

Interprozesskommunikation

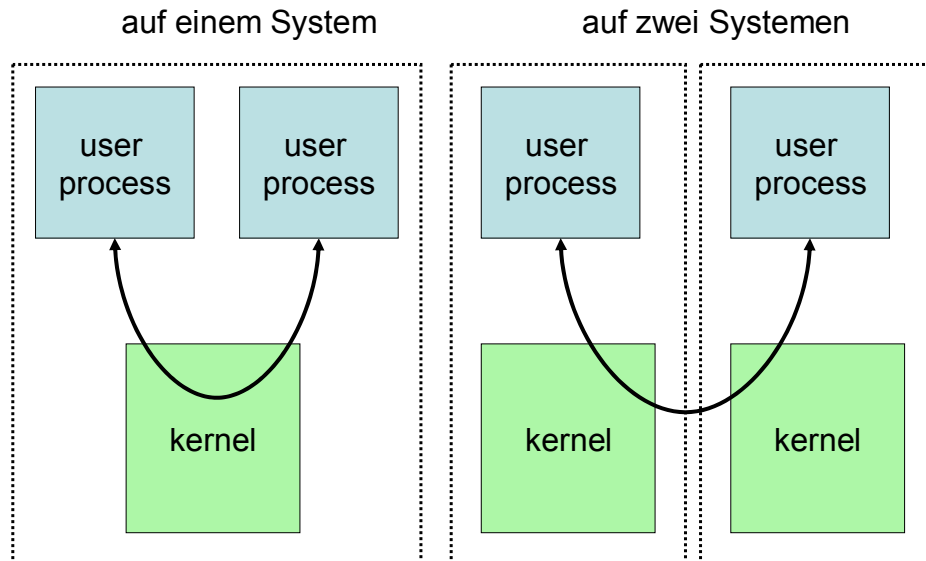
Inhalt

1. Überblick.....	2
2. Pipes	5
3. Fifo.....	22
3.1. Übung IPC-2.....	39

1. Überblick

Hier werden die unterschiedlichen Methoden der Kommunikation zwischen Prozessen betrachtet. Die Interprozesskommunikation (*Interprocess Communication, IPC*) spielt eine Rolle auf Rechner- und Netzebene.

IPC zwischen zwei Prozessen



Die wichtigsten Methoden der IPC auf einem Rechner sind:

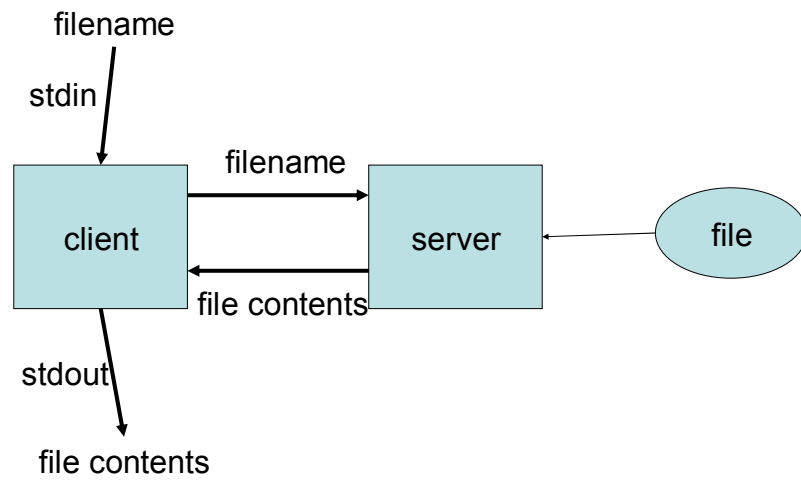
- Pipes,
- FIFOs (**named pipes**),
- Nachrichten Warteschlangen (*message queues*),

- Semaphore (*semaphores*) und
- gemeinsamer Speicher (*shared memory*)

Hier werden Pipes und Fifos behandelt. Die restlichen Methoden und Methoden der IPC in Netzen sind Gegenstand einer eigenen Veranstaltung.

An einem einfachen Beispiel (CS cat) werden die einzelnen Methoden verdeutlicht: eine Client-Server Anwendung, bei der ein Client einen Dateinamen von stdin einliest und den Namen auf einen Kommunikationskanal schreibt. Der Server liest den Dateinamen vom Kommunikationskanal, öffnet die Datei, liest den Inhalt und schreibt ihn über einen Kommunikationskanal zurück. Der Client erwartet den Inhalt der Datei vom Kommunikationskanal und gibt ihn über stdout aus.

