Table of Contents

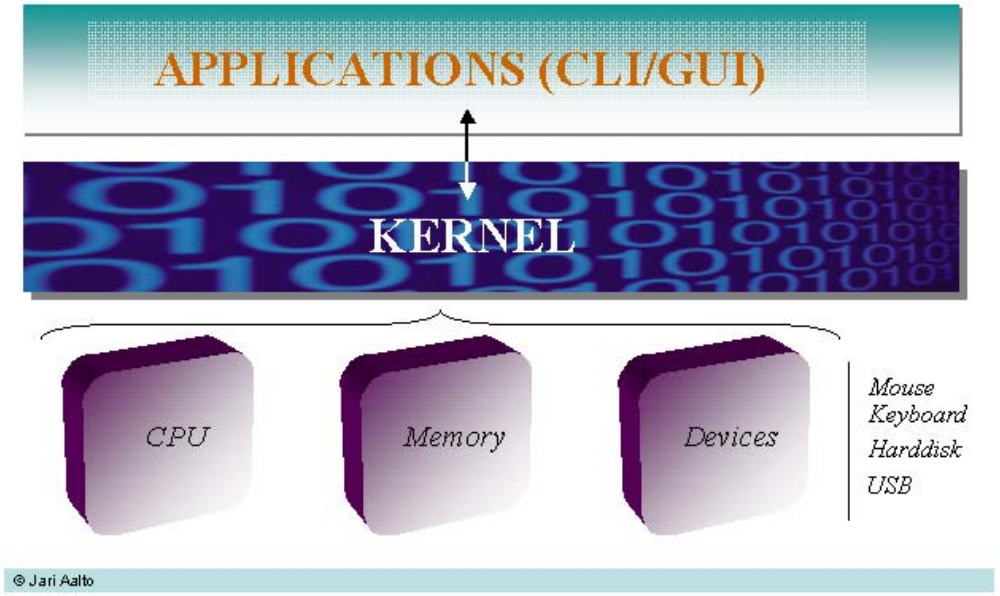
Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

1.0 Linux operating system

1.1 The Operating System

The operating system consists of the Kernel which is the hearth of the whole system. Its responsibilities include managing the system's resources and devices like harddrives, USB peripherals, mouse etc.



Picture 1. The operating system and Kernel

1.2 A portable operating system - History

One of main design objectives of the grandfather Unix, where Linux ows heritage, was to create a computing environment that is consistent across different hardware configurations (IEEE Std. 1003.1; Standard for Information Technology, Portable Operating System Interface - POSIX). In other words, the designers wanted to make Unix a portable operating system. The first step in learning how to use the Unix system is to understand its underlying design. The word portable here refers to the meaning where the *application* can be executed on different computers with a little or no modifications to the source code. On the other hand, an *operating system* runs on a computer and must know a

great deal about the computer hardware it controls. Given differences between CPU chips, bus designs and other computer hardware features, the chances are small that an operating system could be moved from one machine to another with "little or zero modifications".

To minimize porting requirements, there is need to insulate OS from any special features of a particular piece of hardware. In fact, the more generic core is, the more portable it will be. Put another way, the fewer capabilities that Unix requires from a computer, the less OS has to depend on any specific hardware characteristics. The result is that Linux assumes the hardware it runs on is a very simple computer. The operating system is then build on the few capabilities supplied by such a machine. The idea is that every computer providing a minimal number of features should be capable of supporting the Linux system. Therefore the carrying principle has been that a computer consists of two things: *process* and *data*. A process is a currently executing program. Data is everything else.

1.3 Processes (forking)

The operating system is made for multiprocessing. This means that several processes can be executed simultaneously. The OS is a *time-sharing* operating system: it keeps track of all the processes ready to run and executes each one for a fraction of a second, sharing the available CPU time among the executing processes. If there are more CPUs, there is more CPU time to share. The OS defines a process as an instance of an executing program. If the same program is being executed twice, it requires two processes. Everything done on a OS is done by a process.

1.4 Data

The designers of OS made a limiting assumption with a regard to the data handled by the Operating system. To generalize how data is handled, all data is represented by a *stream of characters*. Thus regardless of how the data is generated or stored, it moves operating system a character at a time. Because all data looks alike, the details of how data is stored or retrieved is of no concern. The OS doesn't care which *device* generated the data, as long as the data still looks like a stream of characters. The same is true for output to a device. A *stream* of data are sent regardless of which device receives it.

1.5 Devices

The OS assumes that all sources and destination of data (i.e. devices) are part of the computer's file system. That is, all devices connected to a operating system appear as files in the file system (look at directory `/dev`). Even though each hardware configuration is different, the system views all configurations the same way; as set of files that can send and receive streams of characters to currently active processes. Viewing all devices as files greatly enhances portability. Note the elegance of this design. Because all devices look like files and all files look like streams of characters, a program that can read one character stream can read any character stream. put another way, if a program can read one file, it can read any other file or device on the system (provided that it has permissions to access the file).

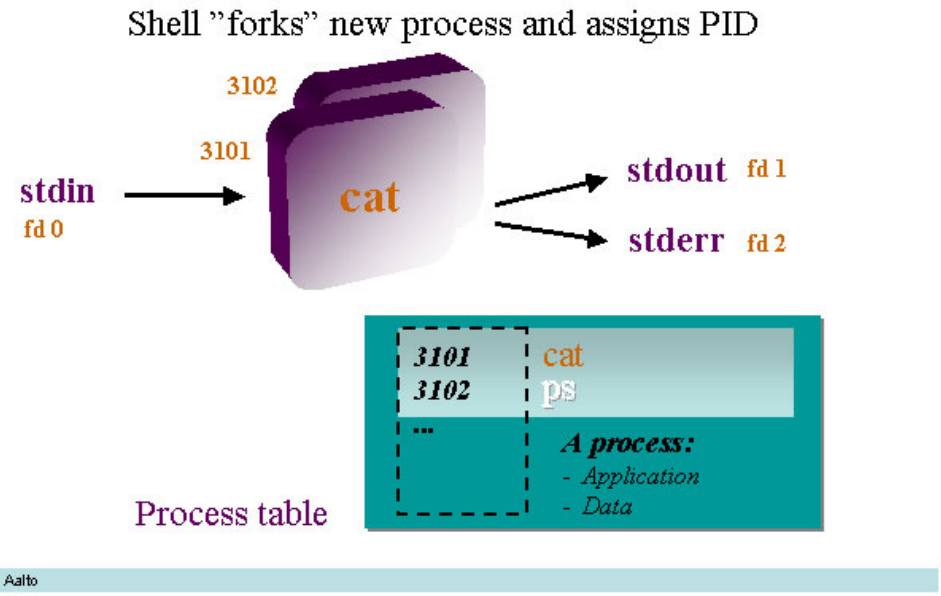
1.6 Processes

A process results when an executable file and the appropriate data streams are combined and then scheduled for execution, usually in response to a command entered by a user, although operating system can also start processes on its own. A process is independent of the program it executes and the data it processes. Put another way, a process can execute any program connected to any data stream on a system. It is important to emphasize that a *process* is not a *program*. A process is an environment within which a program executes. Part of a program's *ENVIRONMENT* includes the data streams it is connected to. A process is dynamic, it exists only while it is being executed. Further the actual connection of an executable program with data streams is done at runtime, when the process is created.

