Kapitel 5: Pipes und FIFOs

5: Pipes und FIFOs

- Neben Signalen gibt es noch weiter Techniken zur Interprozesskommunikation.
- In diesem Kapitel werden Pipes und FIFOs (named Pipes) behandelt.
- Das Konzept der Pipe sollte bereits aus Systemprogrammierung bekannt sein (`cmd1 | cmd 2`)
- Pipes werden von Elternprozessen eingerichtet.
- Pipes können nur zwischen verwandten Prozessen (Eltern-Kind, Kind-Kind) eingerichtet werden.
- Pipes sind halbduplex, d.h. Daten können nur in eine Richtung fließen.
- FIFOs sind vollduplex und können von beliebigen Prozessen benutzt werden.

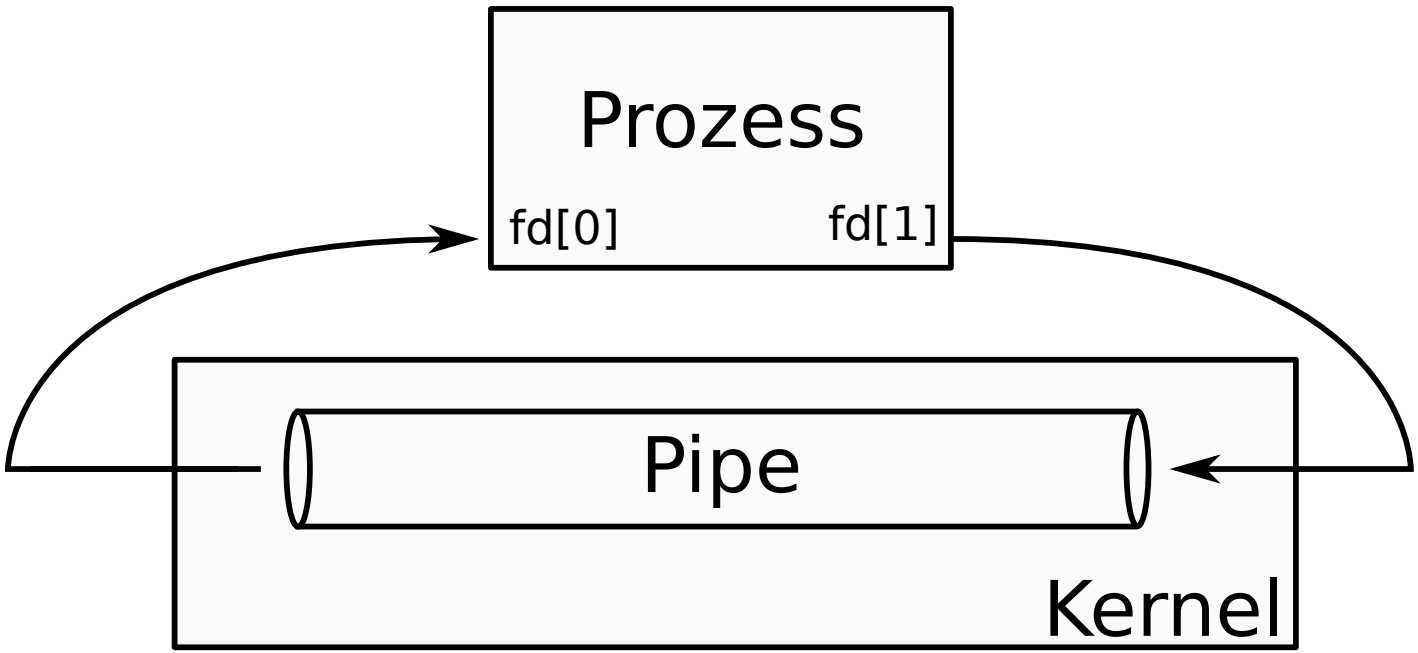
5.1: Pipes

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

- Bei Erfolg wird `0` zurückgegeben, ansonsten `-1`.
- `pipe()` erstellt einen unidirektionalen Datenkanal.
  - `pipefd[0]`: Lese-Ende (Ausgang)
  - `pipefd[1]`: Schreib-Ende (Eingang)
- Da eine Pipe aus zwei Dateideskriptoren besteht, können Dateioperationen wie **`read()`** oder **`write()`** darauf angewandt werden.
- Durch Verwendung von **`fdopen()`** können auch `stdio.h` Funktionen wie `fprintf()` eingesetzt werden.

