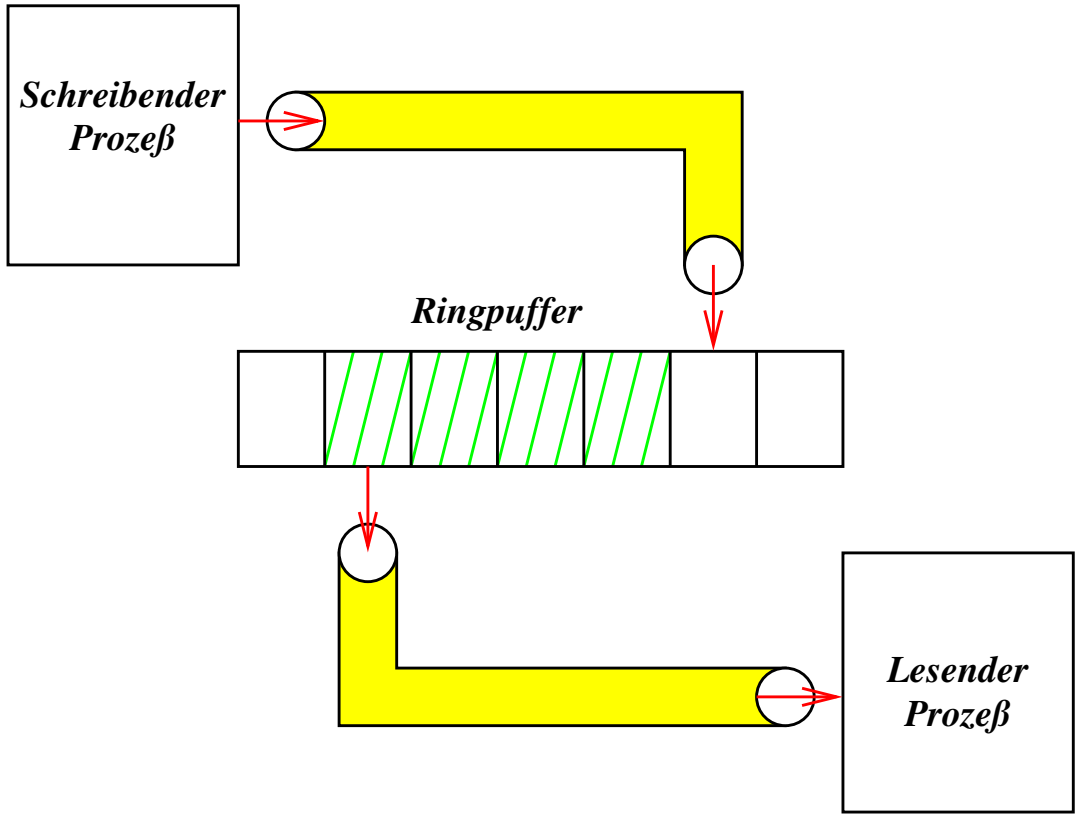
```
theon$ ypcat passwd | iconv -f latin1 | cut -d: -f5 |
> sed 's/ .*//' | sort | uniq -c | sort -rn | head
  54 Michael
  48 Daniel
  46 Alexander
  44 Tobias
  37 Florian
  35 Christian
  33 Matthias
  32 Johannes
  30 Markus
  30 Lukas
theon$
```

- 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.

Schnittstelle für Pipelines zu Kommandos

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? 167

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