Client-Server Beispiel



2. Pipes

Eine **Pipe** stellt eine **Einwegverbindung** zwischen Sender und Empfänger dar.

Eine Pipe wird erzeugt durch:

```
int pipe(int *fildes);
```

In dem Zeiger *fildes* werde zwei Dateideskriptoren zurück geliefert:

- `fildes[0]` zum Lesen
- `fildes[1]` zum Schreiben

Ein negativer Wert als Funktionsergebnis bedeutet, dass keine Pipe erzeugt werden konnte.

Pipes sind eigentlich nur sinnvoll, wenn zwei Prozesse beteiligt sind. Das folgende Beispiel benutzt eine Pipe innerhalb eines Prozesses.

```
main()
{
    int pipefd[2], n;
    char buf[100];

    if (pipe(pipefd) < 0)
        printf("pipe error\n");

    printf("read fd=%d, write fd=%d\n", pipefd[0], pipefd[1]);

    if(write(pipefd[1], "Hello, world\n", 12) != 12)
        printf("write error\n");

    if((n=read(pipefd[0],buf, sizeof(buf))) <= 0)
        printf("read error\n");

    write(1, buf, n); /* fd 1=stdout */
}
```

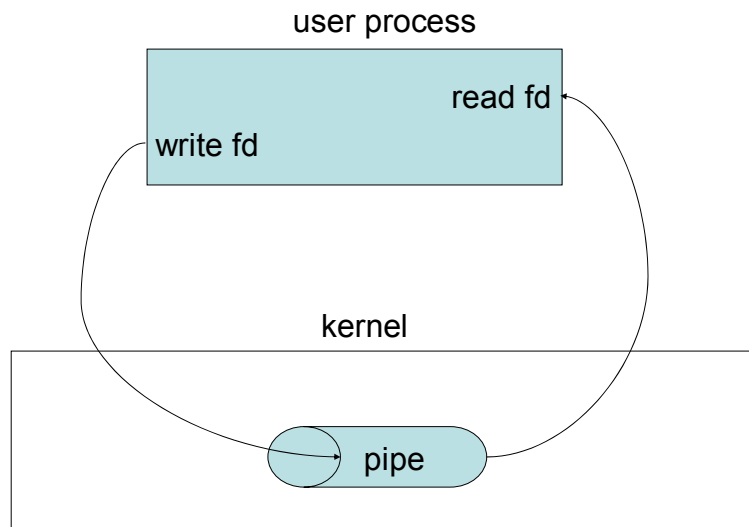
Die Ausgabe ist:

```
hello, world
read fd=3, write=4
```

Achtung: Diese Reihenfolge ist begründet in der Tatsache, dass die Ausgabe von printf gepuffert ist, die von write nicht.

Bei Verwendung mit nur einem Prozess, wie im letzten Beispiel, wird eine Pipe innerhalb des Betriebssystems wie folgt dargestellt:

