Chapter 10:Redirection and Pipes

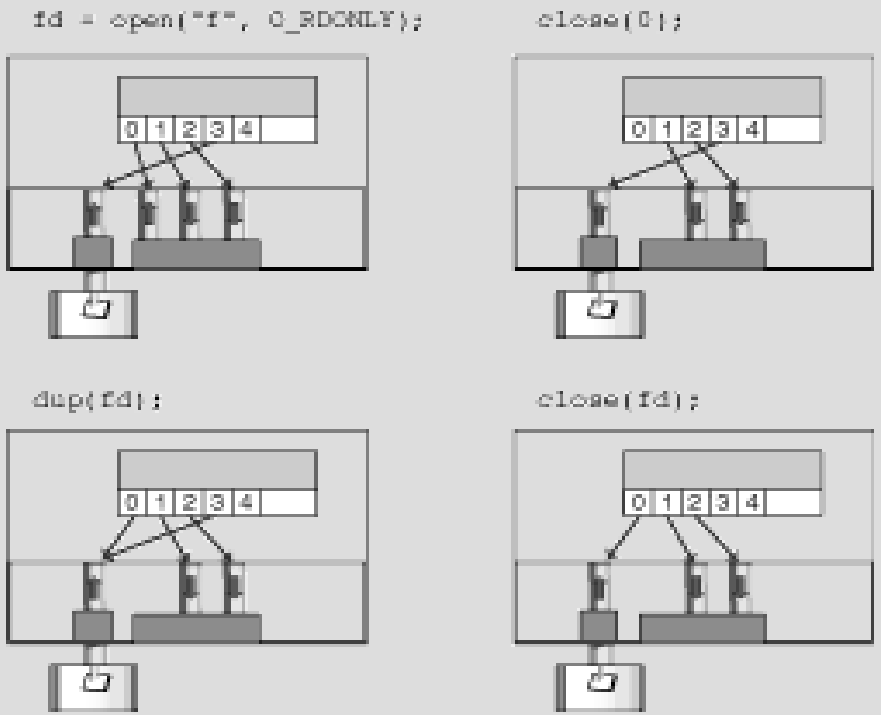
- Unix Philosophy
  - Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
  - Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

Exercise 8.12(Due March 16)

- Determine how a shell responses to SIGINT and SIGQUIT generated from the keyboard.Modify psh2.c so that it responds in the same way. Hint: it might be helpful to investigate what the SA_RESTART flag does in sigaction.

Shell Redirection and Pipes

- zcat `man -w bash` | man2html > sh.html
- tar cf – current | gzip > current.tar.gz
- Unix always assigns the lowest unused file descriptor
- dup(fd) copies the file descriptor fd to the lowest available



IO redirect

- close(0); fd=open(...)
- fd=open(...), close(0), dup(fd)
- fd=open(...),dup2(fd,0)

