Unix: How can I write a code for PIPE (|) in C/ shell script/python?

in unix we do something like
user@user$ cat file | grep ..... More

Costya Perepelitsa
8.3k Views • Upvoted by Shrey Banga (श्रेय), Platform Engineer at Quora
Costya has 30+ answers in C (programming language).

This is much, *much* easier to do in Python than in C, primarily because Python comes with a few libraries that do all of this for you; you have to do all the gory details yourself if you use C.

Python

There are a few modules you can use to start a subprocess and communicate with it, but one of the most convenient is, unsurprisingly, the subprocess module.

Use `subprocess.Popen()` to start the subprocess and return a handle to it. Use the handle to send it input, wait for it to finish, and get its output.

Here's an example REPL session that uses **grep** to filter some input:

```
>>> import subprocess
>>> grep = subprocess.Popen(["grep", "-n", "^ba", "-"],
>>> grep.stdin.write("foo\nbar\nbaz\nqux\n")
>>> grep.stdin.close()
>>> grep.wait()
0
>>> grep.stderr.read()
''
>>> grep.stdout.readlines()
['2:bar\n', '3:baz\n']
```

C

This task requires intimate knowledge of UNIX file descriptors and processes.

First, here are the system calls used to accomplish what you're asking:

- **close:** closes a file descriptor; processes reading from it will be notified that there is no more data to be read ("end of file")
- **fork:** spawns an identical process as the calling program, which runs independently of the calling process, continuing execution from the `fork` call; the calling process is the "parent" and the new copy is the "child"; one important detail of `fork` is that each process' file descriptors refer to the *same* file descriptors
- **exec:** starts the given program and then makes the calling process *become* it, using the calling process' standard input, standard output, and standard error exactly as the calling process left them; after calling `exec`, the calling process is no more -- it finishes execution as the program given as the argument
- **waitpid:** waits for a child process to finish
- **pipe:** creates two file descriptors: one for the read end of the pipe and one for the write end; data written to the write file descriptor can be read from the read file descriptor (duh)
- **dup2:** clobbers a file descriptor with another; useful for redirecting a process' standard input, standard output, and standard error to/from pipes instead of the terminal

The method goes something like this:

1. Create pipes to interface with the process you want to call, for whichever of standard input, standard output, and standard error you need to use.
2. Fork.

The child process:

1. Close the write end of the standard input pipe (if such a pipe was made) and the read ends of the standard output and standard error pipes (if they've been made); these are for the parent's use.
2. Use `dup2` to force the child to use the pipes as standard input, output, and error. For example, after doing this, anything the child tries to write to standard output (e.g. `printf` statements) will go to the write end of the standard output pipe instead.
3. Finally, have the child use `exec` to become the program you want to run. All the changes you've made to the child's standard input, output, and error will carry over, forcing the program to use those pipes.

The parent process:

1. Close the read end of the standard input pipe and the write ends of the

end of the standard input pipe. If the parent uses the write end of the standard input pipe, it should close it when it finishes to signify to the child process that there is no more data.

3. Wait for the child to terminate.
4. Read whatever output data you need from the pipes the child wrote to.

The following is a very simple, error-prone program which calls another program and obtains its output. It takes the program to be called and any arguments it should be given as its own arguments.
For the sake of clarity, I've skipped over a lot of error checking and handling, which you should not do if you want your program to be stable. Every system call, for example, returns a status code that should be checked; I don't do this to avoid clutter in this example.

```c
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

void read_all(int src, int dst) {
    char buf[BUFSIZ];
    ssize_t bytes_read, bytes_written;

    while ((bytes_read = read(src, buf, BUFSIZ)) > 0) {
        bytes_written = 0;
        while (bytes_written < bytes_read)
            bytes_written += write(dst,
                    buf + bytes_written,
                    bytes_read - bytes_written);
    }
}

int main(int argc, char** argv) {
    if (argc < 2) {
        printf("usage: %s <program> [<arg> ...]\n", base
        return EXIT_FAILURE;
    }

    // create pipes for standard input, output, and erro
    int stdin_pipe[2];
    int stdout_pipe[2];
```

```
    pipe(stdout_pipe);
    pipe(stderr_pipe);

    if (fork() == 0) {
        // child process

        // close write end of stdin and read ends of std
        close(stdin_pipe[1]);
        close(stdout_pipe[0]);
        close(stderr_pipe[0]);

        // change child's stdin, stdout, and stderr to u
        dup2(stdin_pipe[0], STDIN_FILENO);
        dup2(stdout_pipe[1], STDOUT_FILENO);
        dup2(stderr_pipe[1], STDERR_FILENO);

        // exec the given program
        if (execvp(argv[1], argv+1) == -1) {
            perror("failed to start subprocess");
            return EXIT_FAILURE;
        }
    }

    // parent process

    // close read end of stdin and write ends of stdout
    close(stdin_pipe[0]);
    close(stdout_pipe[1]);
    close(stderr_pipe[1]);

    // pass input to the child process
    read_all(STDIN_FILENO, stdin_pipe[1]);
    close(stdin_pipe[1]);

    // wait for child to finish
    wait(NULL);

    // read child's stdout and stderr
    puts("\nchild's stdout:");
    fflush(stdout);
    read_all(stdout_pipe[0], STDOUT_FILENO);
    close(stdout_pipe[0]);
    puts("\nchild's stderr:");
    fflush(stdout);
```

```
78
79      return EXIT_SUCCESS;
80  }
```

Here are some example runs (assuming that this program is called `pipe_test`):

Calling `grep -n '^ba' -` (search for lines beginning with "ba"), and passing in input via the terminal.

```
1  $ ./pipe_test grep -n '^ba' -
2  foo
3  bar
4  baz
5  qux
6  ^D
7
8  child's stdout:
9  2: bar
10 3: baz
11
12 child's stderr:
```

Example that writes to standard error:

```
1 $ ./pipe_test cat -x < /dev/null
2 child's stdout:
3
4 child's stderr:
5 cat: invalid option -- 'x'
6 Try 'cat --help' for more information.
```

Written 24 Dec, 2013 • View Upvotes

How does BitTorrent work?

87,550 views

ANSWER STATS

8,372	30	
VIEWS	UPVOTES	EDITS