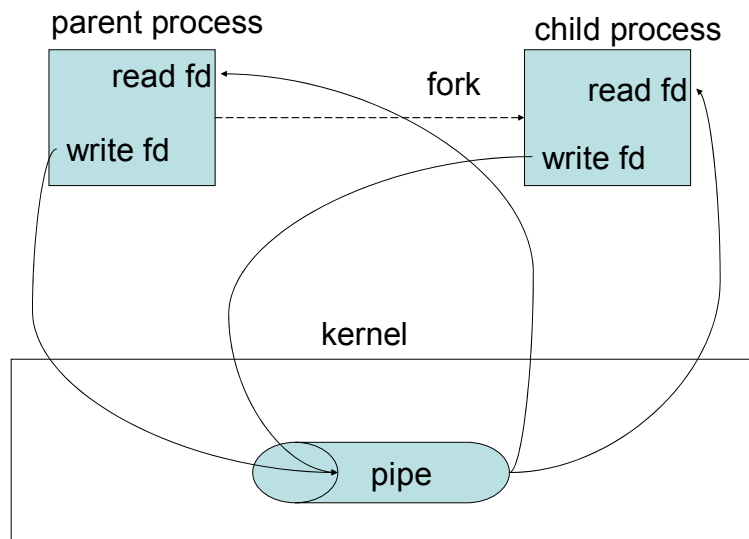
Pipe innerhalb eines Prozesses



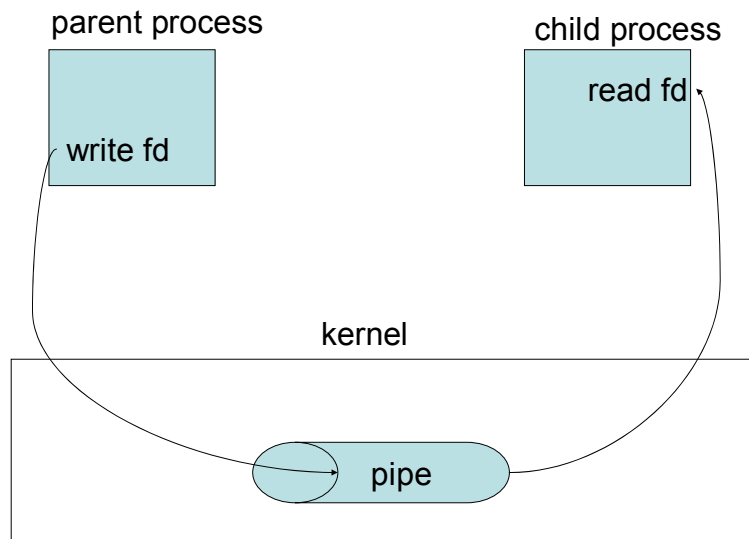
Normalerweise werden Pipes zur Kommunikation von zwei Prozessen verwendet. Dies geschieht in zwei Schritten:

1. Pipe erzeugen und Kopie von sich durch fork erzeugen
2. Vater schließt Lese-Ende, Kind schließt Schrei-Ende der Pipe.

1. Schritt: Pipe innerhalb eines Prozesses nach fork



2. Schritt: Vater schließt Lese-Ende, Kind schließt Schrei-Ende



Ein C-Programm, das eine solche Einwegverbindung realisiert, ist recht einfach:

```
$ cat spipe.c
/* Example: Father | Child */
#include <stdio.h>
#define BUFSIZE 20
main()
{
    int pid, status;
    int p[2]; /* 1==write, 0==read */
    char buf[BUFSIZE];

    if (pipe(p) != 0) {
        fprintf(stderr, "pipe error\n");
        exit(1);
    }
    switch (pid = fork()) {
        case -1: /* fork failed */
            fprintf(stderr, "cannot fork\n");
            exit(2);
        case 0: /* child: read from pipe */
            {int length;
             close(p[1]); /* close write end of pipe */
             do {
                 /* read from pipe */
                 length=read(p[0], buf, BUFSIZE);
                 fprintf(stdout, "%s", buf);
             } while (length>0);
            }
    }
}
```

```

        close(p[0]);
        exit(0);
    }
    default: /* father: write into pipe */
        {int i;
          close(p[0]); /* close read end of pipe */
          /* create example data */
          for (i=getpid(); i>0; i--) {
              sprintf(buf, "%d\n", i);
              /* write into pipe */
              write(p[1], buf, BUFSIZE);
              sleep(2); /* just to see it with ps -el" */
          }
          close(p[1]);
        }
        while (wait(&status) !=pid);
        exit(0);
    } /* switch */
}

```

Damit ist also eine unidirektionale Kommunikation vom Vater zum Kind realisiert.

Unser Client-Server Beispiel erfordert aber eine bidirektionale Kommunikation (Dateiname hin, Inhalt zurück). Dies wird wie folgt realisiert (vgl. dazu nächste Abbildung)

