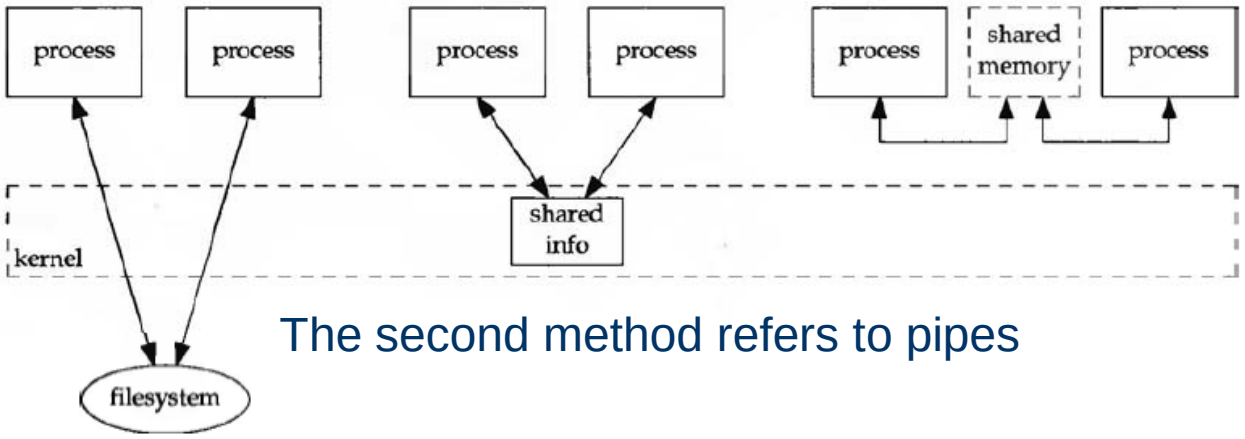
Three ways to implement Interprocess Communications



The second method refers to pipes

Figure 1.1 Three ways to share information between Unix processes.

Sources for these slides include:

- W. Stevens, Unix Network Programming, Volumes 1 and 2
- M. Mitchell, J. Oldham, A. Samuel, Advanced Linux Programming

Interprocess Communication (IPC)

- Linux supports the following IPC mechanisms:
  - Half duplex pipes
  - Full duplex pipes –only by using 2 pipes (other unix systems do support FD over a single pipe)
  - FIFOS (also called named pipes)
  - SYSV style message queues
  - SYSV style shared memory
  - Local Domain (UNIX Domain) sockets – sockets but modifications to the address/name aspect.
  - Network Domain sockets – since can be locally (with address of ‘localhost’, it’s about equivalent to UNIX Domain sockets

Pipe – used in a shell pipeline

- Example of a pipeline - issue the following in a shell:
  - **who | sort | wc**
    2 10 110
  - The shell program creates three processes, two pipes are used as shown below.
  - The shell would use the dup2 call to duplicate the read end of each pipe to standard input and the write end to standard output.
    - int fd[2];
    - pid_t pid;
    - pipe(fd);
    - pid = fork();
    - If(pid == 0) {
      - dup2(fd[0], STDIN_FILENO);
      - exec(whatever);
    - }

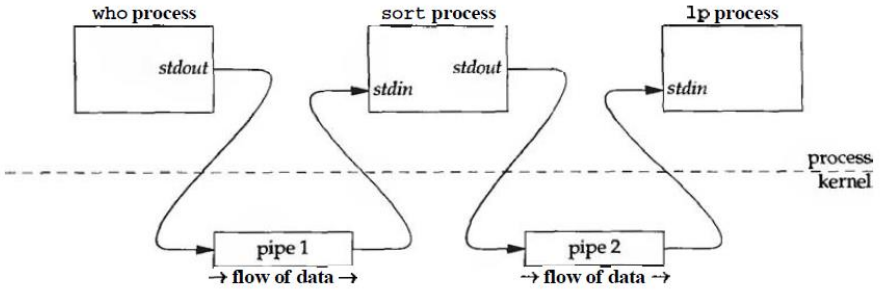


Figure 4.5 Pipes between three processes in a shell pipeline.