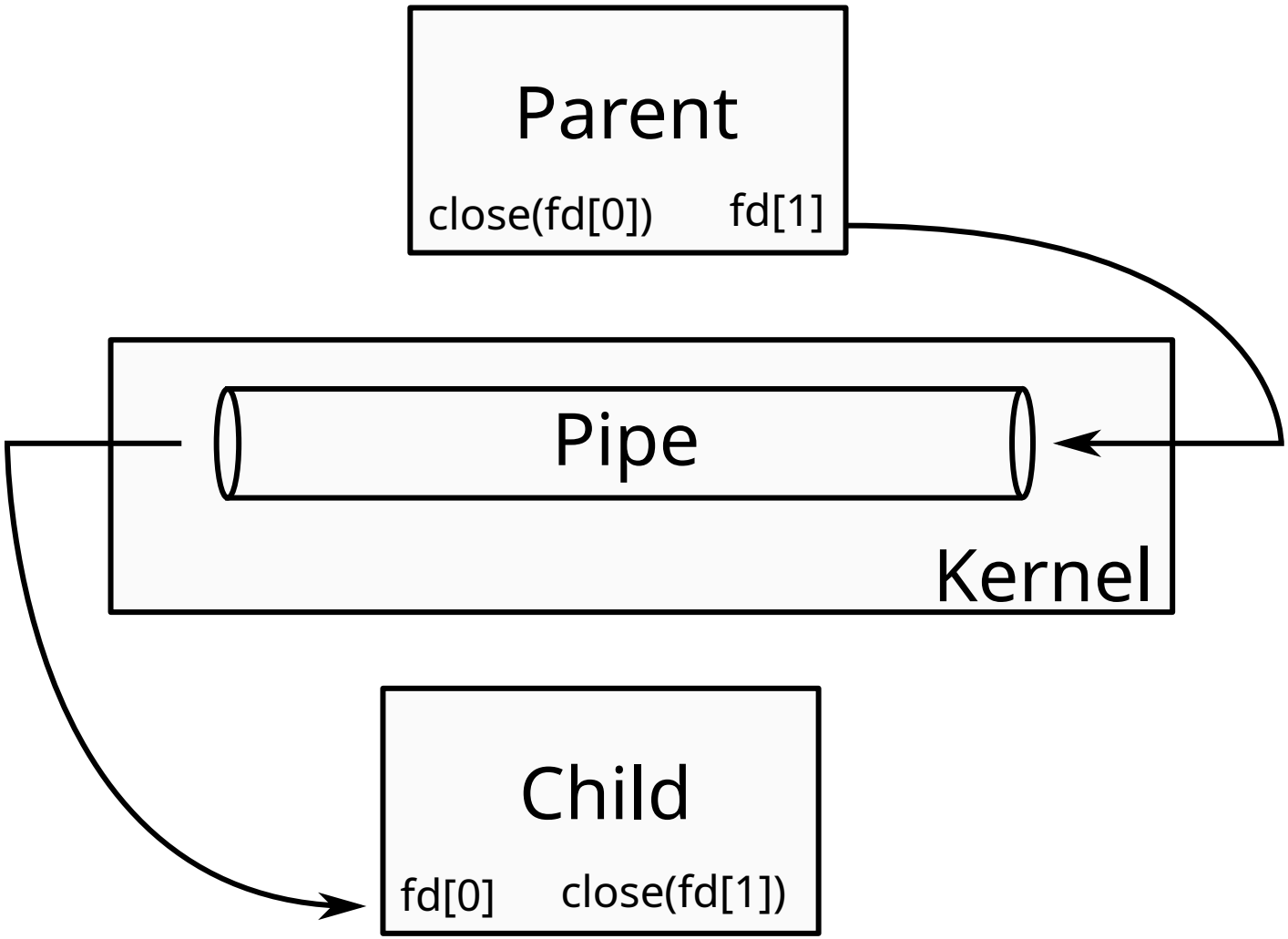
Illustration von pipe()