1. erzeuge Pipe1, erzeuge Pipe2

2. fork

3. Vater schließt Leseende von Pipe1

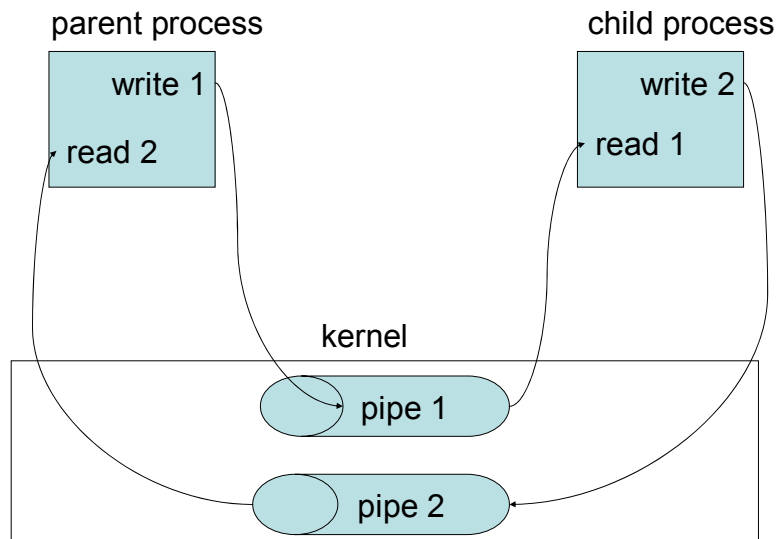
4. Vater schließt Schreibende von Pipe2

5. Kind schließt Schreibende von Pipe1

6. Kind Schließt Leseende von Pipe2

Es entsteht:

Bidirektionale Kommunikation mit zwei Pipes



Dies wird nun einfach in C implementiert. main erzeugt die Pipes und Kind. Der Client läuft als Vater, der Server als Kind.

```

#include <stdio.h>
#include "ipc.c"

#define MAXBUFF 1024

main()
{
    int childpid, pipe1[2], pipe2[2];

    if (pipe(pipe1) < 0 || pipe(pipe2) < 0)
        err_sys("can't create pipes");

    if ( (childpid = fork()) < 0) {
        err_sys("can't fork");
    } else if (childpid > 0) {          /* parent */
        close(pipe1[0]);
        close(pipe2[1]);

        client(pipe2[0], pipe1[1]);

        while (wait((int *) 0) != childpid) /* wait for child */
            ;

        close(pipe1[1]);
    }
}

```

```
    close(pipe2[0]);
    exit(0);
} else {                                /* child */
    close(pipe1[1]);
    close(pipe2[0]);

    server(pipe1[0], pipe2[1]);

    close(pipe1[0]);
    close(pipe2[1]);
    exit(0);
}
}
```

```

client(readfd, writefd)
int readfd;
int writefd;
{
    char    buff[MAXBUFF];
    int n;

    /* Read the filename from standard input, write it to the IPC descriptor. */

    if (fgets(buff, MAXBUFF, stdin) == NULL)
        err_sys("client: filename read error");

    n = strlen(buff);
    if (buff[n-1] == '\n')
        n--;          /* ignore newline from fgets() */
    if (write(writefd, buff, n) != n)
        err_sys("client: filename write error");

    /*Read the data from the IPC descriptor and write to standard output.  */

    while ( (n = read(readfd, buff, MAXBUFF)) > 0)
        if (write(1, buff, n) != n) /* fd 1 = stdout */
            err_sys("client: data write error");
    if (n < 0)
        err_sys("client: data read error");
}

```

```

server(readfd, writefd)
int readfd;
int writefd;
{
    char        buff[MAXBUFF];
    char        errmsg[256], *sys_err_str();
    int         n, fd;
    extern int  errno;

    /* Read the filename from the IPC descriptor. */

    if ( (n = read(readfd, buff, MAXBUFF)) <= 0)
        err_sys("server: filename read error");
    buff[n] = '\0';          /* null terminate filename */

    if ( (fd = open(buff, 0)) < 0) {
        /*Error.  Format an error message and send it back to the client. */
        sprintf(errmsg, ": can't open, %s\n", sys_err_str());
        strcat(buff, errmsg);
        n = strlen(buff);
        if (write(writefd, buff, n) != n)
            err_sys("server: errmsg write error");
    } else {
        /* Read the data from the file and write to the IPC descriptor. */

```

```
while ( (n = read(fd, buff, MAXBUFF)) > 0)
    if (write(writefd, buff, n) != n)
        err_sys("server: data write error");
if (n < 0)
    err_sys("server: read error");
}
```

Innerhalb der E/A Bibliothek existiert eine Funktion, die eine Pipe erzeugt und eine Kommunikation mit einem anderen Prozess erzeugt, der entweder von der Pipe liest oder in sie schreibt.

```
int *popen(char *command, char *type);
```

command ist ein von der Shell ausführbares Kommando. Dabei wird eine Pipe zwischen dem aufrufenden Prozess und dem Kommando erzeugt. Der Rückgabewert von `popen` ist ein FILE Zeiger, der entweder für Eingabe oder Ausgabe verwendet wird, in Abhängigkeit vom Wert von *type* ("r": aufrufender Prozess liest von stdin, "w": aufrufender Prozess schreibt auf stdout). Schlägt `popen` fehl, wird NULL zurück gegeben.

