KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Information and Computer Science Department
ICS 431 Operating Systems
Lab # 13

Inter-Process Communication: Pipes

Objective:

Now that we know how to create processes, let us turn our attention to make the processes communicate among themselves. There are many mechanisms through which the processes communicate and in this lab we will discuss two such mechanisms: Pipes and Signals. A pipe *is used for one-way communication of a stream of bytes*. Signals *inform processes of the occurrence of asynchronous events*. In this lab we will learn how to create pipes and how processes communicate by reading or writing to the pipe and also how to have a two-way communication between processes. This lab also discusses how user-defined handlers for particular signals can replace the default signals handlers and also how the processes can ignore the signals.

By learning about signals, you can "protect" your programs from *Control-C*, arrange for an alarm clock signal to terminate your program if it takes too long to perform a task, and learn how UNIX uses signals during everyday operations.

The system calls covered in this lab are:

- Pipe
- dup / dup2
- Alarm
- Kill
- Signal

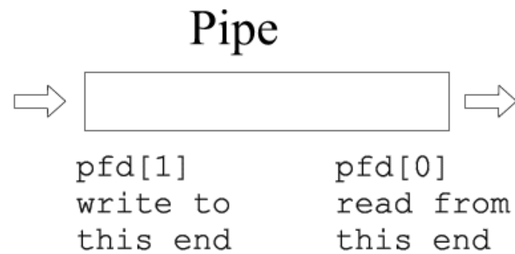
Pipes:

Pipes are familiar to most Unix users as a shell facility. For instance, to print a sorted list of who is logged on, you can enter this command line:
who | sort | lpr

There are three processes here, connected with two pipes. Data flows in one direction only, from **`who`** to **`sort`** to **`lpr`**. It is also possible to set up bidirectional pipelines (from process A to B, and from B back to A) and pipelines in a ring (from A to B to C to A) using system calls. The shell, however, provides no notation for these more elaborate arrangements, so they are unknown to most Unix users. We'll begin by showing some simple examples of processes connected by a one-directional pipeline.

pipe System Call :

```
 int pfd[2];
int pipe (pfd);  /* Returns 0 on success or -1 on error */
```



I/O with a pipe:

These two file descriptors can be used for block I/O

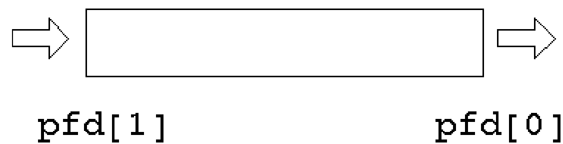
```
write(pfd[1], buf, SIZE);

read(pfd[0], buf, SIZE);
```

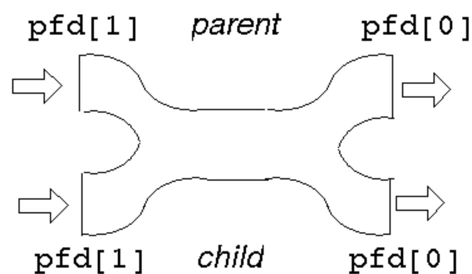
Fork and a pipe:

A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using fork ( ). A pipe opened before the fork becomes shared between the two processes.

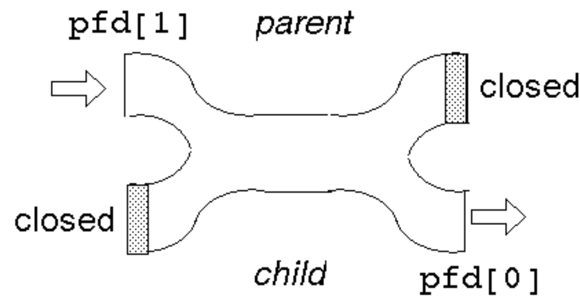
Before fork



After fork



This gives *two* read ends and *two* write ends. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.
Either process can write into the pipe, and either can read from it. Which process will get what is not known.
For predictable behavior, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.



Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end.

When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end.

Pipes use the buffer cache just as ordinary files do. Therefore, the benefits of writing and reading pipes in units of a block (usually 512 bytes) are just as great. A single **write** execution is atomic, so if 512 bytes are written with a single system call, the corresponding read will return with 512 bytes (if it requests that many). It will not return with less than the full block. However, if the writer is not writing complete blocks, but the reader is trying to read complete blocks, the reader may keep getting partial blocks anyhow. This won't happen if the writer is faster than the reader, since then the writer will be able to fill the pipe with a complete block before the reader gets around to reading anything. Still, it's best to buffer writes and reads on pipes, and this is what the Standard I/O Library does automatically.

```
#include <stdio.h>

#define SIZE 1024

main( )
{
 int pfd[2];
 int nread;
 int pid;
 char buf[SIZE];

 if (pipe(pfd) == -1)
 {
  perror("pipe failed");
  exit(1);
 }
 if ((pid = fork()) < 0)
 {
  perror("fork failed");
  exit(2);
 }

 if (pid == 0)
 {
```

```
  /* child */
  close(pfd[1]);
  while ((nread = read(pfd[0], buf, SIZE)) != 0)
    printf("child read %s\n", buf);
  close(pfd[0]);
 }
 else
 {
  /* parent */
  close(pfd[0]);
  strcpy(buf, "hello...");
  /* include null terminator in write */
  write(pfd[1], buf, strlen(buf)+1);
  close(pfd[1]);
 }
}
```

