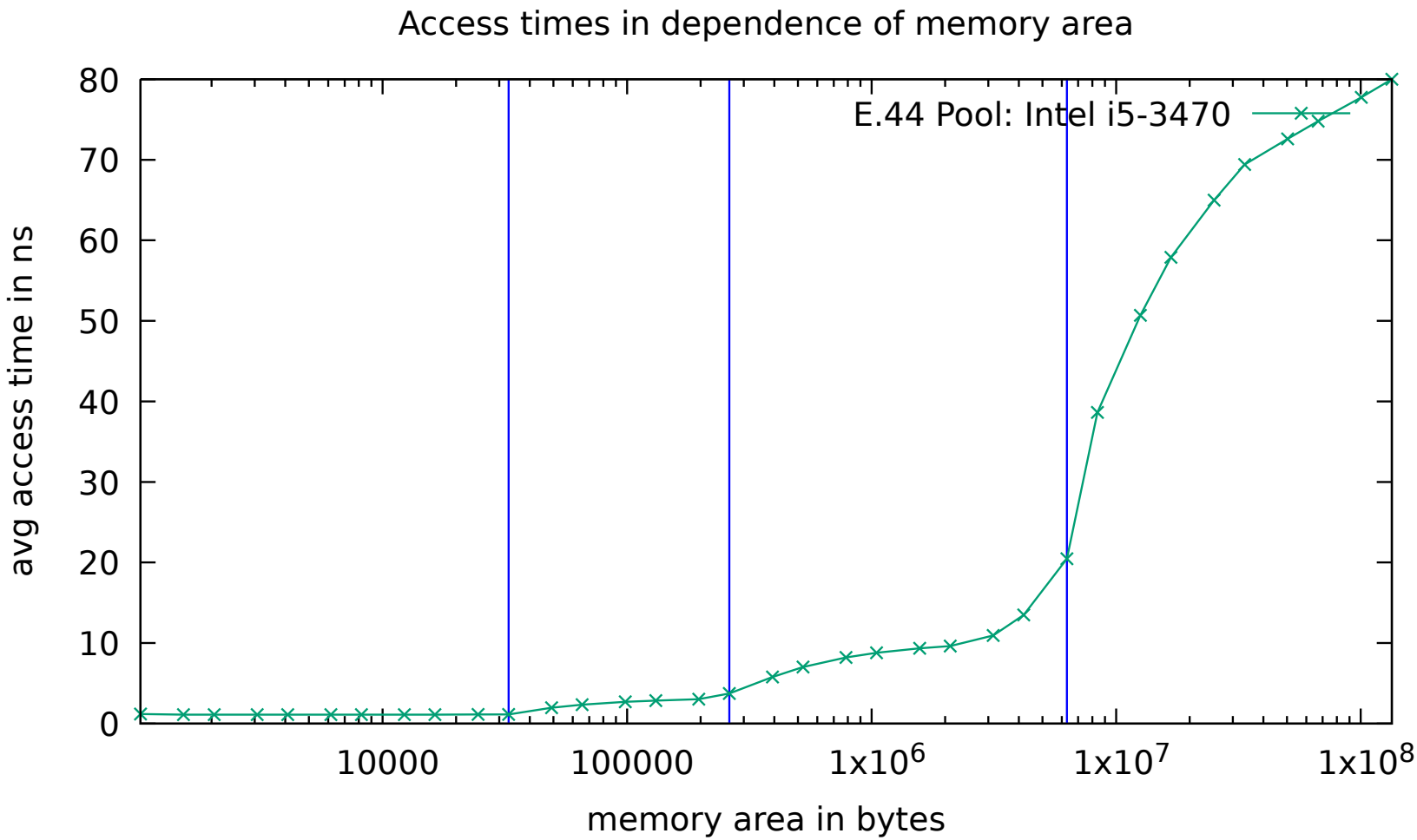
Cache	Kapazität	Taktzyklen
Register		1
L1-Cache	32 KiB	4-5
L2-Cache	256 KiB	12
L3-Cache	20 MiB	38
Hauptspeicher	96 GiB	38 + 58ns