Um eine vorher geöffnete Pipe zu schließen, existiert die Funktion `pclose`.

```
int *popen(FILE *stream);
```

`popen` gibt den Exitstatus des aufgerufenen Kommandos zurück oder -1, wenn vorher kein `popen` aufgerufen wurde.

Somit ist die zweite Lösung für das Client-Server Problem damit recht einfach:

```
#include <stdio.h>

#define MAXLINE 1024

err_sys(char *str) { fprintf(stderr, str); }
```

```

main()
{
    int n;
    char    line[MAXLINE], command[MAXLINE + 10];
    FILE    *fp;

    /* Read the filename from standard input. */
    if (fgets(line, MAXLINE, stdin) == NULL)
        err_sys("filename read error");

    /* Use popen to create a pipe and execute the command. */
    sprintf(command, "cat %s", line);
    if ( (fp = popen(command, "r")) == NULL)
        err_sys("popen error");

    /* Read the data from the FILE pointer and write to standard output. */
    while ((fgets(line, MAXLINE, fp)) != NULL) {
        n = strlen(line);
        if (write(1, line, n) != n)
            err_sys("data write error");
    }

    if (ferror(fp)) err_sys("fgets error");

    pclose(fp); exit(0);
}

```

Der größte **Nachteil** von Pipes besteht darin, dass sie nur zur Kommunikation von Prozessen verwendet werden können, die einen gemeinsamen Vater haben. Dies ist so wegen des fork Systemaufrufes und der Tatsache, dass die geöffneten Dateien nach dem fork gemeinsam verwendet werden.

Somit gibt es **keine** Möglichkeit, unabhängige Prozesse mit Pipes kommunizieren zu lassen!

3. Fifo

FIFO ist die Abkürzung für First In, First Out. Somit ist die Abarbeitungsreihenfolge von Daten so wie bei Pipes - ein Einwegaustausch, bei dem das erste Byte, das geschrieben wird, auch gelesen wird.

Anders als bei Pipes gibt es bei FIFOs Namen, die die Struktur kennzeichnen und somit unabhängigen Prozessen zur Verfügung stehen. Deshalb nennt man FIFOs auch "named pipes".

Ein FIFO wird erzeugt durch den Systemaufruf:

```
int mknod(char *pathname, int mode, int dev);
```

Dabei ist *pathname*, ein normaler Dateiname von Unix, der Name des FIFO. Das Argument *mode* definiert den Zugriff (read, write mit Rechten für owner, group, others). *mode* wird logisch OR verknüpft mit dem Flag `S_IFIFO` (aus `<sys/stat.h>`), um auszudrücken, dass *mknode* einen FIFO erzeugen soll. *dev* wird für FIFOs ignoriert (relevant für andere Objekte).

FIFOs können auch durch das Unix Kommando *mknod* erzeugt werden:

```
/etc/mknod name p
```

Wenn ein FIFO erzeugt ist, muss es explizite zum Lesen oder Schreiben geöffnet werden (`open`, `read`, `fopen`, `fread`).

Für FIFOs (und Pipes) gelten folgenden Regeln für Lese- und Schreiboperationen:

