Lecture 5: CSE 384 (System and Network Programming)

The Linux Command Line, Fifth Internet Edition

Chapters 6-10

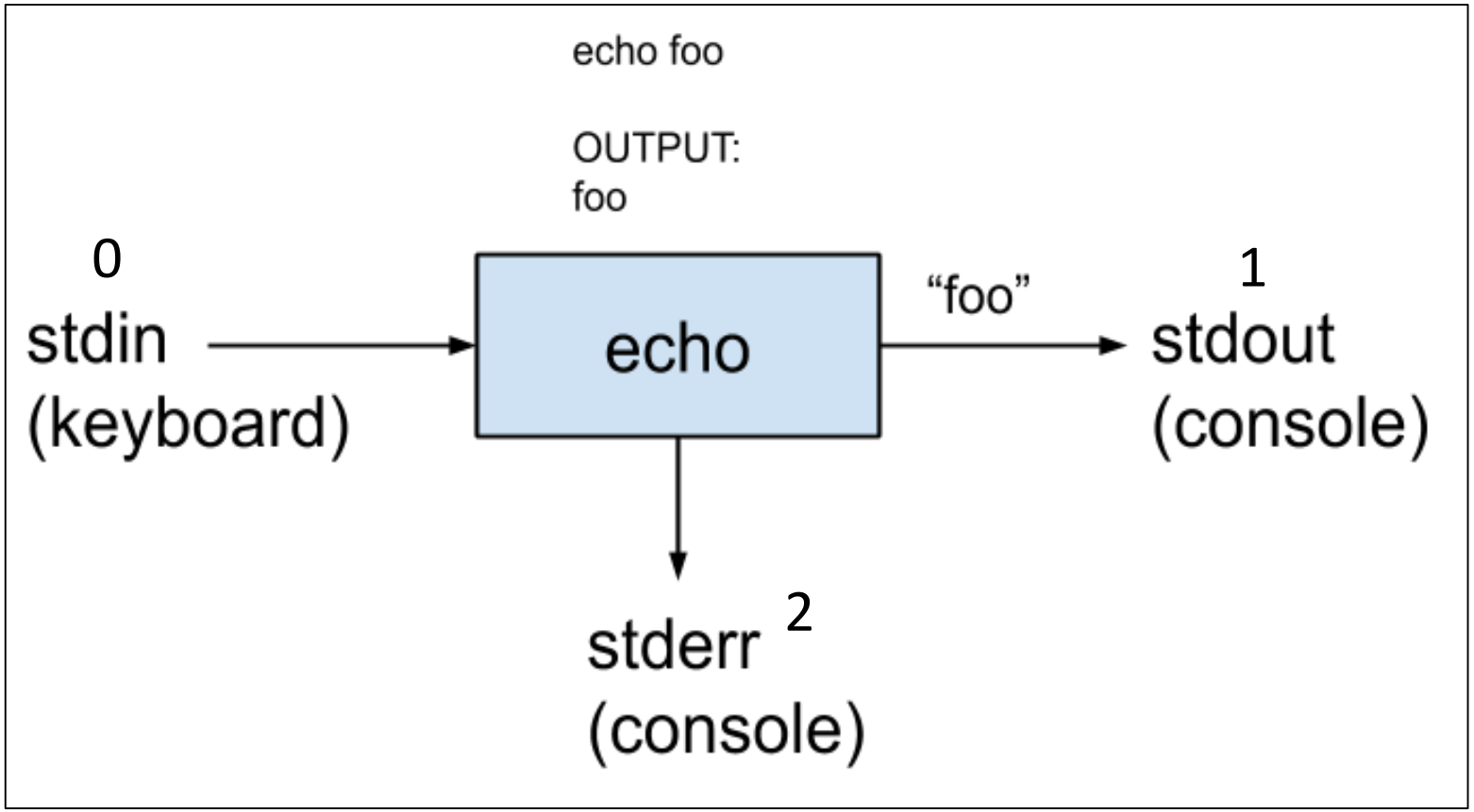
Overview

- I/O Redirection
  - STDIN, STDOUT, STDERR
    - Concept Overview
    - Examples
- Disposing Output
  - /dev/null
- Pipelines
  - Concept Overview
  - Examples

