C Tutorial: Pipes

How to use a pipe

A pipe is a system call that creates a unidirectional communication link between two file descriptors. The pipe system call is called with a pointer to an array of two integers. Upon return, the first element of the array contains the file descriptor that corresponds to the output of the pipe (stuff to be read). The second element of the array contains the file descriptor that corresponds to the input of the pipe (the place where you write stuff). Whatever bytes are sent into the input of the pipe can be read from the other end of the pipe.

Example

This is a small program that gives an example of how a pipe works. The array of two file descriptors is fd[2]. Whatever is written to fd[1] will be read from fd[0].

```
/*
      The simplest example of pipe
*/

#include <stdlib.h>
#include <stdio.h>      /* for printf */
#include <string.h>     /* for strlen */

int
main(int argc, char **argv)
{
      int n;
      int fd[2];
      char buf[1025];
      char *data = "hello... this is sample data";

      pipe(fd);
      write(fd[1], data, strlen(data));
      if ((n = read(fd[0], buf, 1024)) >= 0) {
            buf[n] = 0; /* terminate the string */
            printf("read %d bytes from the pipe: \"%s\"\n", n, buf);
      }
      else
            perror("read");
      exit(0);
}
```

Save this file by control-clicking or right clicking the download link and then saving it as pipe-simple.c.

Compile this program via:

```
gcc -o pipe-simple pipe-simple.c
```

If you don't have gcc, You may need to substitute the gcc command with cc or another name of your compiler.

Run the program:

```
./pipe-simple
```

Pipes between processes

This next example shows the true value of a pipe. We create a pipe between the "/bin/ls -al /" command and the "/usr/bin/tr a-z A-Z" command. This is the equivalent of running the shell command:

```
/bin/ls -al / | /usr/bin/tr a-z A-Z
```

The first command generates a long-format directory listing of the root (/) directory and the second command takes that listing and translates all lowercase characters to uppercase.

We start off by creating a pipe. Then we fork a child process. The parent will use the pipe for command output. That means it needs to change its standard output file descriptor (1) to the writing end of the pipe (pfd[1]). It does this via the dup2 system call: dup2(pfd[1], 1). Then it executes the command in cmd1.

The child will use the pipe for command input. It needs to change its standard input file descriptor (0) to the reading end of the pipe (pfd[0]). It also does this via the dup2 system call: dup2(pfd[0], 0). Then it executes the command in cmd2.

```
/* pipe demo */


#include <stdlib.h>
#include <stdio.h>

void runpipe();

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

      pipe(fd);

      switch (pid = fork()) {

      case 0: /* child */
            runpipe(fd);
            exit(0);

      default: /* parent */
            while ((pid = wait(&status)) != -1)
                  fprintf(stderr, "process %d exits with %d\n", pid,
WEXITSTATUS(status));
            break;

      case -1:
            perror("fork");
            exit(1);
      }
      exit(0);
}

char *cmd1[] = { "/bin/ls", "-al", "/", 0 };
char *cmd2[] = { "/usr/bin/tr", "a-z", "A-Z", 0 };

void
runpipe(int pfd[])
{
      int pid;

      switch (pid = fork()) {

      case 0: /* child */
            dup2(pfd[0], 0);
            close(pfd[1]);    /* the child does not need this end of the pipe */
            execvp(cmd2[0], cmd2);
            perror(cmd2[0]);

      default: /* parent */
            dup2(pfd[1], 1);
            close(pfd[0]);    /* the parent does not need this end of the pipe
*/
            execvp(cmd1[0], cmd1);
            perror(cmd1[0]);

      case -1:
            perror("fork");
            exit(1);
      }
}
```

Save this file by control-clicking or right clicking the download link and then saving it as pipe-exec.c.

Compile this program via:

```
gcc -o pipe-exec pipe-exec.c
```

If you don't have gcc, You may need to substitute the gcc command with cc or another name of your compiler.

Run the program:

```
./pipe-exec
```

Creating a pipe between two child processes

The above example put the parent process into a state where it gave up its standard output to the pipe and the process itself was replaced by the exec of cmd1. If we want to preserve the parent program and its input and output streams but run the pipe between two child processes, we need to fork off two children.