How does a process know which stream to connect to the executable program? there are two ways. first, a stream can be directly accessed by name from inside the process. Second, a process can use the default streams.

1.7 Default streams: stdin, stdout, stderr

Every process, when created, is connected to three default streams: standard input `stdin`, standard output `stderr`, standard error `stderr`. Unless redirected, all three standard streams are connected to the process's terminal.



Picture 2. The process management. Each process is separate entity and is recorded in process table by the unique process id number (PID). File descriptor numbers (fd) are used for I/O when a file or other device is opened. By convention, programs write their normal output to unit 1 `stdout`, and expect input on unit 0 `stdin` and error or diagnostic output goes to unit 2 `stderr`.

The standard input stream is where the process will get characters if the program reads data without referring to a specific file. The standard output stream is where program prints data without referring to a specific file. The standard error is where a process writes error messages to the user when the program doesn't identify a specific error-logging file. The standard error is used to separate error messages from ordinary output. By default, a command will read from that terminal's *keyboard* and write output and errors to that terminal's screen. Because a process

is independent of the data streams it uses, the process doesn't care how the data is it is using is generated. This independence lets operating system to generate and consume character streams in yet a third way - from other processes.

The concept of streams is evident in literally all programming languages. Take for example the **JAVA** programming language. The System.out is equivalent to `stdout`, System.err is equivalent to `stderr` and System.in is equivalent to `stdin`.

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing


```
//  To compile: javac test.cc
//  To run:     java test

public class test {
    public static void main (String args[]) {
        System.out.println("stdout stream - for normal messages");
        System.err.println("stderr stream - for error messages");

        //  The try-catch is required for System.in

        char answer = ' ';

        try {
            System.out.print("Reading stdin now. Type character: ");
            answer = (char) System.in.read();
        }
        catch(Exception error) {
            System.err.println("Couldn't read. The error is: " + error);
        }

        System.out.println( "You wrote: " + answer );
    }
}
```

Here is an example using the **C++** object oriented programming language

```
//  To compile: g++ -o test test.c
//  To run:     test

#include <iostrem.h>

main ()
```


```
{
    cout << "To standard output" << endl;
    cerr << "To standard error"  << endl;

    char answer;

    cin >> answer;

    cout << "You wrote: " << answer << endl;
}
```

Here is an example using the **C** programming language:

```
/*  To compile: gcc -o test test.c
    To run:     ./test
*/

#include <stdio.h>

int main (void)
{
    char answer;

    fprintf(stdout, "To stdout\n");
    fprintf(stderr, "To stderr\n");

    /* writes to stdout  */
    printf("Reading stdin now. Type character: ");

    answer = fgetc(stdin);

    printf("You wrote: %c\n", answer);
    exit(0);
}
```

Here is example using the **Perl** programming language


```
#!/usr/local/bin/perl
#
# No compile needed
# To run: perl test.pl

use strict;

sub Main ()
{
    print STDOUT "To standard output\n";
    print STDERR "To standard error\n";

    print "Reading stdin now. Type character: ";

    my $answer = <STDIN>;

    print "You wrote: $answer\n";
}

Main();

__END__
```

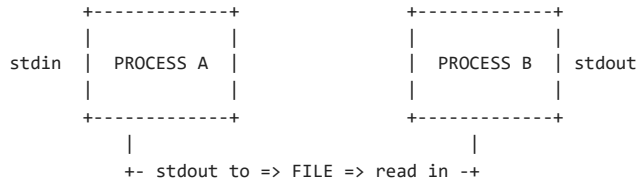
1.8 Pipelines

To make more efficient use of multiprocessing resources, the OS let's output of one process be connected to the input of another. Such connections are called *pipelines*, like illustrated below

```
       +-------------+               +-------------+
       |             |               |             |
stdin  |  PROCESS A  | stdout | stdin |  PROCESS B  | stdout
       |             |               |             |
       +-------------+       ^       +-------------+
                             ^

Pipe (|) character connects A/B
```

Anything done by a pipeline can also be done by storing the output of process A into a file and then using the same file as input for process B.



However, a pipeline is more efficient because it doesn't require that the output from process A ever been stored in a file. It is directly consumed by process B. Pipelines play an important part in the creation of applications with tools. Much of the power comes from these few simple ideas:

- The OS views computers as suppliers of two things: *process* and *data*.
- All *data* is represented by streams of characters.
- All sources and destinations of data (i.e. *devices*) are seen as files in the file system
- A *process* results when an executable file and the appropriate data streams are combined and executed. A process includes one or more currently executing programs.
- Every process is connected to three default *streams*: `stdout`, `stderr` and `stdin`.
- Unless changed, all three default streams are connected to the process's control *terminal*.
- The OS let's the output stream of one process be connected to the input stream of another to form a *pipeline*.

2.0 Basic file commands

2.1 Logging in to a host with ssh and transferring files

```
ssh [user@]host [command]
scp [[user@]host1:]file1 [...] [[user@]host2:]file2
```

To connect to the host, start a terminal program like `telnet` or `ssh` After you have successfully connected to the host, you will see the shell's command prompt.


```
$ ssh login@host.example.com
Password: <TYPE PASSWORD>
```

The command prompt line is now waiting for commonads. You can exit with:

```
logout
```

Or with command:

```
exit                    => same as C-d
```

The `scp(1)` command, part of the SSH installation, provides remote file transfers via secure channel. It has virtually replaced the `ftp(1)` command due to security reasons.

```
+---------+                  +---------+
|         |       SSH        |         |
+ Host A  + ---------------- + Host B  |
|         |    (encrypted)   |         |
+---------+                  +---------+
```

To *upload* files to remote host, the command is:

```
scp file.txt    login@host.example.com:tmp/dir/
    |           |
    FROM FILE   TO DESTINATION
```