Anwendung einer Pipe

- Eine Pipe innerhalb eines Prozesses ist nur bedingt sinnvoll.
- Damit zwei Prozesse über eine Pipe Kommunizieren können, sollte nach einem `pipe()`-Aufruf ein `fork()`-Aufruf erfolgen.
- Es ist zwar möglich aber unüblich, dass beide Prozesse (Eltern und Kind) von der Pipe lesen bzw. schreiben.
- Es ist üblich, dass direkt nach dem `fork()`-Aufruf der eine Prozess das Schreib-Ende und der andere das Lese-Ende schließt.
- Wenn mehrere Prozesse von einer Pipe-Lesen ist unklar welcher Prozess erfolgreich ist (Race Condition).
- Für bidirektionelle Kommunikation sollten immer zwei Pipes eingerichtet werden.

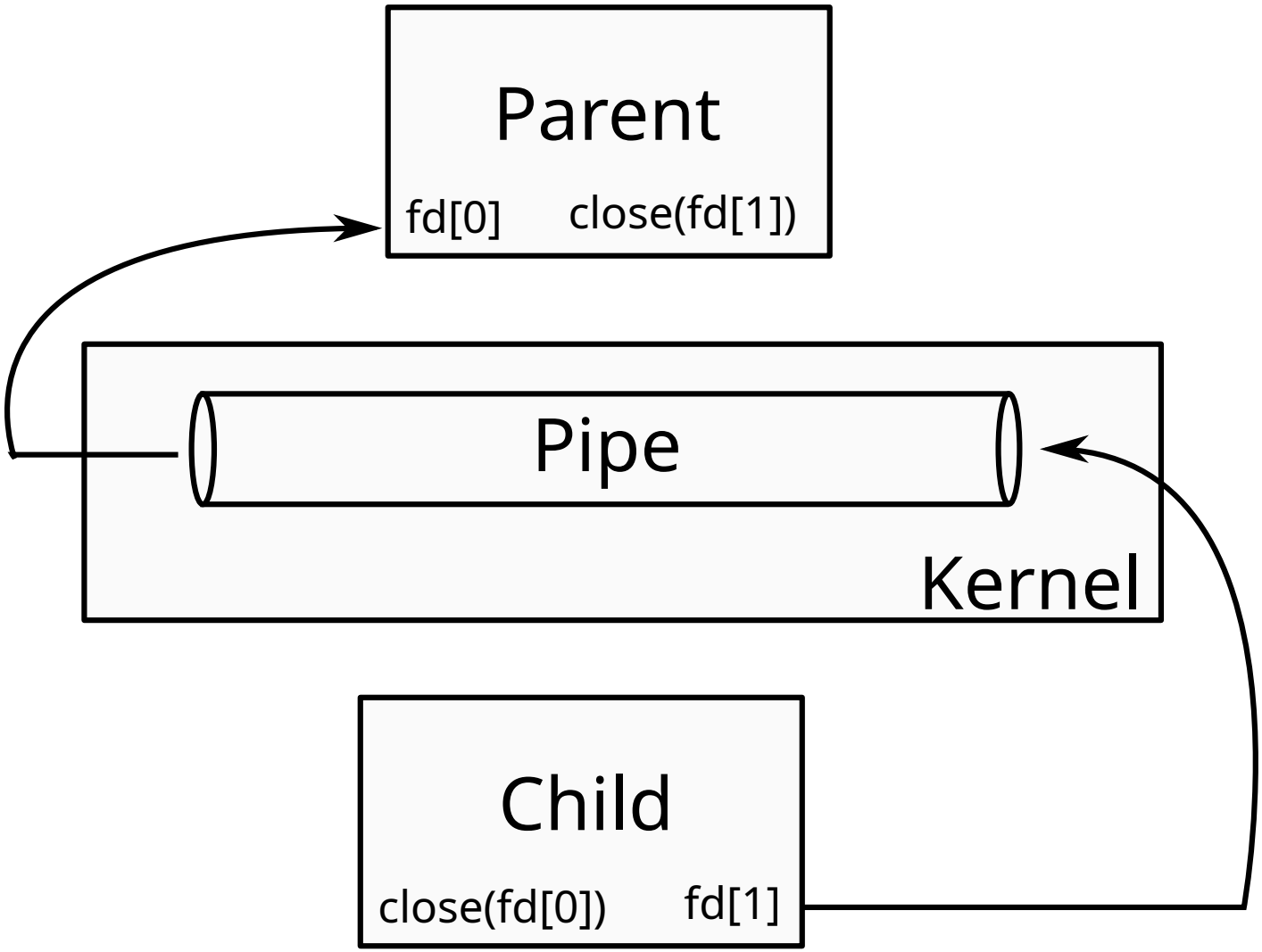
Elternprozess schreibt und Kindprozess liest