I/O Redirection: based on page 54 of the online version the Linux CommandLine

- redirect the input and output of commands to and from files
- connect multiple commands together into powerful command pipelines
  - In keeping with the Unix (Linux) model, everything is a file….
    - programs (such as ls) send output to a file called standard output: called stdout
    - Status (error) messages to another file called standard error: stderr
      - by default, both stdout/stderr are linked to the screen and not saved into a disk file.
    - Many programs take input from yet another file: *stdin (called standard in)*
      - *By default stdin is connected to the keyboard*
  - *Redirection allows us to change where output (stdout/stderr) goes (e.g. the screen), and where input (stdin) comes from (e.g. the keyboard)*

stdin, stdout, stderr



- In python: “print”
- In Java: “System.out.print”
- In C: “printf’
- fprintf(stderr)
- In C++ “std::cout”
- std::cerr

https://github.com/kennyyu/bootcamp-unix/wiki/stdin,-stdout,-stderr,-and-pipes

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ echo "foo"
foo
```

STDIN, STDOUT, and STDERR are files…

- Operating systems (under the hood), are often a tables with indices that point to resources.
  - Files (and other resources) are often referenced using descriptors (integer numbers) that serve as a handle to the resources.
- STDIN == file descriptor 0
- STDOUT = file descriptor 1
- STDERR = file descriptor 2

```
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX  20    /* max #files open at once */

typedef struct _iobuf {
    int  cnt;       /* characters left */
    char *ptr;      /* next character position */
    char *base;     /* location of buffer */
    int  flag;      /* mode of file access */
    int  fd;        /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin   (&_iob[0])
#define stdout  (&_iob[1])
#define stderr  (&_iob[2])

enum _flags {
    _READ   = 01,   /* file open for reading */
    _WRITE  = 02,   /* file open for writing */
    _UNBUF  = 04,   /* file is unbuffered */
    _EOF    = 010,  /* EOF has occurred on this file */
    _ERR    = 020   /* error occurred on this file */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)     ((p)->flag & _EOF) != 0)
#define ferror(p)   ((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->fd)

#define getc(p)    (--(p)->cnt >= 0 \
                ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p)  (--(p)->cnt >= 0 \
                ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()   getc(stdin)
#define putcher(x)  putc((x), stdout)
```

What are files anyway?
Original UNIX "FILE" concept implementation by Ritchie

Source: The C Programming Language 2nd Edition , page 176

Examples

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ ls -l > ls-output
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat ls-output
total 8
drwxrwxr-x 6 mwcorley mwcorley 4096 Jan 22 18:20 code_examples
-rw-rw-r-- 1 mwcorley mwcorley    0 Jan 24 14:29 ls-output
-rw-rw-r-- 1 mwcorley mwcorley    0 Jan 24 14:28 ls-output.txt
```

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ ls -l /bin/usr > ls-output.txt
ls: cannot access '/bin/usr': No such file or directory
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat ls-output
cat: ls-output: No such file or directory
```

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ ls -l /bin/usr 2> ls-output.txt
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat ls-output.txt
ls: cannot access '/bin/usr': No such file or directory
mwcorley@mwcorley-VirtualBox:~/Desktop$
```

Examples

- Redirect STDOUT
  - *ls -l /usr/bin > ls-output.txt*
  - *ls -l /usr/bin 1> ls-output.txt*
- Redirect STDERR
  - [me@linuxbox ~]$ *ls -l /bin/usr > ls-output.txt*
    - What happened?
    - Zero length! The destination file is always rewritten from the beginning.
  - [me@linuxbox ~]$ *ls -l /bin/usr 2> ls-error.txt*
- [me@linuxbox ~]$ *> ls-output.txt*
    - using the redirection operator with no command preceding it will truncate an ex-isting file or create a new, empty file.
- [me@linuxbox ~]$ *ls -l /usr/bin >> ls-output.txt*
- [me@linuxbox ~]$ *ls -l /usr/bin >> ls-output.txt*
  - append redirected output to a file instead of overwriting the file