execve

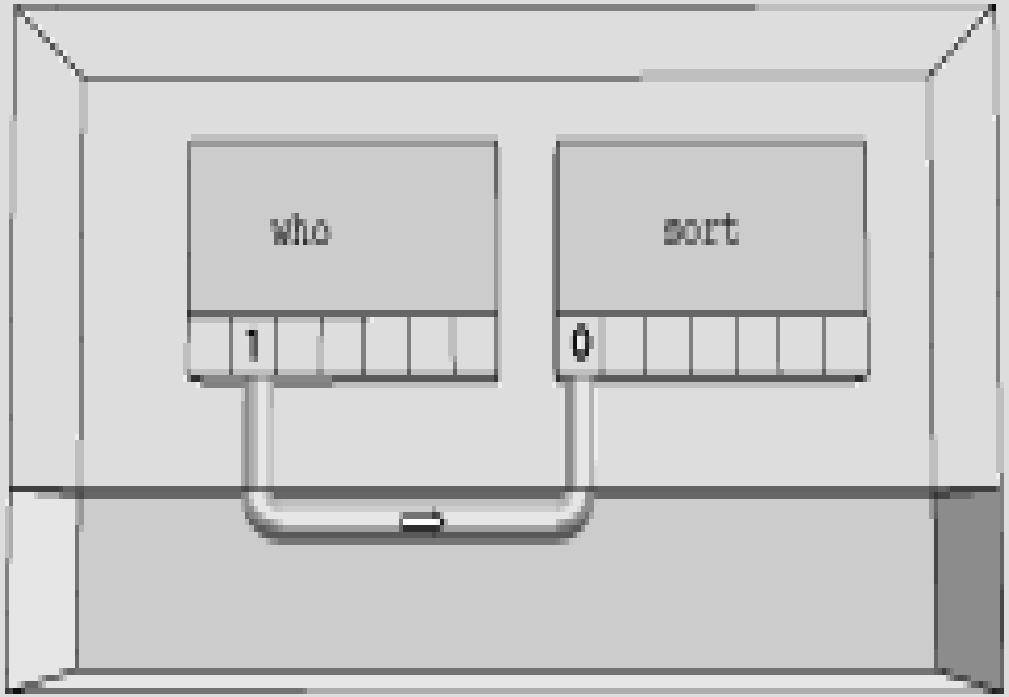
- The program invoked inherits the calling process's PID, and any open file descriptors that are not set to close on exec. Signals pending on the calling process are cleared. Any signals set to be caught by the calling process are reset to their default behaviour.

redirecting IO in shell

- fork()
- open required redirection file descriptors in child
- dup or dup2 standard io in child
- exec code file in child

Pipes

who | sort

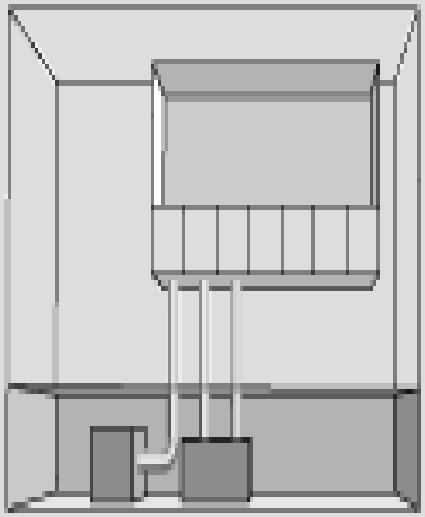


Pipes

- NAME pipe - create pipe
  SYNOPSIS
  #include <unistd.h>
  int pipe(int filedes[2]);
  DESCRIPTION
  pipe creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.
  RETURN VALUE
  On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

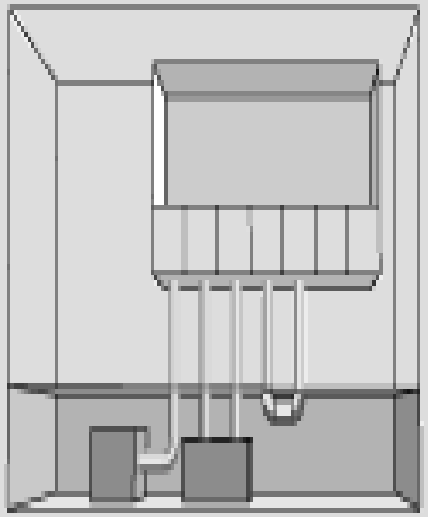
Pipes before-after

Before `pipe`



The process has some usual files open.