Gemessene Speicherzugriffszeiten: Theon

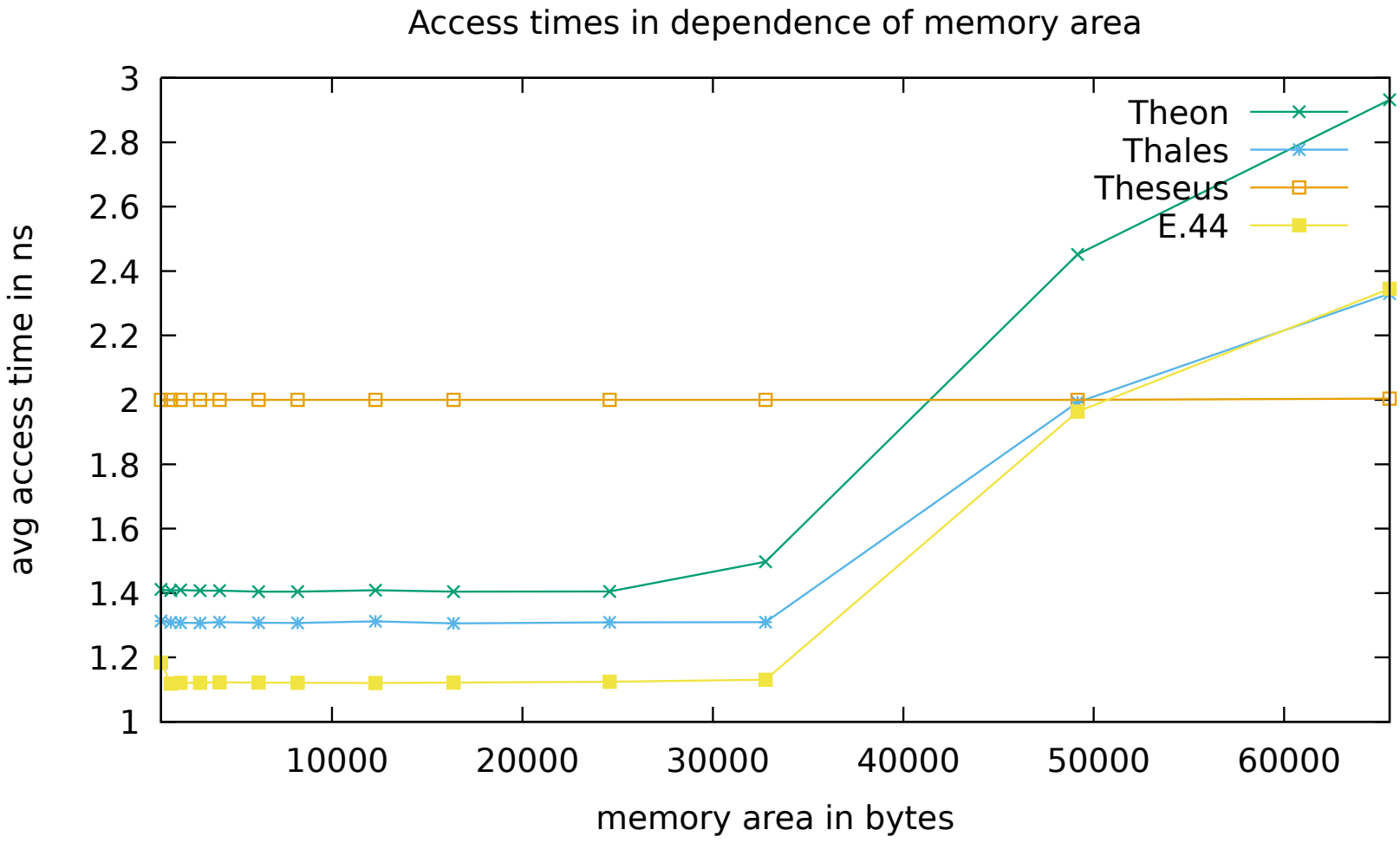


Theon: 1 Intel E5-2650-Prozessor mit 12 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (20 MiB)