Examples

- Redirecting Standard Output and Standard Error to One File
  - [me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
    - Using this method, we perform two redirections.
    - First we redirect standard output to the file ls-output.txt
    - then we redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation 2>&1.
  - [me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
    - *Streamlined notation &> in (modern BASH implementations) to re*direct both standard output and standard error to the file
  - append the standard output and standard error streams to a single file
    - [me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt

Disposing Output (/dev/null) (page 58)

- Sometimes its appropriate to disregard output from a command, we just want to throw it away.
  - error and status messages.
- [me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
- Often called the bit bucket
  - Old Unix concept and because of its universality
  - The old Unix joke: If I want to ignore this lecture, you might say that you're sending everything I say to /dev/null

Redirecting STDIN (page 59)

- Consider the Unix cat program – Concatenate Files
  - The *cat* command reads one or more files and copies them to standard output (note wildcard expand is sorted order)

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ echo "Mike" > file1
mwcorley@mwcorley-VirtualBox:~/Desktop$ echo "Corley" > file2
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat file* > file3
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat file3
Mike
Corley
```

  - Now run *cat* with no file arguments (assumes stdin)
    - [me@linuxbox ~]$ cat
      - *the quick brown fox jumped over the lazy dog*.
      - Type Ctrl-d (i.e., hold down the Ctrl key and press “d”)
        - to tell *cat* that it has reached end of file (EOF) on stdin

Redirecting STDIN

- Now run *cat* with no arguments again….
  - [me@linuxbox ~]$ cat
    - Type: *the quick brown fox jumped over the lazy dog*.
    - Type *Ctrl-d* (signal EOF)

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat > lazy_dog.txt
the quick fox
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat lazy_dog.txt
the quick fox
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat < lazy_dog.txt
the quick fox
mwcorley@mwcorley-VirtualBox:~/Desktop$
```

- Using the < redirection operator

  the source of stdin changed from the keyboard to the file lazy_dog.txt. We see that the result is the same as passing a single file

Pipelines

- read data from *stdin* and send to *stdout* is utilized by a shell feature called *pipelines*.
- Using the pipe operator | (vertical bar)
  - standard output of one command can be piped into the standard input of another.