To *download* files from remote host, the command is:

```
scp -r login@host.example.com:tmp/data $HOME/tmp/
                              |
   From remote dir, under $HOME
```


2.2 Changing and listing directories (cd, ls)

```
cd [directory]
ls [directory ...]
```

Directories are a way to organize the files. It is possible to create directories within directories. The various commands will allow you to move back and forth within directories, to move files into and out of directories, and to create/remove directories. To move to your `$HOME` directory anywhere you can use either one of these commands:

```
cd                      # Go to $HOME
cd ~                    # tilde is a synonym for $HOME
cd $HOME                # Env variables in BIG LETTERS
```

To change into different directory from your home directory (your home directory is the one you are automatically put in when you log onto your account), you would type:

```
cd /tmp                 # System's temporary directory
```

This will take you to the directory just **above** the one you are in:

```
cd ..
```

To list contents of directory:

```
ls
ls $HOME
ls ~/doc ~/tmp
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

A special directory, **current** directory, is indicated with dot. In every directory there are always these two: "." and ".."

```
ls -a
.                       # current directory
..                      # previous directory
```

Exercise: Why nothing happens with this command?

```
cd .
```

2.3 Print working directory (pwd)

```
pwd
```

Where are we? Sometimes while moving from one directory to another we lose track of where we are. To find out what is the current directory, use the `pwd` command (print working directory).

```
cd

pwd
=> /home/foo

cd /tmp
pwd
=> /tmp
```

3.0 Dealing with directories and files

3.1 Remembering directories (pushd, popd)

```
pushd .
cd <directory>
popd
```

Sometimes it may be handy to save the current directory, when you want to do something elsewhere for a while. To push a directory to a stack, use the `pushd(1)` command. To return back to last directory in the stack, use `popd(1)`

```
pwd
=> /home/joe/tmp/test/dir

pushd .                         # save the current location
cd ~/publich_html

.. do something
..and return back

popd                            # return to saved location
pwd
=> /home/joe/tmp/test/dir
```

3.2 Making directory (mkdir)

```
mkdir [options] <directory> ...
```

The `mkdir` command allows you to make a directory while the rmdir removes a directory. To make a directory, you would type:

```
mkdir ~/public_html
```

There is also very useful option `-p`, which makes all the directories if they do not exist yet:

```
mkdir -p ~/tmp/1/2/3/4
```

Exercise: Run the command first without `-p` and examine the output. What happened? Next, use `-p` and verify that the directories got created.

3.3 Removing directory (rmdir)

```
rmdir [options] <directory> ...
```

To remove a directory, it must be empty of all files. After all the files have been removed, you return to the home directory or if it is a subdirectory, the directory right above the directory to be deleted. You would then type (for example, deleting the ~/tmp/test directory):

```
mkdir -p ~/tmp/test
rm ~/tmp/test/*
rmdir ~/tmp/test
```

3.4 Creating an empty file (touch)

```
touch [options] <file> ...
```

you can create an empty file with various of ways, of which the `touch(1)` command may be the most popular. It actually sets the current date and time for the file, but as a side effect creates a file if it does not exist.

```
touch ~/tmp/new.txt
```


You can also use `echo(1)` command to add an empty newline to a file. Note that option `-c` suppresses printing the final newline (contrast to Linux, this is not supported in all Unix platforms).

```
echo -c "" > ~/tmp/new.txt
```

To create a completely empty file, or truncate existing file, you could read from the *null device* (also called bit-bucket) and direct its output to a file. The `/dev/null` is also handy when you want to *empty the contents* of, say, log files, but preserve their names in the directory

```
cat /dev/null >  ~/tmp/new.txt
```

```
for file in *.log; do echo "Found file $file" ; done
                      +----------------------
                      |
                        cat /dev/null > $file
```

3.5 Copy command (cp)

```
cp [options] <file> [files ...] <destination directory>
cp [options] <file> <destination file>
```

The cp(1) command is used to copy files to other filenames or to copy them from one directory to another directory. To copy a file to another filename (the file test to test2, for example), you type:

```
$ cp test test2
```

To copy a file from one directory to another. For example, the file test from your home directory into the $HOME/tmp directory you would type:

```
$ cp test ~/tmp/
```

Another useful point to know in copying files to or from directories is that you can copy all the files at one time using a `*` wildcard with the command. For example to copy all the files from one directory(1) to another directory(2), you would type:

```
$ cp ~/tmp/1/* ~/tmp/2/      # Directories MUST EXIST
```

Exercise: Examine manual page of `cp(1)`. Examine what options `-r` and `-i` would do?

3.6 Move command (mv)

```
mv [options] <file> <...> <to dir>
mv [options] <file> <to file>
```

This command stands for "move" and it is used to move files within your directory structure or to rename your files. To move (rename) your *files*, you would type:

```
$  mv file file.txt
```

The moving of files within directories is exactly the same as in the copy (cp) command.

```
$ mv ~/tmp/file ~/tmp/file.txt
```

Moving *directories* is just the same, here the `tmp` is renamed to `tmp2`

```
$ mv ~/tmp ~/tmp2
```

3.7 Remove command (rm)


```
rm    [options] <file> ... <to dir>
rm -r [options] <file or dir> ...
```

The `rm(1)` command stands for remove and it is the means to permanently delete your files. To remove a file you type:

```
$ rm file1
```

To remove many files, list them in the command

```
$ rm file1 file2 file2
```

Or you can remove all files starting with "file":

```
$ rm file*
```

You **cannot remove a directory** right away with `rm(1)`, because there is separate commands for directory handling. (To experienced users: see the manual page for `-r` and `-f` options to delete directories. *Caution:* you have been warned)

```
$ rm ~/tmp
rm: ~/tmp is a directory            (ERROR MESSAGE DISPLAYED)
```

3.8 Paginating content (more, less)

```
less [options] <file> ...
```

The Linux system has the Windows equivalent pager command `more(1)`, but there is much more better pager program `less(1)` which may need to be installed separately. You can't go back the pages with the traditional `more(1)` command, but `less(1)` offers that and lot more

interesting search and move capabilities. Set the **$PAGER** environment variable permanently to replace `more(1)` with `less(1)`.