Beispiel: Elternprozess schreibt

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <ctype.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <sys/wait.h>
7  #define BUF_LEN 120
8
9  void child(int pfd[2]) {
10   close(pfd[1]);
11   char buf[BUF_LEN];
12   memset(buf,0,BUF_LEN);
13
14   int n = read(pfd[0], buf, BUF_LEN-1);
15   for(int i=0; i < n; i++) buf[i] = toupper(buf[i]);
16   puts(buf);
17   close(pfd[0]);
18 }
19 int main(int argc, char *argv[]) {
20   if(argc !=2) return EXIT_FAILURE;
21   int pfd[2];
22   pipe(pfd);
23   switch(fork()) {
24   case  0: child(pfd); break;
25   case -1: perror("fork"); exit(EXIT_FAILURE);
26   default:
27     close(pfd[0]);
28     write(pfd[1], argv[1], strlen(argv[1]));
29     close(pfd[1]);
30     wait(NULL);
31   }
32 }
```

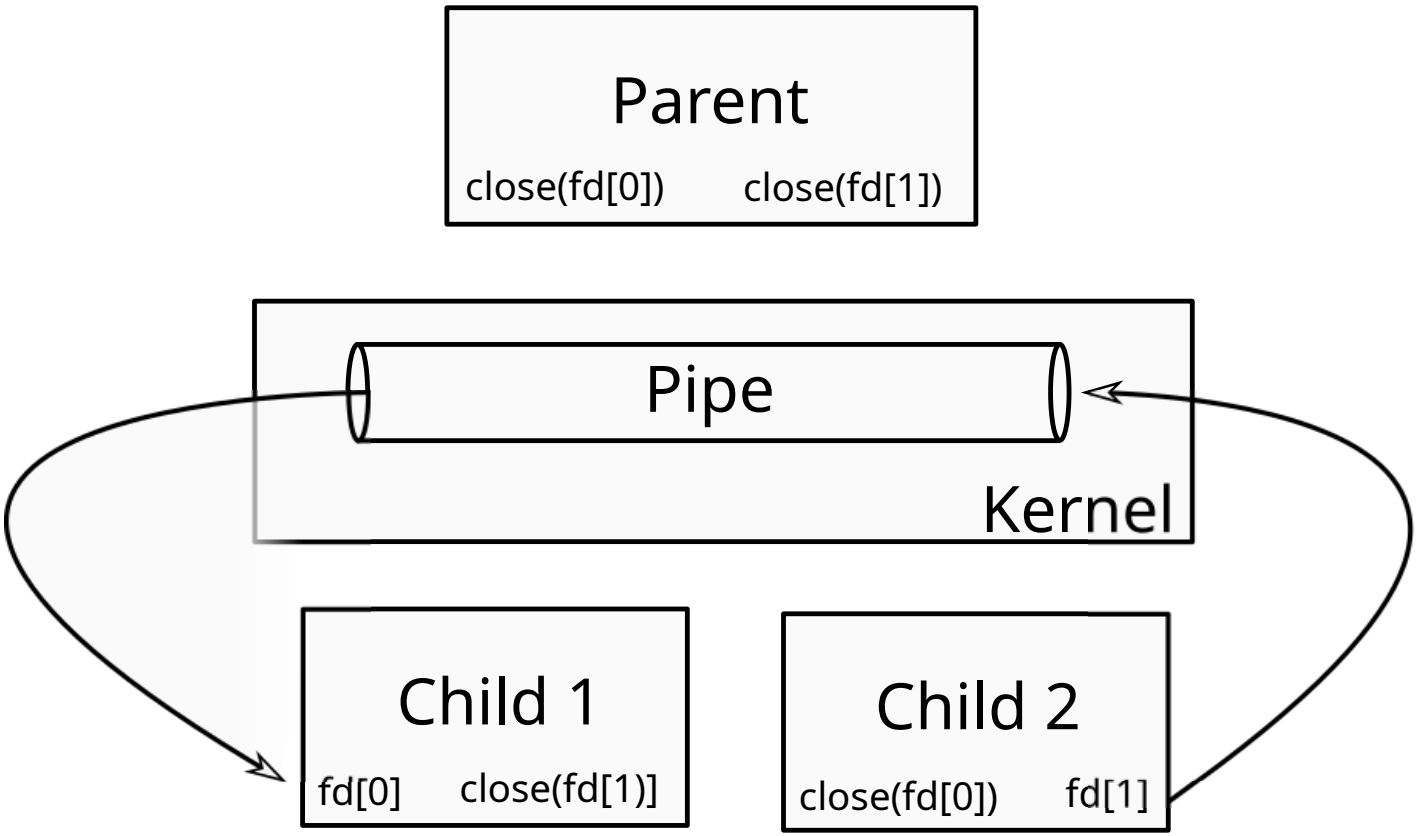
Kindprozess schreibt und Elternprozess liest



Beispiel: Kindprozess schreibt

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <ctype.h>
4  #include <stdio.h>
5  #include <string.h>
6  #define BUF_LEN 120
7
8  void parent(int pfd[2]) {
9    close(pfd[1]);
10   char buf[BUF_LEN];
11   memset(buf,0,BUF_LEN);
12
13   int n = read(pfd[0], buf, BUF_LEN-1);
14   for(int i=0; i < n; i++) buf[i] = toupper(buf[i]);
15   puts(buf);
16   close(pfd[0]);
17 }
18
19 void child(int pfd[2], char *msg) {
20   close(pfd[0]);
21   write(pfd[1], msg, strlen(msg));
22   close(pfd[1]);
23 }
24
25 int main(int argc, char *argv[]) {
26   if(argc !=2) return EXIT_FAILURE;
27   int pfd[2];
28   pipe(pfd);
29   if(fork() == 0) child(pfd, argv[1]);
30   else parent(pfd);
31 }
```