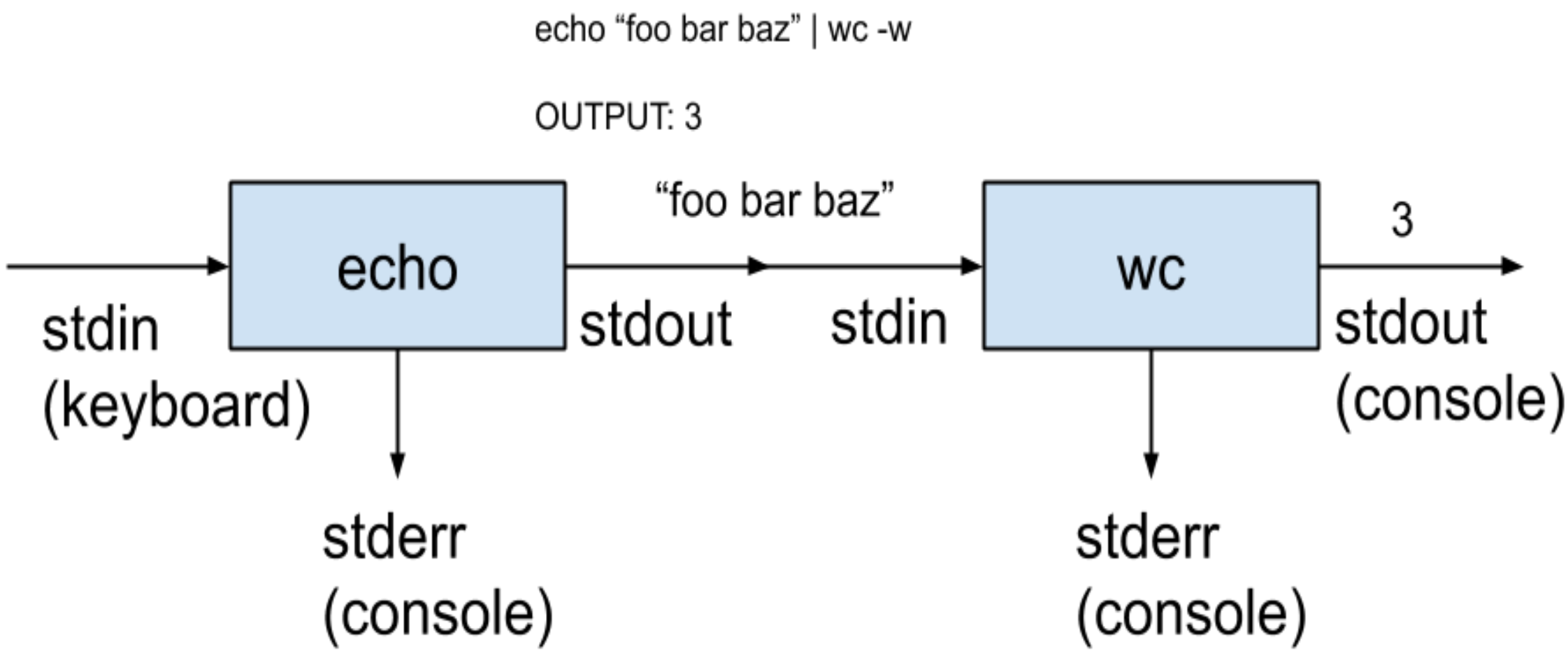
```
command1|command2
```

[me@linuxbox ~]$ ls -l /usr/bin | less

Pipe (|) Versus Redirection (>) - page 62

- The Difference Between **>** and **|**
  - the redirection operator connects a command with a file
    - *command1 > file1 -> echo "hello" > temp*
  - the pipeline operator connects the output of one command with the input of a second command
    - command1 | command2 -> ls -l /usr/bin | less
    - Be carful when you are learning about pipelines
    - What does ***command1 > command2***
      - Answer: sometimes something really bad.
        - # cd /usr/bin
        - # ls > less
          - Notice the # indicates root, (wiped out the less program)

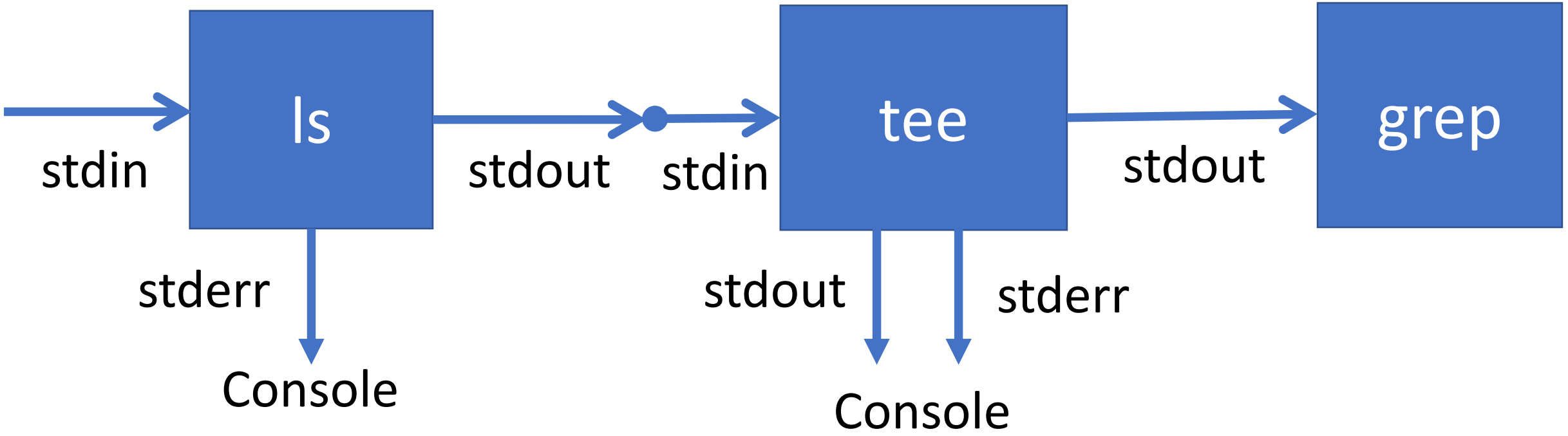
Pipeline (concept)