```
$ export EDITOR='emacs -q -nw'
$ export VISUAL='emacs -q -nw'
$ export LESS='-eimMn'
$ export PAGER='less'
$ export MANPAGER='less'
```


The `less(1)` command has lot of command line switches of which here is only the most interesting ones:

```
-e      Exit at the end-of-file (like more)
-i      Ignore character case during searches
-m      Keep the percent prompt visible (normally only ":")
-M
-n      Do not show line numbers
-N      Display line numbers
-s      Squeeze many blank lines to one
```

The most commonly user commands inside the program are:

```
-------------------------------------------------------------
h       Help
q       Quit
-------------------------------------------------------------
/RE     Search regular expression forward. Type "n" to
        repeat search. "N" to repeat to other direction.
?RE     Search regular expression backward. Type "n" to
        repeat search. "N" to repeat to other direction.
-------------------------------------------------------------
SPC     Next page       d = Down half window
b       Back page       u = Up half window
<       Beginning of file
>       End of file
m L     Mark position with LETTER
' L     Jump to position LETTER
F      Run in "follow mode" for log files, C-c to abort
```

```
------------------------------------------------------------
:NNN    Go to line number NNN
:n      Next file
:p      previous file
-i      Ignore case
-N -n   Line numbers on/off
-s      Squeeze multiple blank lines
-G -g   Highlight on/off
------------------------------------------------------------
```


3.9 Detecting type of file (file)

```
file [options] <file> ...
```

The `file(1)` command uses a table (usually located in /etc/magic) to help it determine what sort of file is in question. If the `file(1)` reports the file as text, it is probably safe to edit or read it with `less(1)`. If the file indicates that it contains executable code, then it may be best examined with the octal dump command `od(1)`.

```
$ ( cd /usr/local/bin; file * ) | less

apachectl:              Bourne shell script text
perl:                   symbolic link to /usr/bin/perl
wkdemenu.pl:            perl commands text
xmms-config:            Bourne shell script text
c++:                    ELF 32-bit LSB executable, Intel 80386
```

To see the real byte content of the file, use `od(1)` with hex and character dump command line options. These options `-x` and `-c` depend on the operating system in use, see man page for more. E.g the hex option might have been `-h` instead of `-x`.

```
$ /bin/echo -e "test\r\n"  > test.tmp
$ od -c -x test.tmp
0000000    7465    7374    0a0a
           t   e   s   t  \n  \n
0000006
```

3.10 Other file system related commands

- `quota(1)`, display home directory usage and limits. Usually called with `-v` option.
- `df(1)` reports disk free in blocks, and i-nodes. See also `bdf(1)` Berkeley disk free usage, reports statistics [May need to be installed separately]
- `du(1)`, disk usage, summarizes disk usage in a specified directory hierarchy.

4.0 The command processor (shell)

Everything done on the OS is done by a process, so how do you create a process? One way is to write programs that know how to create processes. however, this solution requires far too much effort to execute a single command. Like virtually all operating systems, a command processor acts as the interface between a command. The command processor is called *shell*. Several command processors are available in the operating system and more can be downloaded and installed if needed. Three of the most common command processors are *Bourne Shell*: sh (sometimes `sh-bourne`), `ash` ("A shell"), GNU bash, ksh, zsh; The *C shell*: csh and and its derivate tcsh.

You can find out what shells are available in your system by looking into the `bin` directories:

```
$ ls -l /bin /usr/bin /usr/local/bin | grep 'sh$'
```

The mother of all shells is the *Bourne Shell* and nowadays vastly improved *Bourne Again Shell* which was developed to take advantage of the features that were added to the `ksh`, `csh` and `tcsh` shells that tried to improve the original and limited `sh` functionality. The whole purpose of the shell is to start processes running your commands. The shell *prompts* you for a command, using some character delimiter that are typically expressed by using `$` for *Bourne* shell and `%` for *C* shell. When you enter a command, the shell reads the input, interprets the command line and then creates a process to execute it. The shell sits between the user and the operating system. The shell *scans* the entered command line to figure out how to create the process that executes your command.

```
-------------------------------------------------------------
USER       |    SHELL              |   Operating system
==>  ==>   |    ==>   ==>   ==>    |   ==>  ==>  ==>  ==>
           |                       |
Writes     |    Interprets and     |   Runs the process created
command    |    scans the line     |   by shell
-------------------------------------------------------------
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

This may seem counterintuitive, that your command is not directly executed. It appears that the OS finally executes the command you entered. For example, you enter the `date(1)` command and the system prints the current time and date

```
$ date
Tue Oct 31 23:20:51 2000
```

Let's take a closer look at how the `date(1)` command is actually executed. Recall that every process is assigned three standard streams: `stdout`, `stderr`, `stdin`. Here, the output has been printed on the process's `stdout`, which has been connected to your computers terminal. Thus, although you entered on only word `date`, the shell interpreted the command to mean that you wanted the output to go to the terminal. The shell then created a process to execute the command and *connected* its `stdout` to your terminal. Your terminal is actually a file in in the system. See command `tty(1)` to find your which file is your terminal connected to.

```
$ tty
/dev/pts/12
```

Exercise: Explain this command. Use your own tty's value here:

```
$ date > /dev/pts/12
```

4.1 Shell metacharacters

Like all data on the system, the output of a `date(1)` is a stream of characters. Thus, you can send the date command's output to somewhere else than the terminal, like files.How do you create a process whose `stdout` is connected to file? The answer is to use special characters, called *shell metacharacters*, to tell the shell that the `stdout` of this process is to be connected to a specified file. This is usually called *redirecting* the output and it is represented by the ">" character.

```
$ date > time.txt
       ----------
```

In English, this command says to put the time and date into the file `time.txt`. Note that the underlined part is not part of the command that is executed. Instead it describes how the process's output is to be handled: it says, connect the `stdout` of the process to a filename that follows. Shell will recognize the metacharacter ">" even if it is not surrounded by spaces.

```
$ date>time.txt
      |
      Spaces are not necessary, but recommended for readability
```


In reality the statement presented contains two distinct piece of information. One, represented by the word `date`, is the program to execute. the other is a signal to the shell describing how the process's output is to be handled. The date command does not actually care if there is data send to it, it merely **outputs** the date and time. Second, the file `time.txt` does not actually **know** that the characters are coming from the `date(1)` command.