The child that generates output will set its standard output to the writing end of the pipe and the child that consumes that data will set its standard input to the reading end of the same pipe. Once this is done the parent no longer needs the pipe and can close its file descriptors. This is important! If the parent does not close the writing end of the pipe (pfd[1]) then the child that is reading from the pipe will never read an end of file and will never exit.

```
/*
      Sample progam illustrating the use of Unix pipes between processes.


      This forks and execs two commands (cmd1 and cmd2), with the standard
output of
      cmd1 going to the standard input of cmd2.

      The commands are:
      cmd1:
            /bin/ls -al /
            lists all the files in the root directory in the long format
      cmd2:
            /usr/bin/tr a-z A-Z
            translates all input from lowercase to uppercase
      Feel free to modify cmd1 and cmd2 to get them to work on your system or to
do something differnt.

      Key points:
```

0. Processes start with three open file descriptors. File descriptor 0 is the standard input and is typically the keyboard input. File descriptor 1 is the standard output and is typically the virtual terminal that is the window where the shell is running. File descriptor 2 is the standard error and is typically the same as the standard output. If the standard output is redirected to a file or another command, errors can still be sent to the screen where a user can see them.

1. In main(), we use the pipe() system call to create a communication pipe.
A pipe gives us a set of two file descriptors, fd[0] and fd[1].
Anything written to fd[1] can be read from fd[0].
Pipes are unidirectional.

2. We call runsource() and rundest(), which run cmd1 and cmd2 respectively.
Each of these functions forks a child and execs the command.

3. We close both file descriptors of the pipe since we don't need them anymore in the parent process. This is an important step. Otherwise, the child that is reading from the pipe will never detect and end of file since the other end of the pipe will remain open even if the child that was writing into it terminated.

4. We now wait for all child processes to exit. In this demo, we print the exit code of the process. This is the number supplied as an argument to the exit() call. For example, exit(5) will cause a process to exit with the exit code 5. The convention for Unix commands is that an exit code of 0 means that the command executed successfully.

5. In runsource(), we fork() to create a child process. The child process will run cmd1.
First, we need to change the standard output of the process to the writing end of the pipe (fd[1]).

We use the dup2() system call to duplicate the writing file descriptor of the pipe (pfd[1]) onto the standard output file descriptor, 1.

We don't need the input end of the pipe (pdf[0]), so we close it.

6. Once that is done, we simple call execvp() to run the program. This program will overwrite our process' memory. If it fails, then we call the perror() function to print an error message.

7. rundest() is the receiving command of our pipeline and is similar to runsource().
Here, we need to change the standard input of the process to the reading end of the pipe (fd[0]).

We use the dup2() system call to duplicate the reading file descriptor of the pipe (pfd[0]) onto the standard input file descriptor, 0.

We don't need the output end of the pipe (pdf[1]), so we close it.

```
      8. In both runsource() and rundest(), the parent process simply returns
back to main().

*/


#include <stdlib.h>
#include <stdio.h>

char *cmd1[] = { "/bin/ls", "-al", "/", 0 };
char *cmd2[] = { "/usr/bin/tr", "a-z", "A-Z", 0 };

void runsource(int pfd[]);
void rundest(int pfd[]);

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

      pipe(fd);

      runsource(fd);
      rundest(fd);
      close(fd[0]); close(fd[1]);   /* this is important! close both file
descriptors on the pipe */

      while ((pid = wait(&status)) != -1) /* pick up all the dead children */
            fprintf(stderr, "process %d exits with %d\n", pid,
WEXITSTATUS(status));
      exit(0);
}


void
runsource(int pfd[])    /* run the first part of the pipeline, cmd1 */
{
      int pid;    /* we don't use the process ID here, but you may wnat to print
it for debugging */

      switch (pid = fork()) {

      case 0: /* child */
            dup2(pfd[1], 1);  /* this end of the pipe becomes the standard
output */
            close(pfd[0]);          /* this process don't need the other end */
            execvp(cmd1[0], cmd1);  /* run the command */
            perror(cmd1[0]);  /* it failed! */

      default: /* parent does nothing */
            break;

      case -1:
            perror("fork");
            exit(1);
      }
}

void
rundest(int pfd[])      /* run the second part of the pipeline, cmd2 */
{
      int pid;
      switch (pid = fork()) {

      case 0: /* child */
            dup2(pfd[0], 0);  /* this end of the pipe becomes the standard input
*/
            close(pfd[1]);          /* this process doesn't need the other end
*/
            execvp(cmd2[0], cmd2);  /* run the command */
            perror(cmd2[0]);  /* it failed! */

      default: /* parent does nothing */
            break;

      case -1:
            perror("fork");
            exit(1);
      }
}
```

Download this file

Save this file by control-clicking or right clicking the download link and then saving it as pipedemo.c.

Compile this program via:

```
gcc -o pipedemo pipedemo.c
```

If you don't have gcc, You may need to substitute the gcc command with cc or another name of your compiler.

Run the program:

```
./pipedemo
```