Source: https://github.com/kennyyu/bootcamp-unix/wiki/stdin,-stdout,-stderr,-and-pipes

Tee command

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
```



Filters - piping command together

- https://mwcorley79.github.io/MikeCorley/presentations/TLCL-19.01.pdf#page=86 (The Linux Command Line, William Shotts)
  - sort (page 62) --
  - uniq (unique) -- removes duplicates
  - wc (word count)
  - grep (page 63)
  - head and tail (page 64)
  - tee

Chapter 7: The Shell – Expansion and substitution

- Expansion
  - arithmetic
  - Brace
  - Parameter (shell variables)
- Substitution
- Quoting

- https://mwcorley79.github.io/MikeCorley/presentations/TLCL-19.01.pdf#page=86

Chapter 8: Keyboard shortcuts

- https://mwcorley79.github.io/MikeCorley/presentations/TLCL-19.01.pdf#page=86

- Command history
- Completion
- script

Chapter 9: Permissions

- https://mwcorley79.github.io/MikeCorley/presentations/TLCL-19.01.pdf#page=86
  - id – Display user identity
  - chmod – Change a file's mode
  - umask – Set the default file permissions
  - su – Run a shell as another user
  - sudo – Execute a command as another user
  - chown – Change a file's owner Permissions
  - chgrp – Change a file's group ownership
  - passwd – Change a user's pass

umask

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ umask
0002
```

- controls the default permissions given to a file when it is created.
- octal notation to express a mask of bits to be removed from a file's mode attributes.

chmod examples

- chmod g+rx a.out
  - - give the group read + execute permissions
- chmod g+rwx,o-x a.out
  - - give the group read + write + execute permissions, take way execute for the world
- chmod u+rwx,g+rx,o-rwx a.out == chmod 750
  - Give the owner full access (rwx), the group (rx), take away all access to the world

Special permissions: setuid, setgid, sticky bit

- setuid bit (octal 4000).
  - When applied to an executable file,it sets the effective user ID from that of the real user (the user actually running the program) to that of the program's owner.
    - chmod u+s program

```
mwcorley@mwcorley-VirtualBox:~/Desktop/code_examples$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 59640 Mar 22  2019 /usr/bin/passwd
```

- setgid bit (octal 2000), which, like the setuidbit, changes the effective group ID from the real group ID

- sticky bit (octal 1000)
  - prevents users from deleting or renaming files unless the user is either the owner of the directory, the owner of the file, or the superuser
    - often used to control access to a shared directory, such as /tmp.

Essential security related commands

- id – Display user identity
- chmod – Change a file's mode
- umask – Set the default file permissions
- su – Run a shell as another user
- sudo – Execute a command as another user
- chown – Change a file's owner90

Owners, Groups, and Everyone else (the world)

- user may own files and directories.
  - user has control over the files and directories owned
  - Users can, in turn, belong to a group consisting of one or more users who are granted access to files and directories by their respective owners.
  - an user/owner may grant some set of access rights to everybody, which in Unix terms is referred to as the world.