How the command is actually accomplished? first, the shell interprets the command line by scanning line for any *shell metacharacters*. The shell finds ">", which tells that it must connect the process's output to the file that follows it. Because the part "> time.txt" is a directive to the shell, it removes it from the command line and creates a process by running command `date(1)`. All this can be boiled down to one vitally important rule: *every line is examined for metacharacters before it is processed further*. Any metacharacters found are interpreted before any process is created. There are no exceptions, and in fact shell does not try to determine if the resulting command line makes sense. The shells job is to create the process for it if possible. For example, suppose that you make an error while writing the filename. Even though the command line looks odd, the shell dutifully runs the command as you requested. Many of the user errors are caused by typing mistakes like this.

```
$ date > 1
```

The command actually erases any existing file `1` and overwrites the content with the output of the `date(1)` command. If the file *does not exist*, the ">" tells the shell creates one for the output. The shell offers another output redirection that doesn't erase the content of the file, but appends to the end of it: ">" (See POSIX commands cat and date).

```
$ date >> last-login.txt
$ date >> last-login.txt
$ cat last-login.txt
```

Exercise: What happens if you now run this command? Look into the file to see the content

```
$ date > last-login.txt
```

4.2 Redirecting input

The shell also provides a way to redirect the input of a process. The input redirection, `stdin`, metacharacter is "<". When the input is redirected, the *shell* connects the listed file to the process's standard input. Both the metacharacter and the file name following the it are removed from the command line. Virtually all commands that read from a file, can be fed with the redirection.

Note: It is important to understand that some commands are *builtin* to shell (see POSIX and Bash manual section Shell Builtin Commands). These commands are typically **cd**, **echo**, **pwd**, **test**. But there may be also equivalent binary programs located in the system `/bin` and `/usr/bin` directories.

Exercise: What is the difference between the two `cat(1)` commands?

```
echo "this is test" > test.tmp

cat test.tmp
cat < test.tmp
```

Exercise: examine the `echo(1)` manual page and find out what following command does.

```
echo -e "This line\nAnother line\t1\t2\n"
                  |             |
                  Newline       Tab character
```

Exercise: The **end-of-line** convention in the Windows and Linux system is different. The Windows terminates the lines with two characters **\r\n** where Linux system uses only **\n**. Create a file with two lines in Linux style and in Windows style and output both files to the screen using `cat -v FILENAME`

The typical escape sequences are below. See Bash manual section ANSI-C Quoting for more information.

```
\a      Alert, bell
\b      backspace
\c      suppress (don't print) trailing newline
\f      form feed
\n      newline
```

```
\r      carriage return
\t      horizontal tab
\v      vertical tab
\\      regular backslash
\nnn    Octal ascii char, e.g. octal 101, decimal 65 is char 'A'
```


The `cat(1)` command copies its `stdin` to its `stdout`. This means that both standard streams can be redirected in the same command. Commands that let both their standard input and standard output to be redirected are called *filters*. For example to copy a file with the `cat(1)` command, you can use either of these choices:

```
$ cat < file-source > file-destination
      |             |
      |             Shell connects STDOUT
      Shell connects STDIN
```

```
$ cat file-source   > file-destination
      |             |
      cat open FILE Shell connects STDOUT
```

4.3 Ignoring redirection

How does `cat(1)` know when to read from `test.tmp` or its standard input? If `cat(1)` is passed to an *argument*, it assumes the argument is the name of the file (or files if there are multiple arguments) to print. If cat receives *no arguments*, it reads the `stdin`.

Exercise: Following these rules, how do you interpret this command line?

```
$ cat file1 < file2
```

The command "cat file1 < file2" demonstrates an important point. Just because a command's standard input or output has been redirected doesn't guarantee that the redirection will be used. Further, the command doesn't know it has been redirected. Here is another example of the redirection:

```
$ date < test.tmp
Tue Oct 31 23:20:51 2000
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

Unlike the `cat(1)` the `date(1)` command does not check its standard input under any circumstance and the *redirection has no effect*. Also note (by the absence of an error message) that the shell is unaware that the redirection is useless; it simply does what you ask without questions. Incidentally, the same would happen with any command that ignores its standard input, such as `ls(1)`, `cd`, `mkdir(1)` to name a few.

Exercise: Examine next commands in context below. Suppose hat all the files in question exists. Pay attention to the fact, what are the program *arguments* and what are the parts that are handled by the *shell*.

```
$ ls
$ ls > test.tmp
$ ls < test.tmp

$ cat test.tmp
$ cat test.tmp < new-file.tmp
$ cat < new.file.tmp
```

4.4 Redirecting the standard error

The standard error has its own special meta character "2>", which tells the *Bourne Shell* to connect this process's standard error to the listed file. No spaces are allowed between the number 2 and the character >. for example the `cat(1)` prints an error when it cannot find the file. (The exact error message depends on the operating system)

```
$ cat qwerty
cat: qwerty: No such file or directory
```

The error message above is sent `stderr` standard error stream. You can't tell this from the output above, because both the `stdout` and `stderr` is by default connected to the same terminal device you're using. To demonstrate that `stderr` really is used for the error message, we try to redirect the `stdout`

```
$ cat qwerty > test.tmp
cat: qwerty: No such file or directory
```

As expected, the outcome is the same, the shell *creates* `test.tmp` before `cat(1)` command is even run. Thus the outcome of the command is to create an empty file and print the error message to the terminal. Below, the error message is correctly redirected to a file using "2>". The meta character work the same as in standard output redirection ">" where a file is created or emptied if it already exists. To append error message to the file there is similar ">" for standard error "2>>".


```
$ cat qwerty 2> test.tmp
$ cat test.tmp
cat: qwerty: No such file or directory
```

Exercise: Assuming that none file does not exist, explain this command line's output. Who prints the error? Why doesn't the **2>** send the error to errfile, but the error message appears to the current terminal tty(1)?

```
$ cat < none 2> errfile
none: cannot open
```

4.5 HERE documents (in-line redirection)

So far, all the redirections done have involved files. The shell has another input redirection, called *in-line redirection* or commonly named as HERE documents (see POSIX and Here Documents in Advanced Bash-Scripting Guide). The in-line redirection meta character has two parts, the redirection represented by "<" and the stream end *MARKER*. because the entire input stream is listed as part of the command line, a unique token must be used to show where the input stream ends. As an example, this command prints the listed lines. Be very careful when writing the ending marker, it must be to the *left*, at the beginning of the line and written *exactly as you gave it*.

```
$ cat << 'EOF'
>This is an example of
>"HERE is" redirection
>EOF
This is an example of
"HERE is" redirection
$
```

In this command, EOF (End of file marker) is the beginning and ending token. You can use any set of characters to mark the input stream. In-line redirection is a flexible way to create input and simple files. Note the additional ">" characters at the beginning of each line. They are the Bourne shell's *secondary prompt* characters. It is used when a command line spans more than one line and shell]s way of telling that it needs

more input. The output from a command that has in-line redirection ca also be redirected to a file. For example these command lines write a simple file without no editors:

```
$ cat << 'EOF' > test.tmp
> Time is now
>EOF

$ date >> test.tmp
$ cat test.tmp
Time is now
Tue Oct 31 23:20:51 2000
```

Exercise: Will the above work like in previous example?