After `pipe`

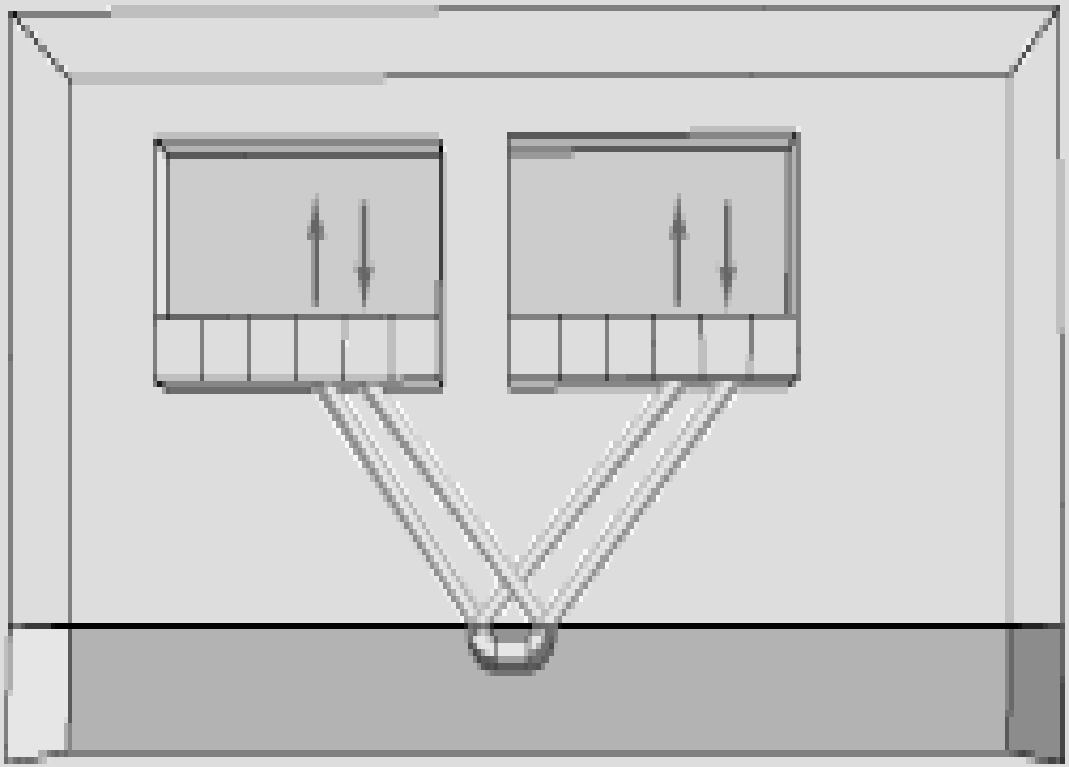


The kernel creates a pipe and sets file descriptors.

Pipedemo.c

```
#include        <stdio.h>
main()
{       int         len, i,
                    apipe[2];               /* two file descriptors */
            char    buf[BUFSIZ];            /* for reading end      */
            if ( pipe ( apipe ) == -1 ){
                    perror("could not make pipe");exit(1);}
            printf("Got a pipe! It is file descriptors: { %d %d }\n", apipe[0], apipe[1]);
            while ( gets(buf) ){                            /* get next line */
                    len = strlen( buf );
                    if (  write( apipe[1], buf, len) != len ){          /* send */
                            perror("writing to pipe");                  /* down */
                            break;}                                     /* pipe */
                    for ( i = 0 ; i<len ; i++ ) buf[i] = 'X' ;
                    len = read( apipe[0], buf, BUFSIZ ) ;               /* read */
                    if ( len == -1 ){                                   /* from */
                            perror("reading from pipe");                /* pipe */
                            break;}
                    if ( write( 1 , buf, len ) != len ){                /* send  */
                            perror("writing to stdout");                /* to    */
                            break;}                                     /* stdout */
                    write( 1, "\n", 1 );
            }
}
```