- The use of dup2 by a shell allows a program or filter to simply operate on data arriving through standard in….and sends output to standard out. It does not need to know the pipe descriptors involved….

  - Note – modified the pipeline- the figure assumes lp instead of wc

.

One-way pipes

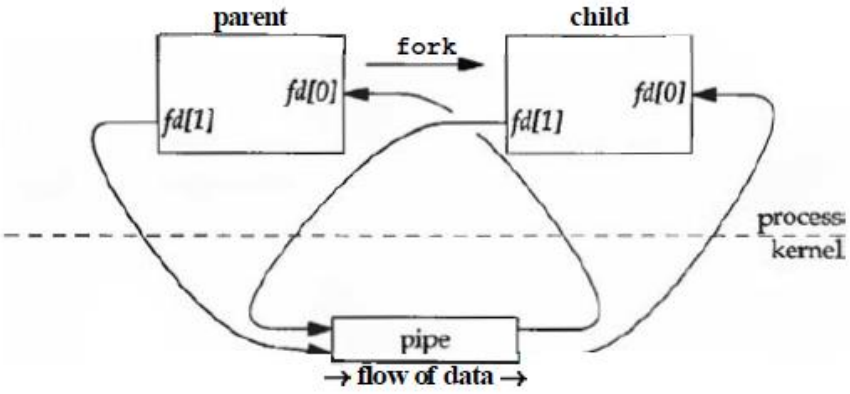


Figure 4.3 Pipe in a single process, immediately after `fork`.

This example involves a parent writing to a child over a pipe

- Parent creates pipe and child process
- Since only the parent writes, the child closes fd[1].
- Child reads from the pipe one char at a time
- Echos to standard out
- While loop terminates once the pipe returns error (from a close)
- Parent closed its read descriptor, writes a single character then closes the pipe.
- Wait is an alternative to waitpid – it suspends the parent until ONE of its children terminate
  - Equivalent to waitpid(-1,&status, 0)

One-way Pipe - simpleEx.c

Based on previous page….. See cCodeex3, simplleEx.c

```
Int pipefd[2]; //pipefd[0] is reader, pipefd[1] is writer
  pid_t cpid;
  int rc = EXIT_SUCCESS;
  char buf[1024];
  char *bufPtr=buf;
  rc =pipe(pipefd); //ignore error
  cpid = fork(); //ignore error case...
  if (cpid == 0)
  {
     close(pipefd[1]);          /* Close unused write end */
     while (read(pipefd[0], bufPtr, 1) > 0) {
       rc = write(STDOUT_FILENO, bufPtr, 1); //ignore error case
       printf("simpleEx(child):  write returns %d bytes, string: %c  \n",rc, buf[0]);
     }
     rc = write(STDOUT_FILENO, "\n", 1); //ignore error case
     close(pipefd[0]);
     exit(EXIT_SUCCESS);
  } else
  {
    /* Parent writes argv[1] to pipe */
    close(pipefd[0]);          /* Close unused read end */
    rc = write(pipefd[1], argv[1], strlen(argv[1]));
    printf("simpleEx(partent: sent %d bytes over pipe (%s) \n",rc, argv[1]);
    close(pipefd[1]);          /* Reader will see EOF */
    wait(NULL);                /* Wait for child */
  }
  return rc;
```

