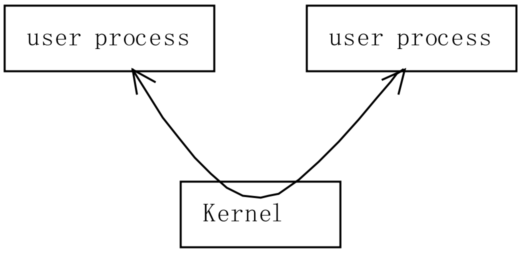
Lecture 2: Interprocess Communication--Pipes

References for Lecture 2:

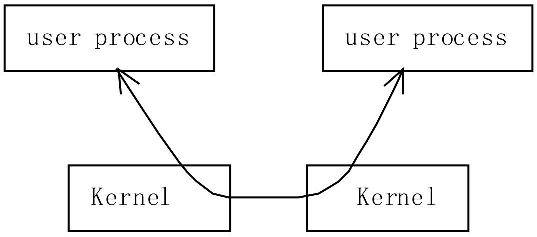
1) Unix Network Programming, W.R. Stevens, 1990,Prentice-Hall, Chapter 3.
2) Unix Network Programming, W.R. Stevens, 1999,Prentice-Hall, Chapter 2-6.

Purposes:

Communication is everywhere from intraprocess to Interprocess.
Interprocess communication has 2 forms:



IPC on one host



IPC on different hosts:
Network Programming

IPC is used for 2 functions:

1) Synchronization---Used to coordinate access to resources among processes and also to coordinate the execution of these processes. They are
   Record locking,
   Semaphores,
   Mutexes and Condition variables.
2) Message Passing---Used when processes wish to exchange information. Message passing takes several forms such as :
   Pipes,
   FIFOs,
   Message Queues,
   Shared Memory.

File or Record Locking:

Used to ensure that a process has exclusive access to a file before using it. Examples: 1) lpr generates a unique sequence number for each print job; see Stevens90:89-91. 2) Couple(husband, wife) share one bank account. In Unix, record locking is better termed as range locking.

For System V:

#include <unistd.h>

int lockf(int *fd*, int *function*, long *size*);

fd---file descripter(not a file pointer)

size--- define the record size or lock area: [offset, offset + size]. size=0 means the rest of the file. Use lseek() to move the current offset. When the offset position is set to the beginning and size=0 then lock the whole file.

function:

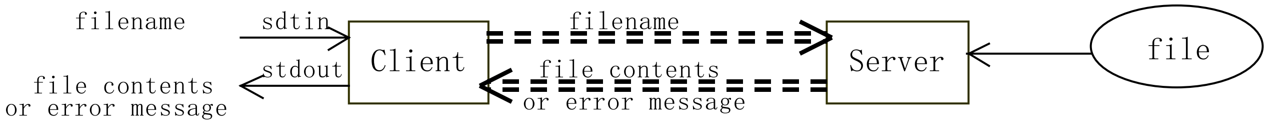
F_ULOCK---unlock a previous lock
F_LOCK ---lock a region(blocking)
F_TLOCK ---Test and lock a region(nonblocking)
F_TEST ---Test a region to see if it is locked.

Example: Use F_TLOCK instead of F_TEST and F_LOCK.

```
…
If (lockf(fd, F_TEST, size)= =0) /* If the region is locked, -1 is returned and the process is in sleep state*/
Re= lockf(fd, F_LOCK, size);    /*a small chance that another process locks between TEST and LOCK*/
 …
rc=lockf(fd, F_TLOCK, size) /* Test + lock done as an atomic operation, If unsuccessful, lockf()
                                         returns –1 and the calling process continues to do other things*/
```

Atomic operation: one or more operations that are treated as a single operation. No other operation can be executed between the start and end of an atomic operation.

Simple Client-Server or IPC model:



Client reads filename from sdtin, write it to the IPC channel, read the file contents from the IPC channel

Server reads filename from the IPC channel, opens file, write the contents on the IPC channel

Pipes:

A pipe provides a one-way flow of data.
Int pipe (int * *filedes*);
Int pipefd[2]; /* pipefd[0] is opened for reading;pipefd[1] is opened for writing */

Example to show how to create and use a pipe:

```
main()
{       int pipefd[2], n;
        char buff[100];
         if (pipe(pipefd) < 0 ) err_sys(“pipe error”);

         printf(“read fd = %d, write fd = %d\n”, pipefd[0], pipefd[1]);
         if (write(pipefd[1], “hello world\n”, 12) != 12) err_sys(“write
     error”);

         if ( (n=read(pipefd[0], buff, sizeof(buff))) < =0) err_sys(“read error”);
         write(1, buff, n); /*fd=1=stdout*/
}
```

Difference:

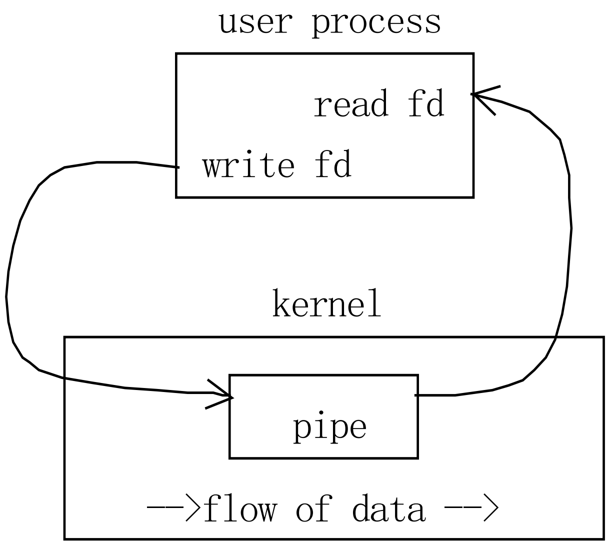
① { *Argument*
Parameter

② { *Call by value*
Call by reference

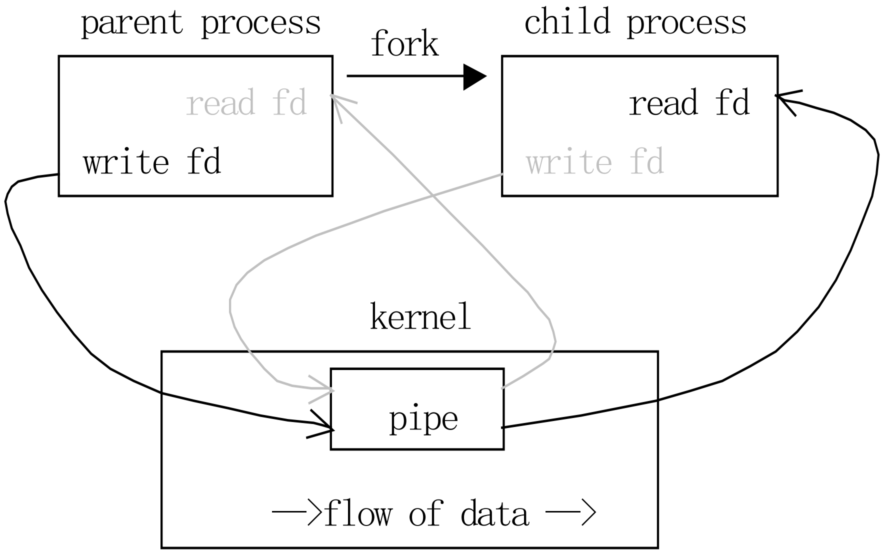
③ { *File descriptor*
File pointer

result:

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

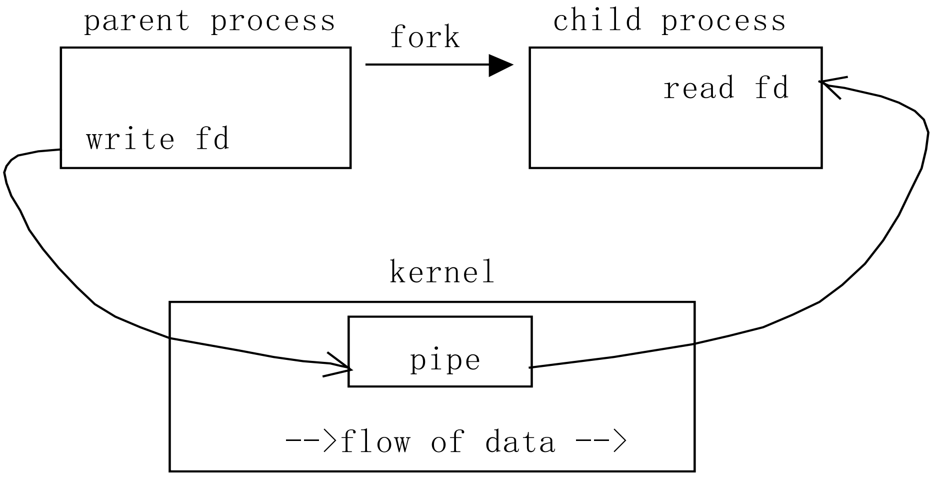


Pipe in a single process



Pipe in a single process, immediately after fork

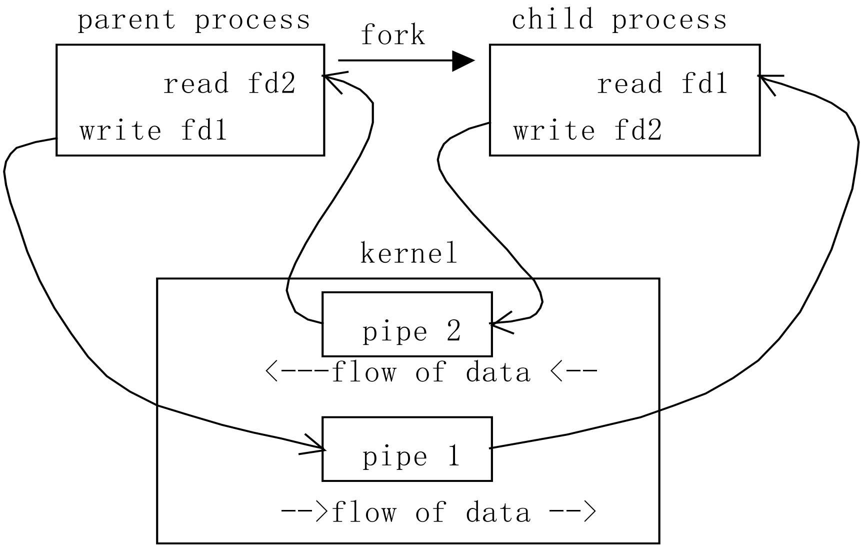
Pipes between two processes: unidirectional



1 pipe between two process: one-way

Steps : 1) opening a pipe
2) forking off another process
3) closing the oppropriate pipes on each end

Pipes between two processes: bidirectional



2 pipes to provide a bidirectional flow of data

Steps: 1) create pipe1 + pipe2 : int pipe1[2], pipe2[2] -----must be the first step
2) forking off a child process, executing another program as a server
3) parent closes read end of pipe 1 + write end of pipe 2,
4) child closes write end of pipe 1 + read end of pipe 2

Question: Can we use only one pipe to finish bi-directional communications?

Program Example of Simple Client-Server Model:

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

            /* step 1: create pipe1 and pipe2 */
            if (pipe(pipe1) < 0 || pipe(pipe2) < 0)
                err_sys("can't create pipes");

            /*step 2: fork a child process */
            if ( (childpid = fork()) < 0) {
                err_sys("can't fork");

            /* step 3: parent closes read end of pipe 1 and write end of pipe 2*/
            } else if (childpid > 0) {
                close(pipe1[0]);
                close(pipe2[1]);

                client(pipe2[0], pipe1[1]); /*client runs in the parent process*/

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

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

            /* step 4: child closes write end of pipe 1 and read end of pipe 2*/
                close(pipe1[1]);
                close(pipe2[0]);

                server(pipe1[0], pipe2[1]); /*server runs in the child process */

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

You may need to replace **err_sys(…)** by **printf(…)** or **perror(…)** for the program to run on your computer.

```
#include        <stdio.h>

#define         MAXBUFF         1024

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

                /* read the filename from standard input */
                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() */

                /* write it to the IPC descriptor, pipe1[1] */
                if (write(writefd, buff, n) != n)
                   err_sys("client: filename write error");;

                /* read the data from the IPC descriptor, pipe2[0],
                   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");
}
```

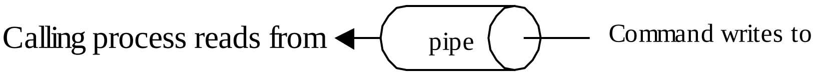
```
#include        <stdio.h>
#define         MAXBUFF         1024

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

                /* read the filename from the IPC descriptor, pipe1[0]*/
                if ( (n = read(readfd, buff, MAXBUFF)) <= 0)
                   err_sys("server: filename read error");
                buff[n] = '\0';       /* null terminate filename */

                /* open the file from the IPC descriptor, pipe1[0]*/
                if ( (fd = open(buff, 0)) < 0) {
                   /* Error.   Format an error message and send it back to the client.      */
                   sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                   strcat(buff, errmesg);
                   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");
                }
}
```

Properties of Pipe:

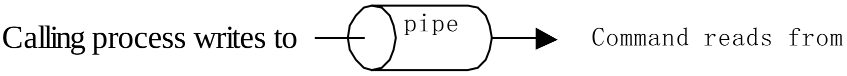
1) Pipes do not have a name. For this reason, the processes must share a parent process. This is the main drawback to pipes. However, pipes are treated as file descriptors, so the pipes remain open even after fork and exec.
2) Pipes do not distinguish between messages; they just read a fixed number of bytes. Newline (\n) can be used to separate messages. A structure with a length field can be used for message containing binary data.
3) Pipes can also be used to get the output of a command or to provide input to a command

```
FILE *popen(const char *command, const char *type);
Int pclose(FILE *stream);
```

When type is “r”:



When type is “w”:



For example:

```
#include <stdio.h>
#define MAXLINE 1024
main()
{ int n;
    char line[MAXLINE];
    FILE *fp;

    fp=popen(“cat .cshrc”, “r”);

    \*read the lines in .cshrc from fp*\
    while ((fgets(line, MAXLINE, fp)) != NULL) {
      n=strlen(line);
      if (write(1, line, n)!=n) printf(“print data error”);
    pclose(fp);
}
```

Notes:

- Please notice the difference between fgets(…) and read(…). See our website FAQ.
- Not every Unix command has output such as mv, cp and …. In such cases, fgets( ) and read( ) will return NULL when reading from fp=popen(…).
- **cd** is a special Unix command. You cannot use popen(“cd”,”r”) or system(“cd”).