```
$ cat << 'EOF'
> Time is now
>EOF > test.tmp
```

Exercise: What would following commands do and why?

```
cat no-such-file.txt    > test.tmp 2> error.log
echo "This text here"   > test.tmp
echo "This text here"  2> test.tmp

cat > test2.tmp

line1
line2
[Press C-d; end of input]
```

4.6 Redirection metacharacter summary


```
STDOUT
........
>   FILE        Write to the file
1>  FILE        Same, using explicit file descriptor number
>>  FILE        Append to the file
1>> FILE        Same

STDERR
........
2>  FILE        Redirect to the file
2>> FILE        Append to the file

STDOUT AND STDERR
..................
>&2             Redirect [fd 1] to fd 2; echo "test" >&2
2>&1 FILE       Redirect fd 2 to fd 1. (stderr comes from stdout)
&> FILE         Redirect both stdout and stdout to FILE

STDIN
.......
< FILE          Open FILE and send the contents to stdout

<< HERE         Read input until here-document is terminated. The
                word that terminates the input is the word
                after "<". An example:

                $ cat << 'EOF'
                > input line 1
                > input line 2
                >EOF
```

The above summarize the most commonly used metacharacters. For more information, see POSIX Appending Redirected Output, Bash manual section Redirections and Advanced Bash-Scripting Guide I/O Redirection

5.0 Process control metacharacters

In most cases, users execute commands interactively. That is, they enter a command which the shell then executes. When the program finishes, they enter another command. This is called *foreground* processing. Executing commands in the foreground means that you have to

wait for the previous process to finish before entering another one. Some commands take a long time to finish. For example, sorting a large file can take a while. Instead of waiting for a command to complete, you can tell the shell to execute a commands in the *background*, which disconnects the process from terminal and runs concurrently with any other command. Background processing makes it possible to execute long running processes while still working interactively because command run in the background was **disconnected** from the terminal, a background process cannot read input from keyboard (stdin is usually keyboard). A background process terminates with an input read **error** if the process attempts to read from keyboard. The background process's put is connected to the terminal. Remember to redirect the output as appropriate or it will interfere your terminal working.

5.1 Background execution (&)

```
command &
```

The kernel doesn't know anything about foreground or background processing. Instead, the shell controls how a process is executed. You can execute a command in the background by putting the *background* metacharacter, an ampersand (&) at the end of the command line. Command without that character run in the *foreground*.

```
$ sleep 15              << FOREGROUND
$ sleep 15 &            << BACKGROUND
[1] 2241
[1]+  Done  sleep 15
```

When you put a command into the background, the shell tells you the *process id number* or PID. This number enables you to keep track of the process with `ps(1)`:

```
$ sleep 15 &
[1] 2248

$ ps
  PID TTY          TIME CMD
17598 pts/0    00:00:00 sleep
17599 pts/0    00:00:00 ps
```

Exercise: The plain `ps` command lists only your processes, but in a multitasking environment there are lot of processes running at the same time. Look at the manual page of the `ps` command and find **three** options that you could use. Try also following example, what do the options

mean

```
$ ps -ef
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

The process id number is the most commonly used identifier to control the process state. A Process can be **stopped**, **continued**, **stopped** and **interrupted** or a specific **signal** can be sent to it. The command to handle processes and various signals is `kill(1)`. Don't let the name mislead you: it is not purely used for terminating the process. The signals that are available are listed in separate manual page, like `signal(7)`. Look at the list of signals and those that are the most important ones. A Some Linux ps command has an option `-l` to list all signals.

Exercise: If kill(1) manual page does not list all available signals, search another the manual page which lists them. Follow the "SEE ALSO" at the end.

```
$ man man
```

Exercise: Try command `man -k signal`. You should see several manual pages for one command. E.g. kill(1) and kill(2). How would you request specific page?

The list of most important signals are listed below. In the `kill -SIGNAL` command where the the **SIG** prefix is left out from the name.

```
-------------------------------------------------------------------
SIGHUP      1   Request Hang up
SIGINT      2   Interrupts (like sending Control-C)
SIGKILL     9   Forced Kill. Leaves no time to clean DO NOT USE !!
SIGTERM     15  Termination signal. [default]
SIGUSR1     ..  User signal 1 - e.g. "re-read configuration file"
SIGUSR2     ..  User signal 2
SIGSTOP     ..  Stop process
SIGCONT     ..  Continue process
-------------------------------------------------------------------
```

An example how to terminate a running process with `-TERM` signal:

```
$ sleep 200 &
[1] 2260
$ kill -TERM 2260
[1]+ Terminated sleep 200
```


Exercise: Start a process in the background (long enough), start **another** terminal and kill the process running in the first terminal.

Exercise: Start a process in the background (long enough), and stop the process. View the process listing (ps) to confirm that it is in stopped state. Send a signal to let the process to continue.

In addition to using the process number, like 2260, you can refer to a process from the **current** terminal by using the shell's sequence number `%N`. There is also built-in shell command (you don't find a manual page) that lists all running processes in **current** terminal. Notice that you can't kill multiple processes with kill command, but you have to deal with them one at the time.

```
$ sleep 200 &
$ jobs
[1]- Running sleep 200 &
[2]+ Running sleep 100 &
$ kill %1
$ kill %2
```

Summary of the commands:

- `bg [jobspec]` Resume the suspended job jobspec in the background, as if it had been started with `&`. If jobspec is not supplied, the current job is used. The return status is zero unless it is run when job control is not enabled, or, when run with job control enabled, if jobspec was not found or jobspec specifies a job that was started without job control.

- `fg` '[jobspec]' Resume the job jobspec in the foreground and make it the current job. If jobspec is not supplied, the current job is used. The return status is that of the command placed into the foreground, or non-zero if run when job control is disabled or, when run with job control enabled, jobspec does not specify a valid job or jobspec specifies a job that was started without job control.

- `jobs [-lpnrs] [jobspec]` The first form lists the active jobs. The options have the following meanings: `-l` List process IDs in addition to the normal information. `-n` Display information only about jobs that have changed status since the user was last notified of their status. `-p` List only the process ID of the job's process group leader. `-r` Restrict output to running jobs. `-s` Restrict output to stopped jobs. If jobspec is given, output is restricted to information about that job. If jobspec is not supplied, the status of all jobs is listed. If the `-x` option is supplied, jobs replaces any jobspec found in command or arguments with the corresponding process group ID, and executes command, passing it arguments, returning its exit status.

