STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Assignment No: 13_b

Subject:- Unix Operating System

System Lab Class :- TYIT

Name PRN

Aditi Sudhir Ghate 2020BTEIT00044

Title:

Filter to convert uppercase characters to lowercase.

Objectives:

1. To learn about STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Theory:

Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations:

1. Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the pipe function.

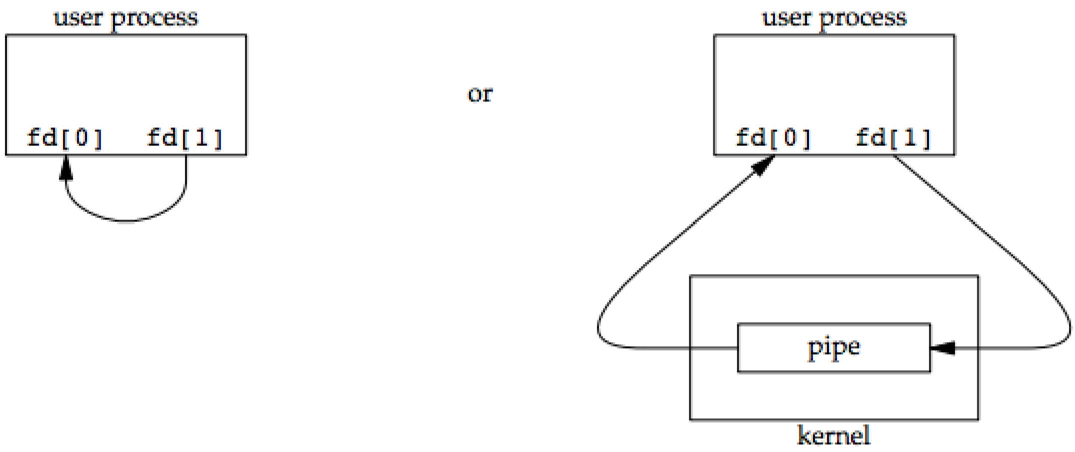
```
#include <unistd.h>
```

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


/* Returns: 0 if OK, −1 on error */
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

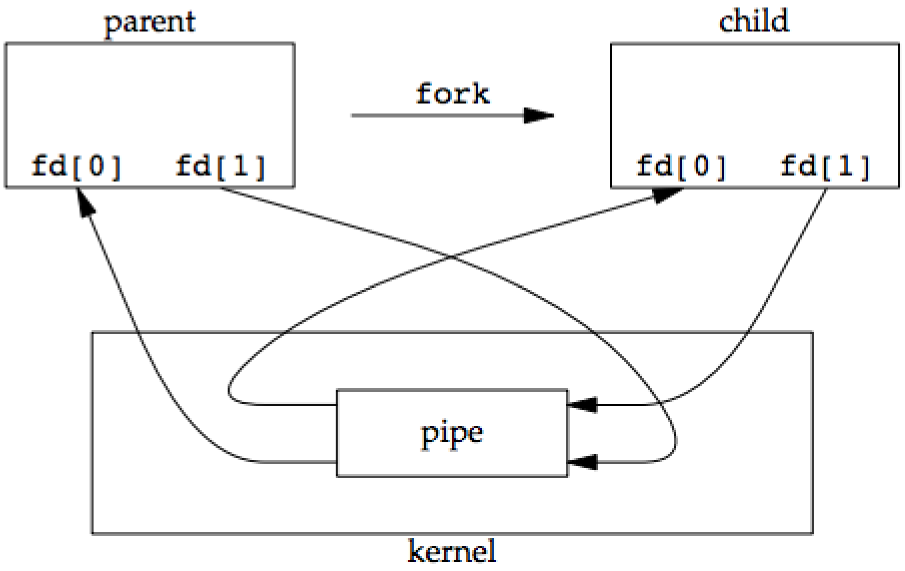
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.