Given that we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't. Once the processes are created they can't be connected, because there's no way for the process that makes the pipe to pass a file descriptor to the other process. It can pass the file descriptor number, of course, but that number won't be valid in the other process. But if we make a pipe in one process *before creating* the other process, it will inherit the pipe file descriptors, and they will be valid in both processes. Thus, two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on, but they must be related, and the pipe must be passed on at birth. In practice, this may be a severe limitation, because if a process dies there's no way to recreate it and reconnect it to its pipes -- the survivors must be killed too, and then the whole family has to be recreated.

In general, then, here is how to connect two processes with a pipe:

Make the pipe.
Fork to create the reading child.
In the child close the writing end of the pipe, and do any other preparations that are needed.
In the child execute the child program.
In the parent close the reading end of the pipe.
If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.

Here's a small program that uses a pipe to allow the parent to read a message from its child:

```
#include <stdio.h>
#include <string.h>

#define  READ    0
#define  WRITE   1

char*  phrase  =  "This is ICS431 lab time" ;
```

```
main ( )
{
 int     fd [2],  bytesread ;
 char    message [100] ;

 pipe ( fd ) ;
 if ( fork (  )  ==  0 )                              /*  child, writer */
 {
    close ( fd [READ] ) ;                             /*  close unused end */
    write  ( fd [WRITE], phrase, strlen (phrase) + 1) ;
    close ( fd [WRITE] ) ;                            /*  close used end  */
 }
 else                                                 /*  parent,  reader  */
 {
    close ( fd [WRITE] ) ;                /* close unused end  */
    bytesread = read (fd [READ], message, 100) ;
    printf ("Read %d bytes : %s\n", bytesread, message) ;
    close ( fd [READ] ) ;                             /*  close used end  */
 }
}
```

Implementation of Redirection:

When a process forks, the child inherits a copy of its parent's file descriptors. When process execs, the standard input, output, and error channels remain unaffected. The UNIX shell uses these two pieces of information to implement redirection.

To perform redirection, the shell performs the following series of actions:

- The parent shell forks and then waits for the child shell to terminate.
- The child shell opens the file "output", creating it or truncating as necessary.
- The child shell then duplicates the file descriptor of "output" to the standard output file descriptor, number 1, and then closes the original descriptor of "output". All standard output is therefore redirected to "output".
- The child shell then exec's the ls utility. Since the file descriptors are inherited during an exec ( ), all of standard output of ls goes to "output".
- When the child shell terminates, the parent resumes. The parent's file descriptors are unaffected by the child's actions, as each process maintains its own private descriptor table.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/file.h>
main (argc, argv)
int  argc ;
char  *argv[ ] ;
{
   int   fd ;       /* file descriptor or pointer */
```

```
    fd = open (argv[1], O_CREAT | O_TRUNC | O_RDWR, 0777) ;
                              /* open file named in argv[1] */
    dup2 (fd, 1) ;           /*  and assign it to fd file pointer */
    close (fd) ;  /* duplicate fd with 1 which is standard output (the monitor) */
    execvp (argv[2], &argv[2]) ;
                 /* the output is not printed on screen but is redirected to "output" file */
    printf ("End\n") ;    /* should never execute */
}
```

dup / dup2 System Call:

```
int dup   (int  oldfd )
int dup2 (int  oldfd, int newfd )
```

dup ( ) finds the smallest free file descriptor entry and points it to the same file as *oldfd*. dup2 ( ) closes *newfd* if it's currently active and then points it to the same file as *oldfd*. In both cases, the original and copied file descriptors share the same file pointer and access mode.

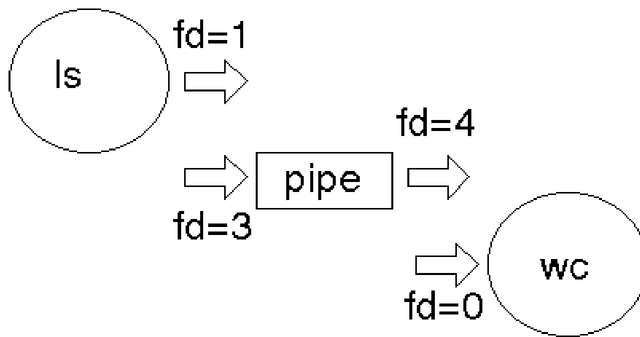
They both return the index of the new file descriptor if successful, and –1 otherwise.
dup/dup2 duplicates an existing file descriptor, giving a new file descriptor that is open to the same file or pipe. The two share the same file pointer, just as an inherited file descriptor shares the file pointer with the corresponding file descriptor in the parent. The call fails if the argument is bad (not open) or if 20 file descriptors are already open.

A pipeline works because the two processes know the file descriptor of each end of the pipe. Each process has a stdin (0), a stdout (1) and a stderr (2). The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.

Suppose one of the processes replaces itself by an "exec". The new process will have files for descriptors 0, 1, 2, 3 and 4 open. How will it know which are the ones belonging to the pipe? It can't.

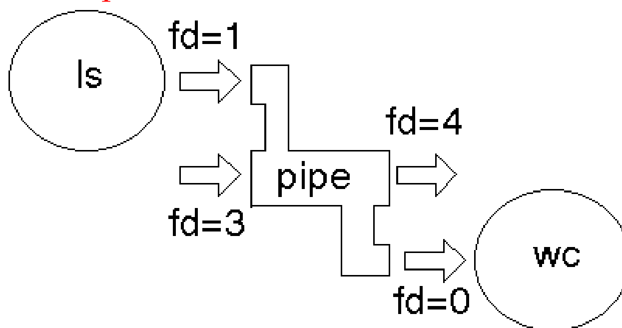
Example:

To implement "ls | wc" the shell will have created a pipe and then forked. The parent will exec to be replaced by "ls", and the child will exec to be replaced by "wc" The write end of the pipe may be descriptor 3 and the read end may be descriptor 4. "ls" normally writes to 1 and "wc" normally reads from 0. How do these get matched up?

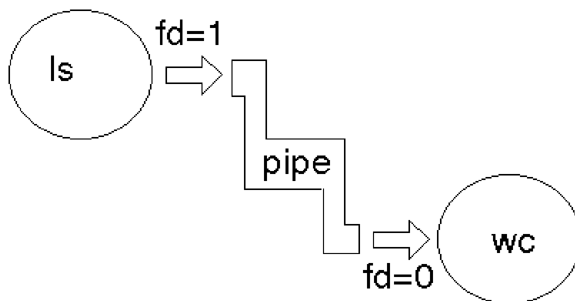


The dup/dup2 function call takes an existing file descriptor, and another one that it "would like to be". Here, fd=3 would also like to be 1, and fd=4 would like to be 0. So we *dup* fd=3 as 1, and *dup* fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

After dup



After close



The UNIX shells use unnamed pipes to build pipelines. They use a trick similar to the redirection mechanism to connect the standard output of one process to standard input of another. To illustrate this approach, here's the program that executes two named programs, connecting the standard output of the first to the standard input of the second.

```
#include <stdio.h>
#include <string.h>

#define  READ    0
#define  WRITE   1
```

```
main (argc, argv)
int  argc ;
char*  argv[] ;
{
 int     pid, fd [2] ;

 if (pipe(fd) == -1)
 {
   perror("pipe failed");
   exit(1);
 }
 if ((pid = fork( )) < 0)
 {
   perror("fork failed");
   exit(2);
 }
 if ( pid  !=  0 )                          /*  parent, writer */
 {
    close ( fd [READ] ) ;              /*  close unused end */
    dup2 ( fd [WRITE], 1) ;         /* duplicate used end to standard out */
    close ( fd [WRITE] ) ;                /*  close used end  */
    execlp ( argv[1], argv[1], NULL) ;       /*  execute writer program  */
 }
 else                                  /*  child,  reader  */
 {
    close ( fd [WRITE] ) ;                /* close unused end */
    dup2 ( fd [READ], 0) ;              /* duplicate used end to standard input */
    close ( fd [READ] ) ;               /* close used end */
    execlp ( argv[2], argv[2], NULL) ;        /* execute reader program */
 }
}
```

Run the above program as

a.out who wc

Variations:

Some common variations on this method of IPC are:

A pipeline may consist of three or more process (such as a C version of `ps | sed 1d | wc -l`). In this case there are lots of choices The parent can fork twice to give two children.
The parent can fork once and the child can fork once, giving a parent, child and grandchild.
The parent can create two pipes before any forking. After a fork there will then be a total of 8 ends open (2 processes * two ends * 2 pipes). Most of these will have to be closed to ensure that their ends up only one read and only one write end.

As many ends as possible of a pipe may be closed before a fork. This minimizes the number of closes that have to be done after forking.
A process may want to both write to and read from a child. In this case it creates two pipes. One of these is used by the parent for writing and by the child for reading. The child for writing and the parent for reading use the other.

Exercises

Note:

Lab Problems will be given during the lab based on material covered in this lab manual