- `kill [-ssigspec] [-nsignum] <jobspec|pid>` Send a signal specified by sigspec or signum to the process named by job specification jobspec or process ID pid. sigspec is either a signal name such as SIGINT (with or without the SIG prefix) or a signal number; signum is a signal number. If sigspec and signum are not present, SIGTERM is used. The `-l` option lists the signal names. If any arguments are supplied when `-l` is given, the names of the signals corresponding to the arguments are listed, and the return status is zero. exit_status is a number specifying a signal number or the exit status of a process terminated by a signal. The return status is zero if at least one signal was successfully sent, or non-zero if an error occurs or an invalid option is encountered.

5.2 Making pipelines (|)

```
command1 | command2 ...
```

Pipelines (see POSIX) connect ouutput from command1 to command2. It allows reading input without the need of temporary file. A pipeline connects processes by making that `stdout` of one command the `stdin` of the other. One command reads input directly from the output of another, so there is no need for using any temporary files. Examine that manual page of the `cut(1)` command that can be used to get pieces of the input. The most commonly used option is `-c` which works by counting the character positions.

```
$ date > file.tmp         # (1) using temporary file
$ cut -c1-3 file.tmp
Tue

$ date | cut -c1-3        # (2) pipeline, no temporary files
Tue
```

This command line is executed by two separate processes, one for `date(1)` and one for the `cut(1)` command. The processes are created in a special way that connects the first standard output with second standard input. There can be any number of pipelines involved. Pipelines show the flexibility of the design at its best. A process is independent of the data it uses, so it doesn't care where its input stream come from, even it it is another process. In fact a pipeline is special kind of redirection.

Exercise: Extract time from the `date(1)` listing

Further reading:

- Useless Use of Cat Award by Era Eriksson

5.3 Sequential lists (;)

```
command ; command ....
```

The sequential processing meta character (see POSIX), semicolon(`;`) enables executing two or mode commands, after another from the single command line.

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing


```
echo "The current date is" ; date
                           |
                           Separate two commands apart
```

The semicolon(;) does not provide any special processing. It simply groups multiple command to be executed sequentially. The above commands would have to be written separately without the semicolon, but the output wouldn't be exactly the same.

```
echo "The current date is"
date
```

Exercise: Have `ps(1)` listing stored in file `processes.txt`. Print Print date, empty line and file `process.txt` to the screen or to the file by using only one command line prompt. If your operating system does not support the **echo** "\n" to print newlines, just print a single dot with **echo** "." instead to separate the lines.

The use of semicolon, sequential processing, is rarely used interactively, but it can come in handy when combined with the other shell features. The semicolon is interpreted **before** most other metacharacters. This means that the commands to the left of the semicolon are separate from the commands to the right, regardless of what metacharacters appear in the each command. For example, there is only one `stdout` redirection below.

Exercise: what will appear in the file test.tmp? Another question: the meta character & means putting process to the background, but which process is put background here? Whole line?

```
$ date ; echo "test 2" ; echo "test 3" > test.tmp &
[1] 2214
$ cat test.tmp
```

5.4 Grouping commands () and {}

```
( compound-list )
{ compound-list; }
```


The shell lets you group commands together by using parentheses (see POSIX, Bash manual section Grouping Commands and Advanced Bash Scripting guide). Each set of parentheses runs the command in a *subshell*. The shell interprets parentheses **before** semicolons, which makes it possible to redirect multiple commands' output.

```
$ ( date ; who -a ) > system-status.txt

+-SHELL--------------------------------------+
|    +-SHELL--------+ stdout                 |
|  ( |  date ; who  | ) > system-status.txt |
|    +--------------+                        |
+--------------------------------------------+
```

Compare this to command line without the parentheses:

```
$ date ; who -a  > system-status.txt

+-SHELL---+   +-SHELL-----------------------------+
|  date   | ; |  who  > system-status.txt         |
+---------+   +-----------------------------------+
```

The difference is that in the first example, the output of the both commands appear in in the file system-status.txt. Your login shell's standard input, output and error streams were not redirected when you logged in the system, so the terminal is the default output destination. The *parentheses* work by creating a new sub shell that executes the command line listed inside the parentheses. Note that the `stdout` of the sub shell gets connected to the file, and not to the default terminal. Because the commands inside the parentheses are not redirected, their output goes to the sub shell's standard output.

Unlike other metacharacters, `()` must enclose an entire command line. As a result, the shell requires that a *command terminator* appear to the right of the closing parentheses. A command terminator is a **newline** metacharacter that terminates a command, such as ampersand (&), semicolon (;), pipe (|) and other characters.

Parentheses can be used to send the output of several commands down a pipeline. Commonly used line selection commands are `head(1)` and `tail(1)`.

The second grouping command differs from the first one in one importamt aspect: everything is run in *current* process environment. Notice that there **must** be terminating semicolong at the end of last command:

```
{ echo "one; echo "two"; } > file.txt
                          |
                          Semicolon required
```

The difference is substle but important especially in shell programs. Here is another example, where command "run-this" is called OR if it fails we run 2 more commands:

```
run-this || { echo "[ERROR]"; echo "Program failure, check PATH"; }
```

Exercise: Study following command sequence. What does it do?

```
$ { date ; ls -F; } | head -n 3
```

Exercise: Can we nest multiple sub shells? Do following commands work?

```
$ ( ( date ) )
$ ( sleep 5 ) ; ( cd .. ; pwd )
$ ( cd .. ; cd .. ; pwd )
```

Exercise: modify the above commands so, that you add the background execution meta character (&) after each lines. What commands are put to background?