Full duplex using two pipes – mainpipe.c

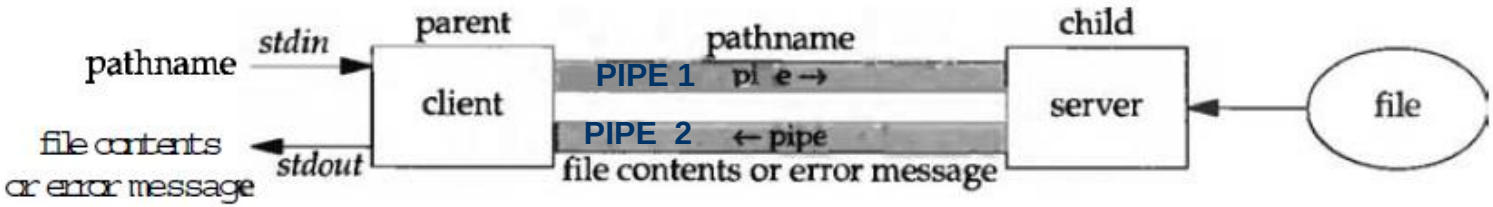


Figure 4.7 Implementation of Figure 4.1 using two pipes.

- Example: see cCodeex3, make mainpipe1 (mainpipe.c, client.c, server.c)
- A parent process creates two pipes, pipe1 is for data sent from the client to the server. And pipe 2 is for data sent from he server to the client.
  - int pipe1[2], pipe2[2]; rc = pipe(pipe1); rc = pipe(pipe2
- The parent forks a child process which invokes the server: server(pipe1[0], pipe2[1]);
- The parent invokes the client : rc =client(pipe2[0], pipe1[1]);
- The client program gets a file name from standard in (user enters it) and writes the information to the server program over pipe 1.. The client then prepares to loop, reading data from Pipe 2 (the contents of the file)
- The server program reads the file name from Pipe 1, opens the file, and sends the contents over pipe 2.

Alternative solution usng popen (and cat)

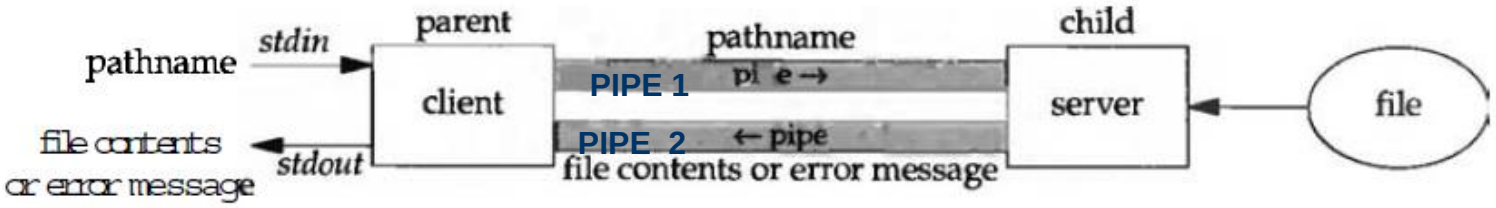


Figure 4.7 Implementation of Figure 4.1 using two pipes.

- Example: see cCodeex3, make mainpopen, source mainpopen.c
- File *popen(const char *command, const char *type) : pipe stream to or from a related process
  - The Command is a shell command line.
  - If Type is r, the calling process reads the standard output of the command
  - If Type is w, the calling process writes to the standard input of the command
- It greatly simplifies things as it replaces a fork, exec, pipe setup. It setups the child process to run a program and allows the parent and child to use standard i/o using the descriptors for reads and writes.

- The example program creates a command : “cat file “ where the file is entered by the user (so the calling process should read (cat) to standard out)
  - fp = popen(command, "r"); //a child process will now cat the file to the pipe
  - //The parent process simply loops and reads the data (the file contents)
  - while (fgets(buff, MAXLINE, fp) != NULL)
  - fputs(buff, stdout);
  - pclose(fp);

Full duplex pipe – fduplex.c

- This appears to not work on Linux….which confirms that to support full duplex pipes, two unidirectional pipes are required.
- Results when running on Linux:
  - ./fdplexpipe1
  - ./fdplexpipe1: parent: Succeeded to create pipe: fd[0]:3 fd[1]:4
  - ./fdplexpipe1: parent: Succeeded to write 1 byte, rc:1
  - ./fdplexpipe1: parent: read rc : -1 errno:9
  - ./fdplexpipe1: parent: Error on read 1, errno:9
  - jjm@jjm-VirtualBox:~/courses/codeExamples/cCode/cCodeex3/V1$ ./fdplexpipe1: child read p
  - ./fdplexpipe1: child write error, errnno:9
- TODO: Run on FreeBSD and MACOS

