**STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions**

**Assignment No: 13\_b**

**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>

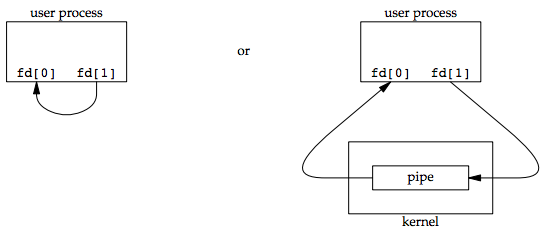
int pipe(int fd[2]);

/\* 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.

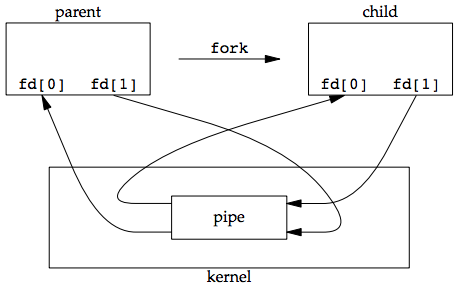
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

[](https://notes.shichao.io/apue/figure_15.2.png)

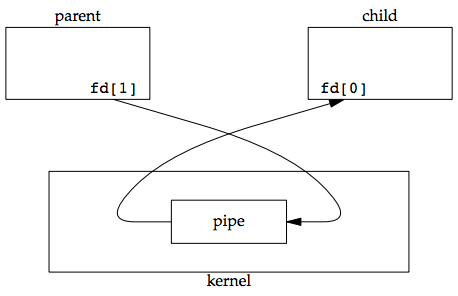
The fstat function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the S\_ISFIFO macro.

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:

[](https://notes.shichao.io/apue/figure_15.3.png)

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.

[](https://notes.shichao.io/apue/figure_15.4.png)

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](https://github.com/shichao-an/apue.3e/blob/master/)

#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);

}

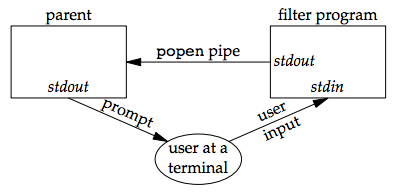
exit(0);

}

#### 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.

[](https://notes.shichao.io/apue/figure_15.13.png)

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/>