- Wenn durch `read` weniger Daten angefordert werden, als im FIFO vorhanden sind, liefert nur die angeforderte Datenmenge zurück. Der restliche Inhalt des FIFO kann durch wiederholte `read` Kommandos ausgelesen werden.
- Wenn ein Prozess bei einer `read` Operation mehr Daten anfragt, als im FIFO vorhanden sind, so wird nur die vorhandene Datenmenge zurück geliefert. Der Leseprozess muss berücksichtigen, dass u.U. weniger Daten geliefert werden, als angefordert wurden.
- Wenn der FIFO leer ist und Daten angefordert werden (`read`) und zudem kein Prozess den FIFO zum Schreiben geöffnet hat, wird 0 zurückgeliefert; dies bedeutet dann EOF.
- Wenn ein Prozess weniger Daten in den FIFO schreiben (`write`) will, als der FIFO an Kapazität zur Verfügung stellt, so ist die `write` Operation **atomar**. D.h. Schreiben zwei Prozesse gleichzeitig in einen FIFO, so werden zuerst die Daten des ersten Prozesses, dann die Daten des zweiten Prozesses (oder umgekehrt) geschrieben; ein Mischen der Daten beider Prozesse ist somit verhindert.
Wenn jedoch nicht genug Platz im FIFO ist, kann Mischen nicht ausgeschlossen werden!
- Schreibt ein Prozess in einen FIFO, aber es existiert kein Prozess, der den FIFO zum Lesen geöffnet hat, dann wird das Signal `SIGPIPE` erzeugt und `write` gibt 0 zurück. Ist kein Signalhandler aktiv (Normalfall) dann terminiert der Schreibprozess.

Das Verhalten von `open` und `read` bei FIFOs (und Pipes) ist durch das Flag `O_NDELAY` (=no delay) bestimmt.

Bedingung	normal	O_NDELAY gesetzt (durch fcntl)
<code>open</code> FIFO, read-only, kein Prozess hat FIFO zum Schreiben offen	warten bis Prozess FIFO zum Schreiben geöffnet hat	return ohne Warten, kein Fehler
<code>open</code> FIFO, write-only, kein Prozess hat FIFO zum Lesen offen	warten bis Prozess FIFO zum Lesen geöffnet hat	return ohne Warten, Fehler (errno=ENXIO)
<code>read</code> FIFO, keine Daten	warten bis Daten in FIFO oder kein Prozess mehr vorhanden, der FIFO zum Schreiben offen hat (return 0); ansonsten return gelesene datenmenge	return ohne Warten mit Wert 0, kein Fehler
<code>write</code> FIFO, FIFO voll	warten bis ausreichend Platz, dann Schreiben der Daten in FIFO	return ohne Warten mit Wert 0, kein Fehler

Betrachten wir einen Dämon (z.B. print spooler), der auf Client Anfragen durch Lesen einer Pipe wartet. Wenn ein Client geschrieben hat und kein anderer Client die Pipe geöffnet hat, so müsste der Dämon von EOF ausgehen, die Pipe schließen und erneut öffnen, um weiterhin Anfragen entgegen nehmen zu können. Dies kann verhindert werden, indem der Dämon die Pipe nicht nur zum Lesen, sondern selbst zum Schreiben öffnet. Die Schreibseite wird aber nie benutzt.

Das Beispielproblem (CS cat) wird nun mit FIFOs gelöst. Dabei ändern sich die Funktionen `client()` und `server()` nicht. Lediglich das Hauptprogramm ist angepasst.:

```

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <stdio.h>
#include "ipc.c"
extern int errno;

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS 0666
#define MAXBUFF 1024

main()
{
    int childpid, readfd, writefd;

    if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST))
        err_sys("can't create fifo 1");
    if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
        unlink(FIFO1);
        err_sys("can't create fifo 2");
    }

    if ( (childpid = fork()) < 0) {
        err_sys("can't fork");
    }
}

```

```

} else if (childpid > 0) {          /* parent */
    if ( (writefd = open(FIFO1, 1)) < 0)
        err_sys("parent: can't open write fifo");
    if ( (readfd = open(FIFO2, 0)) < 0)
        err_sys("parent: can't open read fifo");

    client(readfd, writefd);

    while (wait((int *) 0) != childpid) /* wait for child */
        ;

    close(readfd);
    close(writefd);

    if (unlink(FIFO1) < 0)
        err_sys("parent: can't unlink FIFO1");
    if (unlink(FIFO2) < 0)
        err_sys("parent: can't unlink FIFO2");
    exit(0);

} else {          /* child */
    if ( (readfd = open(FIFO1, 0)) < 0)
        err_sys("child: can't open read fifo");
    if ( (writefd = open(FIFO2, 1)) < 0)
        err_sys("child: can't open write fifo");