Named Pipes – first with related processes

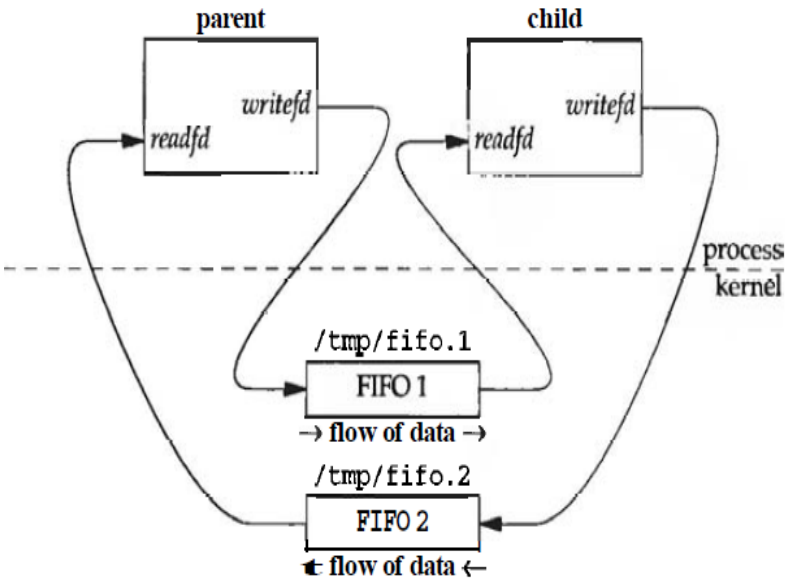


Figure 4.17 Client–server example using two FIFOs.

See ./cCodeex4/V1 mainfifo.c

- Make mainfifo1
- To run: ./mainfifo
  - ./mainfifo1
  - ./mainfifo1(Version:Version 1.00) pid:10901 Entered with 1 arguements
  - client(10901): Please enter the file name (it can be the full path or just the name of a file in the cur dir
  - server: Entered, readfd:3, writefd:4
- Then enter a file name (with a path if needed).
  - I know that readme.txt is in the cur dir so I enter that and I see the contents displayed to std out.
- The include fifo.h id's the named pipes
  - #define FIFO1 "./fifo.1"
  - fifo.h:#define FIFO2 "./fifo.2"
  - fifo.h:#define SERV_FIFO "./fifo.serv"
- It issues two calls to mkfifo
  - mkfifo(FIFO1, FILE_MODE)
  - mkfifo(FIFO2, FILE_MODE)
- The parent forks a child that opens FIFO1 and FIFO2 for RDONLY and WRONLY resp.. The child then calls the server program
- The parent then calls the client program.

Named Pipes – second with unrelated processes

In cCodeex4/V1 there are two example programs.

- Issue ‘make pirnt-PROGS’ to show all programs
- ‘make clientfifo1’ and ‘make serverfifo1’.
  - These run the client and server as separate programs. They isuse the ‘make Parent creates pipe and child process
- The client issues a pair of mkfifo calls and then does the following
  - writefd = open(FIFO1, O_WRONLY, 0);
- readfd = open(FIFO2, O_RDONLY, 0);
  - client(readfd, writefd);
- The server issues a pair of mkfifo calls and then does the following:
- readfd = open(FIFO1, O_RDONLY, 0);
  - writefd = open(FIFO2, O_WRONLY, 0);
  - rc = server(readfd, writefd);

The second program requires the two programs mainclient.c and mainserver.c to be built and run.

- This program goes one small step further….and demonstrates a server that can handle any number of clients.
- This is the real advantage of using Named Pipes…it’s an easy way for unrelated programs to exchange data.