Gemessene Speicherzugriffszeiten: E.44-Pool



E.44: 1 Intel i5-3470-Prozessor mit 4 Kernen
Caches: L1 (32 KiB), L2 (256 KiB), L3 (6 MiB)


Access times in dependence of memory area
avg access time in ns
memory area in bytes
Theon
Thales
Theseus
E.44
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
10000
20000
30000
40000
50000
60000

- Ein Cache ist in sogenannten *cache lines* organisiert, d.h. eine *cache line* ist die Einheit, die vom Hauptspeicher geladen oder zurückgeschrieben wird.
- Jede der *cache lines* umfasst – je nach Architektur – 32 - 128 Bytes. Auf der Theseus sind es beispielsweise 64 Bytes.
- Jede der *cache lines* kann unabhängig voneinander gefüllt werden und einem Abschnitt im Hauptspeicher entsprechen.
- Das bedeutet, dass bei einem Zugriff auf `a[i]` mit recht hoher Wahrscheinlichkeit auch `a[i+1]` zur Verfügung steht.
- Entweder sind Caches vollassoziativ (d.h. jede *cache line* kann einen beliebigen Hauptspeicherabschnitt aufnehmen) oder für jeden Hauptspeicherabschnitt gibt es nur eine *cache line*, die in Frage kommt (*fully mapped*), oder jeder Hauptspeicherabschnitt kann in einen von *n* *cache lines* untergebracht werden (*n-way set associative*).

- Es gibt keine portable Möglichkeit, die Cache-Konfiguration zu ermitteln.
- Unter Linux gibt es in der Hierarchie unter */sys/devices/system/cpu/cpu0/cache* für jeden der Caches der *cpu0* ein Verzeichnis *index0*, *index1* usw.
- In jedem Cache-Verzeichnis finden sich folgende Dateien:

level	Level des Cache, typischerweise 1, 2 oder 3
type	„Data“, „Instruction“ oder „Unified“
coherency_line_size	Größe einer *cache line*
ways_of_associativity	wieviele *cache lines* kommen in Frage, um einen Abschnitt unterzubringen? Alle *cache lines*, die auf diese Weise zusammengehören, werden einem Set zugeordnet.
number_of_sets	Zahl der Sets (s.o.)
size	Produkt aus *coherency_line_size*, *ways_of_associativity* und *number_of_sets*.

Ziel ist es, die zur Verfügung stehenden CPUs auszulasten, d.h. es sollte möglichst wenig Zeit damit verbracht werden, auf Speicherzugriffe zu warten. Dazu gibt es folgende Ansätze:

- Cache-optimierte Datenstrukturen mit möglichst wenig Indirektionen durch Zeiger. Eine Matrix sollte beispielsweise zusammenhängend im Speicher abgelegt werden und nicht mit Hilfe einer Zeigerliste (also beispielsweise mit **double**$**$) realisiert werden.
- Entsprechend kann es sich lohnen, lineare Listen blockweise zu implementieren oder B-Bäume bzw. B*-Bäume statt binären ausgeglichenen Bäumen zu verwenden.
- Einsatz fortgeschrittener Optimierungstechniken wie *instruction schedulung*, ggf. in Verbindung mit *loop unrolling*.