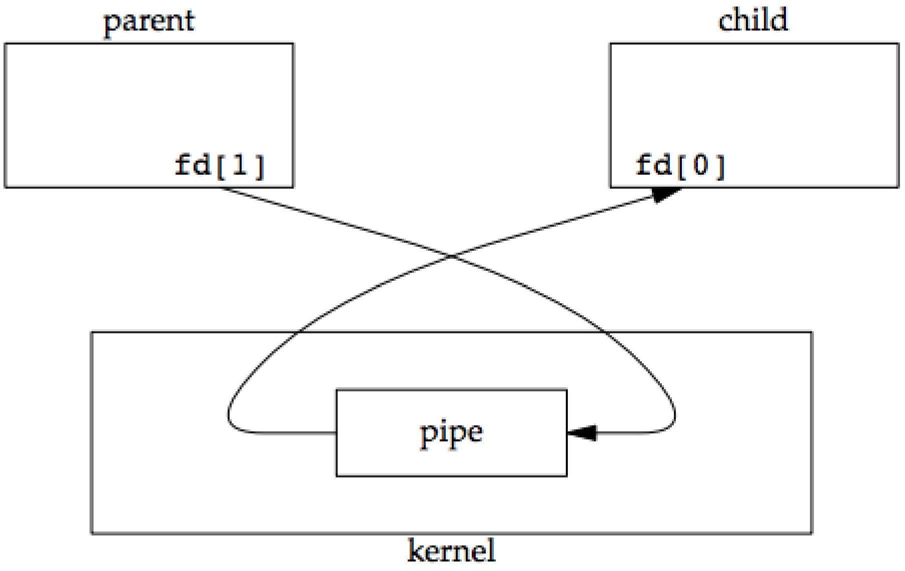
The fstat function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the S_ISFIFO macro.

POSIX.1 states that the st_size member of the stat structure is undefined for pipes. But when the fstat function is applied to the file descriptor for the read end of the pipe, many systems store in st_size the number of bytes available for reading in the pipe, which is nonportable.

A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa. The following figure shows this scenario:



What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (*fd[0]*), and the child closes the write end (*fd[1]*). The following figure shows the resulting arrangement of descriptors.



For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

When one end of a pipe is closed, two rules apply:

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
   - Technically, we should say that this end of file is not generated until there are no more writers for the pipe.
   - It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.
   - Normally, there is a single reader and a single writer for a pipe. (The FIFOs in the next section dicusses that there are multiple writers for a single FIFO.)

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

When we’re writing to a pipe (or FIFO), the constant PIPE_BUF specifies the kernel’s pipe buffer size. A write of PIPE_BUF bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than PIPE_BUF bytes, the data might be interleaved with the data from the other writers. We can determine the value of PIPE_BUF by using pathconf or fpathconf.

Example: creating a pipe between a parent and its child

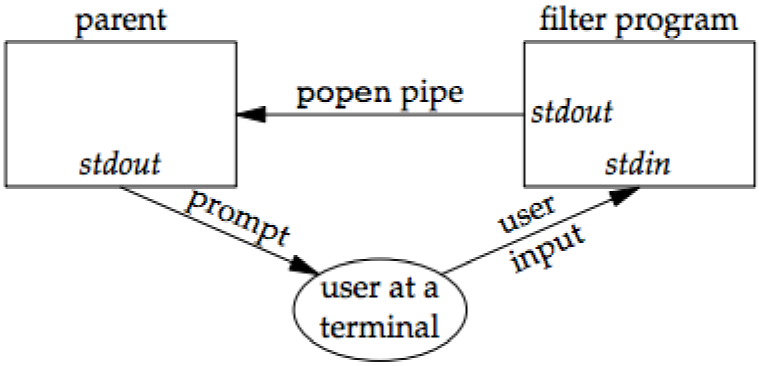
ipc1/pipe1.c

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Example: transforming input using popen

One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Consider an application that writes a prompt to standard output and reads a line from standard input. With the popen function, we can interpose a program between the application and its input to transform the input. The following figure shows the arrangement of processes in this situation.



The following program is a simple filter to demonstrate this operation:

Code:

```
//file apue.h

/*
 * Our own header, to be included before all standard system headers.
 */
#ifndef _APUE_H
#define _APUE_H

#define _POSIX_C_SOURCE 200809L

#if defined(SOLARIS)              /* Solaris 10 */
#define _XOPEN_SOURCE 600
#else
#define _XOPEN_SOURCE 700
#endif
```

```
#include <sys/types.h>          /* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h>        /* for winsize */
#if defined(MACOS) || !defined(TIOCGWINSZ)
#include <sys/ioctl.h>
#endif



#include <stdio.h>              /* for convenience */
#include <stdlib.h>             /* for convenience */
#include <stddef.h>             /* for offsetof */
#include <string.h>             /* for convenience */
#include <unistd.h>             /* for convenience */
#include <signal.h>             /* for SIG_ERR */

#define MAXLINE     4096                    /* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void    Sigfunc(int);   /* for signal handlers */

#define min(a,b)    ((a) < (b) ? (a) : (b))
#define max(a,b)    ((a) > (b) ? (a) : (b))

/*
 * Prototypes for our own functions.
 */
char      *path_alloc(size_t *);                        /* {Prog pathalloc} */
long       open_max(void);                                      /* {Prog openmax} */

int                  set_cloexec(int);                                  /* {Prog setfd} */
void       clr_fl(int, int);
void       set_fl(int, int);                            /* {Prog setfl} */

void       pr_exit(int);                                        /* {Prog prexit} */

void       pr_mask(const char *);                       /* {Prog prmask} */
Sigfunc *signal_intr(int, Sigfunc *);           /* {Prog signal_intr_function} */

void       daemonize(const char *);                     /* {Prog daemoninit} */

void       sleep_us(unsigned int);              /* {Ex sleepus} */
ssize_t   readn(int, void *, size_t);           /* {Prog readn_writen} */
ssize_t   writen(int, const void *, size_t);/* {Prog readn_writen} */

int                  fd_pipe(int *);                                    /* {Prog sock_fdpipe} */
int                  recv_fd(int, ssize_t (*func)(int,
                            const void *, size_t));     /* {Prog recvfd_sockets} */
int                  send_fd(int, int);                                         /* {Prog sendfd_sockets} */
int                  send_err(int, int,
                             const char *);                     /* {Prog senderr} */
int                  serv_listen(const char *);                         /* {Prog servlisten_sockets} */
int                  serv_accept(int, uid_t *);                         /* {Prog servaccept_sockets} */
int                  cli_conn(const char *);                    /* {Prog cliconn_sockets} */
int                  buf_args(char *, int (*func)(int,
                             char **));                         /* {Prog bufargs} */

int                  tty_cbreak(int);                                   /* {Prog raw} */
int                  tty_raw(int);                                              /* {Prog raw} */
int                  tty_reset(int);                                    /* {Prog raw} */
void       tty_atexit(void);                                    /* {Prog raw} */
```

```
struct termios    *tty_termios(void);                          /* {Prog raw} */

int                ptym_open(char *, int);                     /* {Prog ptyopen} */
int                ptys_open(char *);                                       /* {Prog ptyopen} */
#ifdef   TIOCGWINSZ
pid_t     pty_fork(int *, char *, int, const struct termios *,
                           const struct winsize *);     /* {Prog ptyfork} */
#endif

int                lock_reg(int, int, int, off_t, int, off_t); /* {Prog lockreg} */

#define read_lock(fd, offset, whence, len) \
                             lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
                             lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
                             lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
                             lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
                             lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t    lock_test(int, int, off_t, int, off_t);                  /* {Prog locktest} */

#define is_read_lockable(fd, offset, whence, len) \
                             (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
                             (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void     err_msg(const char *, ...);                              /* {App misc_source} */
void     err_dump(const char *, ...) __attribute__((noreturn));
void     err_quit(const char *, ...) __attribute__((noreturn));
void     err_cont(int, const char *, ...);
void     err_exit(int, const char *, ...) __attribute__((noreturn));
void     err_ret(const char *, ...);
```

```
void	err_sys(const char *, ...) __attribute__((noreturn));

void	log_msg(const char *, ...);				/* {App misc_source} */
void	log_open(const char *, int, int);
void	log_quit(const char *, ...) __attribute__((noreturn));
void	log_ret(const char *, ...);
void	log_sys(const char *, ...) __attribute__((noreturn));
void	log_exit(int, const char *, ...) __attribute__((noreturn));

void	TELL_WAIT(void);		/* parent/child from {Sec race_conditions} */
void	TELL_PARENT(pid_t);
void	TELL_CHILD(pid_t);
void	WAIT_PARENT(void);
void	WAIT_CHILD(void);

#endif	/* _APUE_H */
```

//file myuclc.c

```
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we’re careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

We compile this filter into the executable file `myuclc`, which we then invoke from the program in the following code using `popen`:

//file popen1.c

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;

    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

Conclusion:

The concept of pipe has been learned. File has converted from lowercase to upper case using filter.

References:

- https://notes.shichao.io/apue/ch15/