Kindprozess schreibt und Kindprozess liest



Beispiel: Kinder lesen und schreiben

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <ctype.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <sys/wait.h>
7  #define BUF_LEN 120
8
9  void cread(int pfd[2]) {
10   close(pfd[1]);
11   char buf[BUF_LEN];
12   memset(buf,0,BUF_LEN);
13   int n = read(pfd[0], buf, BUF_LEN-1);
14   for(int i=0; i < n; i++) buf[i] = toupper(buf[i]);
15   puts(buf);
16   close(pfd[0]);
17 }
18
19 void cwrite(int pfd[2], const char *msg) {
20   close(pfd[0]);
21   write(pfd[1], msg, strlen(msg));
22   close(pfd[1]);
23 }
24
25 int main(int argc, char *argv[]) {
26   if(argc !=2) return EXIT_FAILURE;
27   int pfd[2]; pipe(pfd);
28   if(fork() == 0) { cread(pfd); return EXIT_SUCCESS; }
29   if(fork() == 0) { cwrite(pfd,argv[1]); return EXIT_SUCCESS; }
30   close(pfd[0]); close(pfd[1]);
31   wait(NULL); wait(NULL);
32 }
```

Beispiel: Pipes mit fdopen

```
1  #include <stdlib.h>
2  #include <ctype.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <sys/wait.h>
6  #include <unistd.h>
7  void cread(int pfd[2]) {
8    close(pfd[1]);
9    size_t len;
10   char *line = NULL;
11   FILE *fp = fdopen(pfd[0], "r");
12   getline(&line, &len, fp);
13   for(size_t i=0; i < len; i++) line[i] = toupper(line[i]);
14   puts(line);
15   fclose(fp);
16   free(line);
17 }
18
19 void cwrite(int pfd[2], const char *msg) {
20   close(pfd[0]);
21   FILE *fp = fdopen(pfd[1], "w");
22   fputs(msg,fp);
23   fclose(fp);
24 }
25
26 int main(int argc, char *argv[]) {
27   if(argc !=2) return EXIT_FAILURE;
28   int pfd[2]; pipe(pfd);