- Diese Technik bemüht sich darum, die Instruktionen (soweit dies entsprechend der Datenflussanalyse möglich ist) so anzuordnen, dass in der Prozessor-Pipeline keine Stockungen auftreten.
- Das lässt sich nur in Abhängigkeit des konkret verwendeten Prozessors optimieren, da nicht selten verschiedene Prozessoren der gleichen Architektur mit unterschiedlichen Pipelines arbeiten.
- Ein recht großer Gewinn wird erzielt, wenn ein vom Speicher geladener Wert erst sehr viel später genutzt wird.
- Beispiel: `x = a[i] + 5; y = b[i] + 3;`
  Hier ist es sinnvoll, zuerst die Ladebefehle für `a[i]` und `b[i]` zu generieren und erst danach die beiden Additionen durchzuführen und am Ende die beiden Zuweisungen.

axpy.c

```
// y = y + alpha * x
void axpy(int n, double alpha, const double* x, double* y) {
   for (int i = 0; i < n; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Dies ist eine kleine Blattfunktion, die eine Vektoraddition umsetzt. Die Länge der beiden Vektoren ist durch n gegeben, x und y zeigen auf die beiden Vektoren.
- Aufrufkonvention:

Variable	Register
n	%o0
alpha	%o1 und %o2
x	%o3
y	%o4

Resultat von gcc -O3

axpy.s

```
        add     %sp, -120, %sp
        cmp     %o0, 0
        st      %o1, [%sp+96]
        st      %o2, [%sp+100]
        ble     .LL5
        ldd     [%sp+96], %f12
        mov     0, %g2
        mov     0, %g1
.LL4:
        ldd     [%g1+%o3], %f10
        ldd     [%g1+%o4], %f8
        add     %g2, 1, %g2
        fmuld   %f12, %f10, %f10
        cmp     %o0, %g2
        faddd   %f8, %f10, %f8
        std     %f8, [%g1+%o4]
        bne     .LL4
        add     %g1, 8, %g1
.LL5:
        jmp     %o7+8
        sub     %sp, -120, %sp
```

- Ein *loop unrolling* fand hier nicht statt, wohl aber ein *instruction scheduling*.

- Der C-Compiler von Sun generiert für die gleiche Funktion 241 Instruktionen (im Vergleich zu den 19 Instruktionen beim gcc).
- Der innere Schleifenkern mit 81 Instruktionen behandelt 8 Iterationen gleichzeitig. Das orientiert sich exakt an der Größe der *cache lines* der Architektur: $8 * \textbf{sizeof}(\textbf{double}) == 64$.
- Mit Hilfe der prefetch-Instruktion wird dabei jeweils noch zusätzlich dem Cache der Hinweis gegeben, die jeweils nächsten 8 Werte bei x und y zu laden.
- Der Code ist deswegen so umfangreich, weil
  - die Randfälle berücksichtigt werden müssen, wenn n nicht durch 8 teilbar ist und
  - die Vorbereitung recht umfangreich ist, da der Schleifenkern von zahlreichen bereits geladenen Registern ausgeht.

- Der gcc kann mit der Option „-funroll-loops“ ebenfalls dazu überredet werden, Schleifen zu expandieren.
- Bei diesem Beispiel werden dann ebenfalls 8 Iterationen gleichzeitig behandelt.
- Der innere Schleifenkern besteht beim gcc nur aus 51 Instruktionen – ein *prefetch* entfällt und das Laden aus dem Speicher wird nicht an den Schleifenanfang vorgezogen. Entsprechend wird hier das Optimierungspotential noch nicht ausgereizt.

- Hierbei wird auf den Aufruf einer Funktion verzichtet. Stattdessen wird der Inhalt der Funktion genau dort expandiert, wo sie aufgerufen wird.
- Das läuft so ähnlich ab wie bei der Verwendung von Makros, nur gelten weiterhin die bekannten Regeln.
- Das kann jedoch nur gelingen, wenn der Programmtext der aufzurufenden Funktion bekannt ist.
- Das klappt bei Funktionen, die in der gleichen Übersetzungseinheit enthalten sind und in C++ bei Templates, bei denen der benötigte Programmtext in der entsprechenden Headerdatei zur Verfügung steht.
- Letzteres wird in C++ intensiv (etwa bei der STL oder der *iostreams*-Bibliothek) genutzt, was teilweise erhebliche Übersetzungszeiten mit sich bringt.