Micro Assignment - 03

Introduction to Pipes

Consider a command like the following, where the output of a command is sent for further processing to another command. Shell creates two processes. The

```
$ ls | wc -l
```

first process executes the ls and the second the wc. The STDOUT of the first process feeds a pipe. The STDIN of the second process is fed by the same pipe. This implements the desired data flow to implement the above command chain.

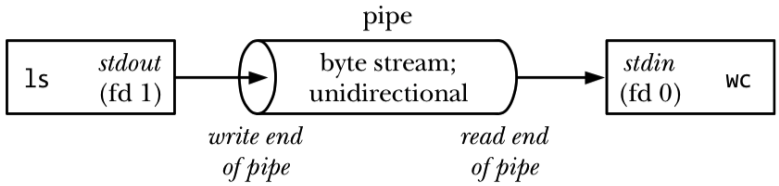

Figure 1: Using a pipe to connect two processes.

Pipe is a byte Stream - There is no notion of messages or message boundaries. There is no random access on a pipe. Use the pipe() system call to create a pipe.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Returns 0 on success, or −1 on error

Reading from an empty pipe will block until at least one byte is there.

If the write end of the pipe is closed, then the read end will return with EOF.

Pipes are unidirectional – one end is used for writing and other end is used for reading.

Pipes have limited capacity - normally 65536 bytes

On success, two file descriptors are available to the calling process. filedes[1] is the descriptor for writing and filedes[0] is the descriptor for reading.

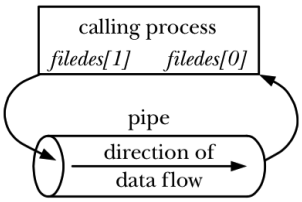

Figure 2: Process file descriptors after creating a pipe.

Piping Across Processes

Pipes have very little purpose within a process. They are primarily used for transfer of data across two processes – a parent and a child processes. To setup this interconnection among the two processes, we create the pipe and then fork(). The fork() creates the child process. The child process inherits all including

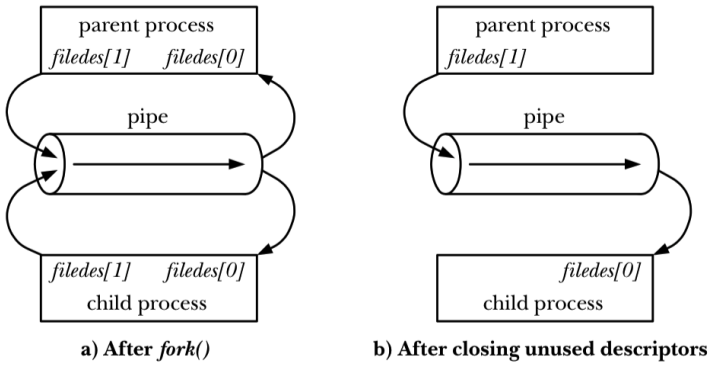

Figure 3: Setting up a pipe to transfer data from a parent to a child

system variables from the parent. This means the reference to the newly created pipe will be copied to the child as well.

Normally, pipes are created for sending data from the parent to the child. Other forms of data transfer are possible as well.

Because data goes from parent to child, parent closes the read (filedes[0]) and child closes write (filedes[1]) descriptors. After closing the configuration looks like the one shown in (b) in the Figure above.

NOTE: Communication using pipes is NOT restricted to parent-child. It can be used for communication between "related" processes.

It can be used for communication between a process and its grandchild. In this case, the process creates the pipe. Its child passes it along to the grandchild.

Another possibility is between two siblings. In this case, a process creates a pipe and creates two children, who use the pipe to communicate between them. This is the pattern used by shells.

ASIDE: A Unix process at startup has the file descriptors open as shown in Figure 4.

File descriptor	**Purpose**	**POSIX name**	***stdio* stream**
0	standard input	`STDIN_FILENO`	*stdin*
1	standard output	`STDOUT_FILENO`	*stdout*
2	standard error	`STDERR_FILENO`	*stderr*

Figure 4: Standard file descriptors

Pseudo-Code for Piped Communications

A process that wants to use pipe for communicating with a chid needs to do the following:

1. Create the pipe using the pipe() system call.
2. Create the child using the fork() system call.
3. Parent closes the read descriptor.
4. Child closes the write descriptor (assuming that the parent is the writer and the child reader).
5. Parent writes to filedes[1] and child reads from filedes[0].

There is one problem with the above pseudo-code. The parent and child need to know the file descriptors allocated to the pipe. This would not work if the parent and child are programs that already exist. To solve this problem, we need to use the dup2() system call. Using this system call we can rewire a well known file descriptor such as STDOUT (1) and STDIN (0) to the input and output of the pipe, respectively.

The figure below shows an example program that follows the above pseudo-code.

```
#include <unistd.h>

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

Returns (new) file descriptor on success, or −1 on error

```
int
main(int argc, char *argv[])
{
    int pfd[2];                             /* Pipe file descriptors */
    char buf[BUF_SIZE];
    ssize_t numRead;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s string\n", argv[0]);

    if (pipe(pfd) == -1)                    /* Create the pipe */
        errExit("pipe");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* Child  - reads from pipe */
        if (close(pfd[1]) == -1)            /* Write end is unused */
            errExit("close - child");

        for (;;) {              /* Read data from pipe, echo on stdout */
            numRead = read(pfd[0], buf, BUF_SIZE);
            if (numRead == -1)
                errExit("read");
            if (numRead == 0)
                break;                      /* End-of-file */
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("child - partial/failed write");
        }

        write(STDOUT_FILENO, "\n", 1);
        if (close(pfd[0]) == -1)
            errExit("close");
        _exit(EXIT_SUCCESS);

    default:            /* Parent - writes to pipe */
        if (close(pfd[0]) == -1)            /* Read end is unused */
            errExit("close - parent");
```

What Do You Need To Do?

Study this HANDOUT carefully. Understand the usage patterns of pipes in UNIX/Linux.

```
        if (write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
            fatal("parent - partial/failed write");

        if (close(pfd[1]) == -1)            /* Child will see EOF */
            errExit("close");
        wait(NULL);                         /* Wait for child to finish */
        exit(EXIT_SUCCESS);
    }
}
```

Figure 5: Example program using pipes for intercommunication.

Write a program to implement the following command pipeline. One process

```
$ ls | wc -l
```

should execute the "ls" command. Another process executes the "wc -l" command. Use a pipe to funnel the output of the ls command to the input of the process running the "wc -l" command. Use the ideas discussed in this handout along with concepts you have learned in the previous assignments to complete this assignment.