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ id
uid=1000(mwcorley) gid=1000(mwcorley) groups=1000(mwcorley),4(adm)
,24(cdrom),27(sudo),30(dip),46(plugdev),112(lpadmin),129(sambashar
e)
```

File Attributes: Read, Write, Execute Permissions

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

```
mwcorley@mwcorley-VirtualBox:~/Desktop$ cat /etc/shadow
cat: /etc/shadow: Permission denied
mwcorley@mwcorley-VirtualBox:~/Desktop$
mwcorley@mwcorley-VirtualBox:~/Desktop$
mwcorley@mwcorley-VirtualBox:~/Desktop$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1543 Jan 25 18:23 /etc/shadow
mwcorley@mwcorley-VirtualBox:~/Desktop$
```

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me    0 2016-03-06 14:52 foo.txt
```

The first 10 characters of the listing are the *file attributes*. The first of these characters is the *file type*. Table 9-1 describes the file types we are most likely to see (there are other, less common types too):

Table 9-1: File Types

Attribute	File Type
-	A regular file.
d	A directory.
l	A symbolic link. Notice that with symbolic links, the remaining file attributes are always "rwxrwxrwx" and are dummy values. The real file attributes are those of the file the symbolic link points to.
c	A *character special file*. This file type refers to a device that handles data as a stream of bytes, such as a terminal or `/dev/null`.
b	A *block special file*. This file type refers to a device that handles data in blocks, such as a hard drive or DVD drive.

Owner	Group	World
rwx	rwx	rwx

Table 9-2 describes the effect the `r`, `w`, and `x` mode attributes have on files and directories:

Table 9-2: Permission Attributes

Attribute	Files	Directories
`r`	Allows a file to be opened and read.	Allows a directory's contents to be listed if the execute attribute is also set.
`w`	Allows a file to be written to or truncated, however this attribute does not allow files to be renamed or deleted. The ability to delete or rename files is determined by directory attributes.	Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set.
`x`	Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set as readable to be executed.	Allows a directory to be entered, e.g., cd *directory*.

File Attributes	Meaning
`-rwx------`	A regular file that is readable, writable, and executable by the file's owner. No one else has any access.
`-rw-------`	A regular file that is readable and writable by the file's owner. No one else has any access.
`-rw-r--r--`	A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world-readable.
`-rwxr-xr-x`	A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else.
`-rw-rw----`	A regular file that is readable and writable by the file's owner and members of the file's group owner only.
`lrwxrwxrwx`	A symbolic link. All symbolic links have “dummy” permissions. The real permissions are kept with the actual file pointed to by the symbolic link.
`drwxrwx---`	A directory. The owner and the members of the owner group may enter the directory and create, rename and remove files within the directory.
`drwxr-x---`	A directory. The owner may enter the directory and create, rename, and delete files within the directory. Members of the owner group may enter the directory but cannot create, delete, or rename files.

- chmod - change the mode (permissions) of a file or directory
  - only the file’s owner or the superuser can change the mode of a file or directory
  - supports two distinct ways of specifying mode changes: octal number representation, or symbolic representatio

  - Octal (base 8), hexadecimal (base 16)
    - number systems often used to express numbers on computers.

Table 9-4: File Modes in Binary and Octal

Octal	Binary	File Mode
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

By using three octal digits, we can set the file mode for the owner, group owner, and world.

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me    0 2016-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me    0 2016-03-06 14:52 foo.txt
```

Table 9-4: File Modes in Binary and Octal

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me    0 2016-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me    0 2016-03-06 14:52 foo.txt
```

By passing the argument “600”, we were able to set the permissions of the owner to read and write while removing all permissions from the group owner and world. Though remembering the octal to binary mapping may seem inconvenient, we will usually have only to use a few common ones: 7 (`rwx`), 6 (`rw-`), 5 (`r-x`), 4 (`r--`), and 0 (`---`).

`chmod` also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts.

- Who the change will affect
- Which operation will be performed
- What permission will be set.

rwx-r--r

111010

chmod examples

chown

SU and SUDO