Pipes after fork



pipedemo2.c

```
#include        <stdio.h>
#defineCHILD_MESS          "I want a cookie"
#definePAR_MESS    "testing.."
main()
{       int       pipefd[2];              /* the pipe      */
         int       len;                    /* for write     */
         char      buf[BUFSIZ];            /* for read      */
         int       read_len;
         if ( pipe( pipefd ) == -1 ){
                   perror("cannot get a pipe");
                   exit(1);}
         switch( fork() ){
                   case -1:fprintf(stderr,"cannot fork");
                            exit(1);
                   case 0: len = strlen(CHILD_MESS);
                            while ( 1 ){
                                     if (write( pipefd[1], CHILD_MESS, len) != len ) exit(2);sleep(5);}
                   default:len = strlen( PAR_MESS );
                            while ( 1 ){
                                     if ( write( pipefd[1], PAR_MESS, len)!=len ) exit(3);sleep(1);
                                     read_len = read ( pipefd[0], buf, BUFSIZ );
                                     if ( read_len <= 0 )break;
                                     write( 1 , buf, read_len );
                                     write( 1, "\n", 1 );}
         }
}
```

Technical Details

- read() on a pipe blocks until data appears
- write() on a pipe blocks until space is available in the pipe
- When all writers close the writing end read returns 0 (i.e. eof)
- Multiple readers cause problems.
- When all readers have closed the reading end then write causes a SIGPIPE

Named Pipes

- Problem with pipes: really only connects parent and child
- What about process that aren't related? use named pipes: fifo's
  - Works like a pipe but looks like a file
  - can be opened and write to or read from like a file.

Fifo's

NAME
mkfifo - make FIFOs (named pipes)
SYNOPSIS
mkfifo [OPTION] NAME...
DESCRIPTION
Create named pipes (FIFOs) with the given NAMEs.
-m, --mode=MODE
set permission mode (as in chmod), not a=rw - umask
--help
display this help and exit
--version

output version information and exit

fifo_srvr.c

```
#include <stdio.h>
#include <sys/stat.h>  /* for mkfifo */
#include <sys/types.h>  /* for open */
#include <sys/stat.h>  /* for open */
#include <fcntl.h>    /* for open */
#include <unistd.h>   /* for write */
#define BUFFSIZE 8192
main () {
  int fd,n;
    char buf[BUFFSIZE];
    unlink("apipe");
    if (-1 == mkfifo("apipe",S_IRWXU)) { perror("Error, could not make fifo\n");}
    if ((fd = open("apipe",O_RDONLY)) == -1) {perror("open");exit(1);}
    while ((n=read(fd,buf,BUFFSIZE))>0)
      if (write(1,buf,n) != n) {printf("cp: write error on file stdout\n"); exit(1);}
exit(0);
}
```

fifo_clnt.c

```
#include <stdio.h>
#include <sys/stat.h>  /* for mkfifo */
#include <sys/types.h>  /* for open */
#include <sys/stat.h>  /* for open */
#include <fcntl.h>    /* for open */
#include <unistd.h>   /* for write */
main () {
   int fd,i;
   int len,n;
   char buf[100];
   if ((fd = open("apipe",O_WRONLY))<0){perror("open in client" ); exit(1);};
      for (i=0;i<11;i++) {
       len = sprintf(buf,"From the writer, this is item %d\n",i);
      n=write(fd,buf,len);
      printf("client: wrote %d bytes\n",n);
   }
   close(fd);
   exit(0);
}
```

working with shared libraries

- library path: LD_LIBRARY_PATH
- ldconfig
- nm
- gcc -shared
- info binutils

nm

- list library symbols
- #!/bin/bash
  a=`ls /usr/lib/*.a /lib/*.a /usr/X11R6/lib/*.a /usr/local/lib/*.a`
  for file in $a
  do
   p=`nm -A -g -f p --defined-only $file 2>/dev/null | grep " $1 "'
  if ((! $?))
   then
     echo $p
   fi
  done
  a=`ls /usr/lib/*.so /lib/*.so /usr/X11R6/lib/*.so /usr/local/lib/*.so`
  for file in $a
  do
   p=`nm -D -A -g -f p --defined-only $file  2>/dev/null | grep " $1 "`
   if ((! $?))
   then
     echo $p
   fi
  done