29   if(fork() == 0) { cread(pfd); return EXIT_SUCCESS; }
30   if(fork() == 0) { cwrite(pfd,argv[1]); return EXIT_SUCCESS; }
31   close(pfd[0]); close(pfd[1]);
32   wait(NULL); wait(NULL);
```

5.2: Named Pipes (FIFOs)

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

- `pathname`: Name der Pipe, welche im Dateisystem angelegt werden soll.
- `mode`: Zugriffsrechte (siehe `open()`).
- Bei Erfolg wird `0` ansonsten `-1` zurück gegeben.
- Auf Named Pipes können elementare I/O-Funktionen wie `open()`, `read()`, `write()` oder `unlink()` angewandt werden.

Regeln für FIFO-Zugriffe

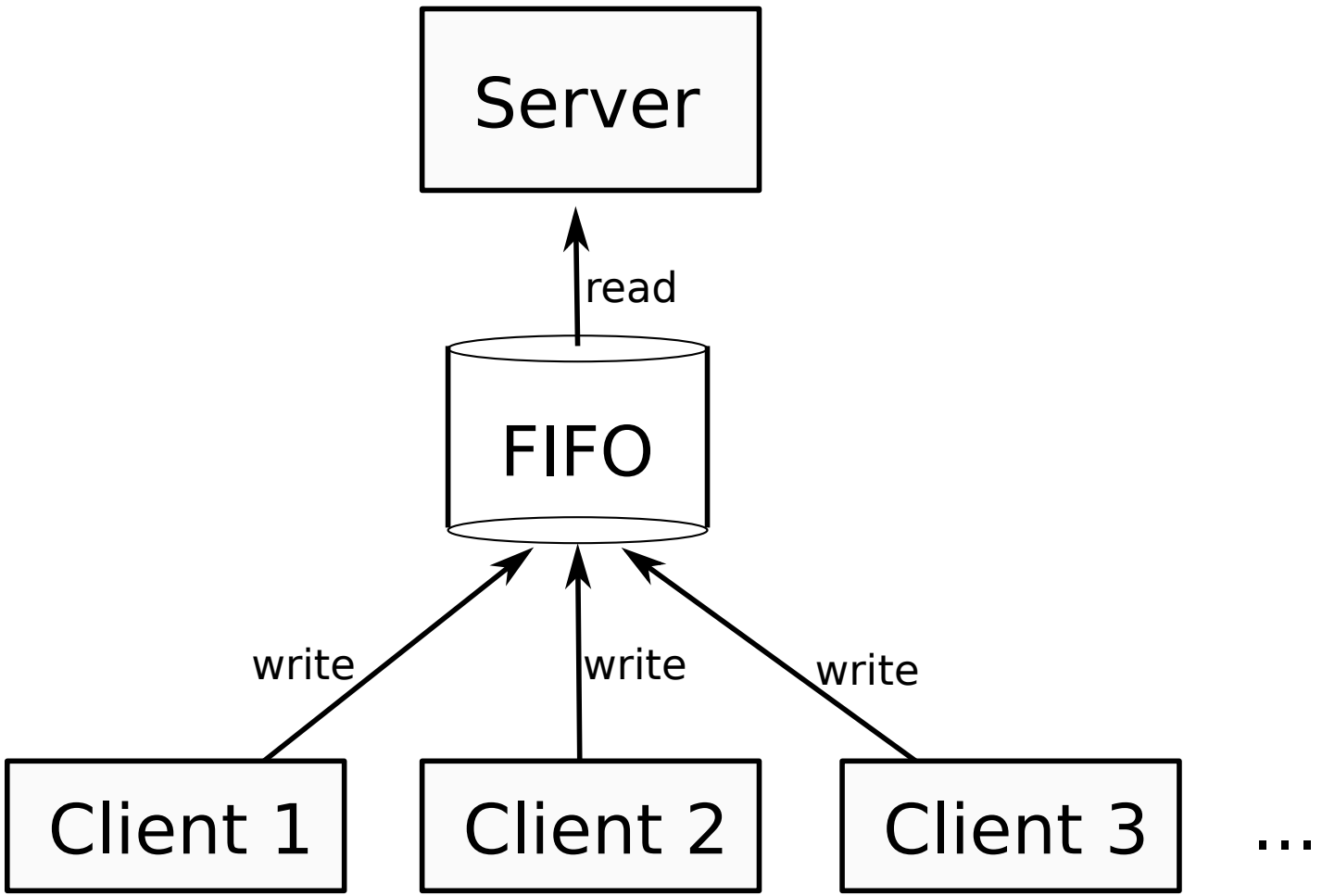
1. Bei `open()` **ohne** Flag `O_NONBLOCK` werden Lese- und Schreibzugriffe blockiert.
2. Bei `open()` **mit** Flag `O_NONBLOCK` werden Lesezugriffe nicht blockiert.
3. Beim Schreiben in eine Pipe ohne Leser wird das Signal `SIGPIPE` zugestellt.
4. Schließt der letzte Schreiber eine FIFO wird beim lesen EOF zurück geliefert.
5. Daten welche nicht länger als `PIPE_BUF` (`limits.h`) Bytes sind werden atomar geschrieben.

Einsatz von Named Pipes

- Ersatz für temporäre Dateien
  - Der Inhalt von Dateien wird auf Platte geschrieben.
  - Der Inhalt von FIFOs wird in den Speicherbereich des Kernels (Arbeitsspeicher) geschrieben.
  - FIFOs sind daher effizienter als temporäre Dateien.

- Datenaustausch zwischen lokalen Client und Server.
  - Server legt ein FIFO an, deren Namen allen Clients bekannt ist.
  - Server öffnet die Named Pipe zum lesen.
  - Die Clients öffnen die Named Pipe zum schreiben, um Nachrichten an den Server zu senden.
  - Beispielanwendung: Drucker-Spooler.

Client-Anfragen



Beispiel: FIFO-Server