```

```
server(readfd, writefd);
```

```
close(readfd);
```

```
close(writefd);
```

```
exit(0);
```

```
}
```

```
}
```

```

client(readfd, writefd)
int readfd;
int writefd;
{
    char    buff[MAXBUFF];
    int n;

    /*
     * Read the filename from standard input,
     * write it to the IPC descriptor.
     */

    if (fgets(buff, MAXBUFF, stdin) == NULL)
        err_sys("client: filename read error");

    n = strlen(buff);
    if (buff[n-1] == '\n')
        n--;          /* ignore newline from fgets() */
    if (write(writefd, buff, n) != n)
        err_sys("client: filename write error");

    /*
     * Read the data from the IPC descriptor and write
     * to standard output.
     */
}

```

```
while ( (n = read(readfd, buff, MAXBUFF)) > 0)
    if (write(1, buff, n) != n) /* fd 1 = stdout */
        err_sys("client: data write error");
if (n < 0)
    err_sys("client: data read error");
}
```

```

server(readfd, writefd)
int readfd;
int writefd;
{
    char        buff[MAXBUFF];
    char        errmsg[256], *sys_err_str();
    int         n, fd;
    extern int  errno;

    /*
     * Read the filename from the IPC descriptor.
     */

    if ( (n = read(readfd, buff, MAXBUFF)) <= 0)
        err_sys("server: filename read error");
    buff[n] = '\0';          /* null terminate filename */

    if ( (fd = open(buff, 0)) < 0) {
        /*
         * Error.  Format an error message and send it back
         * to the client.
         */

        sprintf(errmsg, ": can't open, %s\n", sys_err_str());
        strcat(buff, errmsg);
        n = strlen(buff);
    }
}

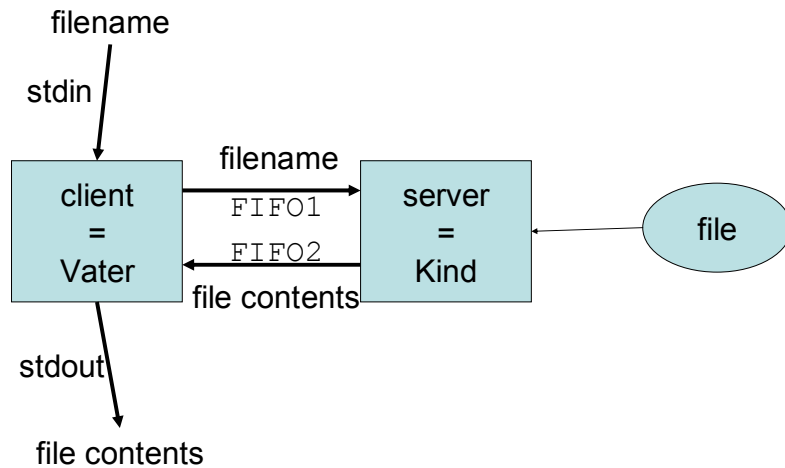
```

```
    if (write(writefd, buff, n) != n)
        err_sys("server: errmesg write error");
} else {
    /*
     * Read the data from the file and write to
     * the IPC descriptor.
     */

    while ( (n = read(fd, buff, MAXBUFF)) > 0)
        if (write(writefd, buff, n) != n)
            err_sys("server: data write error");
    if (n < 0)
        err_sys("server: read error");
}
}
```

Zuerst werden in `main` die FIFOs erzeugt. Nach dem `fork` muss sowohl Vater als auch Kind die FIFOs öffnen. Der Vater wartet, bis das Kind terminiert ist, dann werden die FIFOs geschlossen.

Client-Server mit FIFOs



Vorsicht (Deadlockgefahr):

Die Reihenfolge der `open` Aufrufe im o.a. Beispiel ist wichtig. Wird die Reihenfolge vertauscht, entsteht ein Deadlock: Wenn der Vater FIFO1 zum Schreiben öffnet, wartet er, bis das Kind FIFO1 zum Lesen geöffnet hat. Wäre die Reihenfolge beim Kind vertauscht, so würde das Kind warten, bis der Vater FIFO2 zum Schreiben geöffnet hat; Vater wartet auf Kind, Kind wartet auf Vater - Deadlock.