```
1  #include <sys/stat.h>
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <limits.h>
6
7  #define FIFO "/tmp/foo.fifo"
8
9  int main() {
10   mkfifo(FIFO, S_IRUSR | S_IWUSR);
11
12   int fd = open(FIFO, O_RDONLY);
13
14   while(1) {
15     char buf[PIPE_BUF+1];
16     int n = read(fd, buf, PIPE_BUF);
17
18     if(n<0) break;
19     if(n>0) write(1,buf,n);
20     if(n==0) sleep(0.1);
21   }
22   close(fd);
23   unlink(FIFO);
24 }
```

Beispiel: FIFO-Clients

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5  #include <sys/wait.h>
6
7  #define FIFO "/tmp/foo.fifo"
8  #define CHILD 0
9
10 int main() {
11   FILE *fifo = fopen(FIFO, "w");
12
13   for(int i=0; i < 3; i++) {
14     if ( fork() == CHILD ) {
15       fprintf(fifo,"Testnachricht␣#%d\n", i);
16       fflush(fifo);
17       exit(EXIT_SUCCESS);
18     }
19   }
20   for(int i=0; i < 3; i++) wait(NULL);
21   fclose(fifo);
22 }
```

Zusammenfassung

Sie sollten . . .

- . . . das Konzept der Pipes verstanden haben.
- . . . wissen was eine *Named Pipe* ist.
- . . . den Umgang mit `popen()` beherrschen.