Der eigentliche Vorteil von FIFOs (unabhängige Prozesse können Daten austauschen) wird in der folgenden Realisierung des CS-cat Problems verdeutlicht. Es sind zwei Programme, eines für Server, eins für den Client.

```
#include <stdio.h>
#include "fifo.h"
#include "ipc.c"

#define MAXBUFF 1024

main()
{
    int readfd, writefd;

    /* Open the FIFOs. We assume the server has already created them. */

    if ( (writefd = open(FIFO1, 1)) < 0)
        err_sys("client: can't open write fifo: FIFO1");
    if ( (readfd = open(FIFO2, 0)) < 0)
        err_sys("client: can't open read fifo: FIFO2");

    client(readfd, writefd);

    close(readfd);
    close(writefd);

    /* Delete the FIFOs, now that we're finished. */

    if (unlink(FIFO1) < 0)
        err_sys("client: can't unlink FIFO1");
}
```

```
if (unlink(FIFO2) < 0)
    err_sys("client: can't unlink FIFO2");

exit(0);
}
```

```

client(readfd, writefd)
int readfd;
int writefd;
{
    char    buff[MAXBUFF];
    int n;

    /* Read the filename from standard input, write it to the IPC descriptor. */
    if (fgets(buff, MAXBUFF, stdin) == NULL)
        err_sys("client: filename read error");

    n = strlen(buff);
    if (buff[n-1] == '\n')
        n--;          /* ignore newline from fgets() */
    if (write(writefd, buff, n) != n)
        err_sys("client: filename write error");

    /* Read the data from the IPC descriptor and write to standard output. */
    while ( (n = read(readfd, buff, MAXBUFF)) > 0)
        if (write(1, buff, n) != n) /* fd 1 = stdout */
            err_sys("client: data write error");
    if (n < 0)
        err_sys("client: data read error");
}

```

```

#include <stdio.h>
#include "fifo.h"
#include "ipc.c"

#define MAXBUFF 1024

main()
{
    int readfd, writefd;

    /* Create the FIFOs, then open them - one for reading and one for writing. */
    if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST))
        err_sys("can't create fifo: FIFO1");
    if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
        unlink(FIFO1);
        err_sys("can't create fifo: FIFO2");
    }

    if ( (readfd = open(FIFO1, 0)) < 0)
        err_sys("server: can't open read fifo: FIFO1");
    if ( (writefd = open(FIFO2, 1)) < 0)
        err_sys("server: can't open write fifo: FIFO2");

    server(readfd, writefd);

    close(readfd);
}

```

```
close(writefd);  
  
exit(0);  
}
```

```

server(readfd, writefd)
int readfd;
int writefd;
{
    char        buff[MAXBUFF];
    char        errmsg[256], *sys_err_str();
    int         n, fd;
    extern int  errno;

    /* Read the filename from the IPC descriptor. */
    if ( (n = read(readfd, buff, MAXBUFF)) <= 0)
        err_sys("server: filename read error");
    buff[n] = '\0';          /* null terminate filename */

    if ( (fd = open(buff, 0)) < 0) {
        /* Error.  Format an error message and send it back to the client. */
        sprintf(errmsg, ": can't open, %s\n", sys_err_str());
        strcat(buff, errmsg);
        n = strlen(buff);
        if (write(writefd, buff, n) != n)
            err_sys("server: errmsg write error");
    } else {

```

```
/* Read the data from the file and write to the IPC descriptor. */
while ( (n = read(fd, buff, MAXBUFF)) > 0)
    if (write(writefd, buff, n) != n)
        err_sys("server: data write error");
if (n < 0)
    err_sys("server: read error");
}
}
```

Die Programme sind etwa wie folgt zu starten (Server im Hintergrund, Client interaktiv):

```
$ ipc05s &
$ ipc05c
Dateiname
Dateiinhalte
.....
$
```

3.1. Übung IPC-2

Entwickeln Sie Client Server Programm, das die Funktionalität von "cat Datei" abbildet. Dabei soll der Dateiname dem Server über einen FIFO mitgeteilt werden und der Dateiinhalt über einen FIFO zurückgegeben werden (wie CS cat). Zusätzlich soll der Server einem Log-Dämon den Dateinamen über einen FIFO mitteilen, der dann jeden Request mit Datum, Uhrzeit und Dateiname in einer Datei ablegt. Der Log-Dämon muss den Client informieren, wenn mehr als 10 mal die selbe Datei angefordert wurde.

Verdeutlichen Sie sich die Kommunikationsstruktur, bevor Sie anfangen zu programmieren. Begründen Sie Ihren Ansatz.