5.5 Command substitution and back quotes (``)

```
 `command`
 $(command)
```

Note: The POSIX

```
defines syntax $(command) for better readability (Bash FAQ 82).
See also Bash manual section Command substitution
```

Let's take of an example code to create a simple log file with separate commands.


```
echo "Logged in" > $HOME/login.log
date >> $HOME/login.log

cat $HOME/login.log

Logged in
Mon Nov  6 23:23:07  2000
```

You can do the above, only in **one** command by using the *command substitution* metacharacters, represented by `$()`. Any command appearing inside parentheses gets executed and replaced by its `stdout` **before** any part of the command is evaluated again.

```
echo "logged in $(date)" > login.log
|               |
|               1) run this and get "Mon Nov  6 23:23:07  2000"
|
2) Now the shell sees a command
echo "logged in Mon Nov  6 23:23:07  2000" > login.log
```

The POSIX command substitution `$()` can also nest, while the this is very hard with the old backtick syntax. An example of nested use:

```
echo "Basename is: $(basename $(pwd) )"
                        |          |
                        |          return current path
                        |
                        extract last component
```

Command substitution can be used to insert a command's output in in-line redirection. Thus the above "Logged in" message and the date can be put to the same line with the following:

```
$ cat << EOF > login.log
> Logged in $(date)
> $(who -a | grep "username" )
> EOF
```


6.0 The filters

The multiprocessing capabilities are used in two ways. First the OS lets you create background processes, providing a way to run more than one command at the time. Second, multiprocessing makes command pipelines possible. Pipelines are very handy in everyday work and in creating applications. pipelines apply several processing steps sequentially to get the job done in a single efficient command line. To work, pipelines require **commands** that process their `stdin` (instead of reading from file) and write output to the `stdout`. The errors are reported in a pipeline to `stderr`, otherwise the errors were not visible during the process. If we use a command in a pipeline that does not read `stdin`, the pipeline is useless. The `cat(1)` however uses the pipeline.

```
ls | date         # nonsense
ls | cat          # works
```

There are many commands that can be used as *filters*. Filters are important part of the operating system. The more you know about the filters, the more productively you can use it form command line and from the *shell scripts*. Your ability to solve problems to using pipelines relies on your knowledge of the utility programs. Thus the first step in learning how to create pipelines is becoming familiar with the important tools. The documentation lists considerably more than 200 utility programs. One approach is to look at each one as a potential component in a pipeline. However, this is not usually productive. When you create applications (shell programs), here is brief start list of the most commonly used utilities.

```
awk         A text processing language that has a C/Perl like syntax
cat         Printing files (see command line options)
comm        Reports common files in two files (See diff)
cut         Select pieces of information from output
diff        Report difference of two files
grep        Search for regular expression from files
join        Combine lines from from two files into single line
nl          Number lines
nroff       Format manual pages or convert them to text
od          Octal of hexadecimal print of file
```

```
paste       Combine lines from two or more files
pr          Paginates files (header and footer)
sed         Edit the text stream (stream editor)
sort        Sort the lines in file
head        Print the read o file
tail        Print the tail of file
tee         Send the input to separate file(s)
tr          Translate characters (see sed)
uniq        Print unique lines
```


The important thing about filters is that they can be placed anywhere in a pipeline. This increases your processing options. Also filters implement general purpose tasks, such as searching or sorting. non-filter commands usually perform system tasks, like `ls(1)`, `umount(1)`, `date(1)`.

7.0 File name expansion and pattern matching (globbing)

See POSX standard Pattern Matching Notation.

7.1 Brackets to list multiple choices []

Listing lot of similar files through chapter 1 and 5, culd be done by listing all the files in the command line:

```
ls -l chapter.1 chapter.2 chapter.4 chapter.4 chapter.5
```

This is lot of typing. Because the filenames have common parts, shell provides a set of metacharacters that act as *wildcards* for file names. (A Windows user must forget what he knows about wildcards). For example the filenames listed are quite similar. In fact only the last character is different. What is needed here is *range* from 1-5 of possible characters that can be put to the end of filename. The shell provides a file name-matching range meta character, represented by `[]` the range of characters are listed inside brackets:

```
ls chapter.[12345]
```

The important point here is that a bracket matches only **one** character. If you need multiple numbers 10-29, you would have to write:

```
ls [12][1234567890]
```

It is important to understand how the shell interprets file name-matching metacharacters. The entire purpose of file name-matching metacharacters is to ask the shell to alter the command line by expanding a file name pattern into all the file names that match the pattern **before** the command line is executed. If you enter the above command, the shell actually converts that command line to the original equivalent before the command is executed.

```
ls -l chapter.1 chapter.2 chapter.4 chapter.4 chapter.5
```

Then it creates a process running the `ls(1)` command sending it the five file names as arguments. The `ls(1)` command does not know whether the original command line contained file name metacharacters, because all it sees is the result of the expansion. If the directory contains only two chapters the line is translated into:

```
ls -l chapter.1 chapter.2
```

because the shell can find only two file names that match the pattern. Note that the files are listed in alphabetical order. What happens if file name pattern does not identify any files in the current directory? The shell assumes that the argument is to be passed **as is** to the command being executed. This can happen if you are in a wrong directory and use pattern that does not match any file names.

```
$ ls chapter.[abc]
ls: chapter.[abc]: No such file or directory
```

If the range being matched is sequential, the range can be expressed using the dash(-) character like [1-5]. For a range to be valid, the characters must be lexicographically following each other: range [5-1] is therefore invalid. The order or repetition of the characters inside the range does not matter. All three above are identical:

```
ls chapter[12345]
ls chapter[1-45]
ls chapter[541-4]
```

The shell also has an *exclusion* range meta character, represented by the an exclamation point inside brackets [!]. Assuming that the chapter files have only numeric extension, the pattern chapter.[!5] matches all the files, but not the number 5. Although the [!5] is shorter way of writing

[1-46-9], there is more to it. The "!" actually says that **any** character can be accepted as long as the character is not 5, so the listing might produce result:

```
chapter.a
chapter.b
chapter.#
chapter.-
```


7.2 Question mark for one character (?)

In addition to specifying a range of characters, another shell meta character, question mark (?) requires a single character. the pattern

```
ls chapter.???
```

would match all chapter files that have three letter extension, but not those files that have none, one or two letter extensions.

7.3 Asterisk wildcard (*)

The most general file name-matching meta character is (*) which matches **zero** or **more** of any character. To match every file name that starts with "chapter", use `chapter.*`, which also matches file without any extension. Depending on the files in the current directory, any of the following might also produce same listing:

```
ls c*
ls ch*
ls chap*
ls chapter*
ls *              # same, but DIFFERENT
ls                # same, but DIFFERENT
```

When used by itself, the (*) matches all the file names in the current directory. Note that shell expands the star (*), not the `ls(1)` command, while it can be called without arguments. The problem with the star is that it may expand to huge number of files and shell do have the **argument count limit** of some thousand characters. If this exceeds, the shell gives and expansion error.

All the previous examples show the file name-matching meta character appearing at the end of the name. This placement is not required. File name-matching metacharacters can appear anywhere in a string or command line. Here is some shell file name patters:


```
ls *file
ls *file*
ls [abc]*
ls ???
ls ???*
ls *[!23]
```

Filename matching metacharacters can also appear in a larger path name. For example to find all the files names `tmp` in the first level sub directories in your home directory:

```
$ ls /home/peter/*/tmp
/home/peter/code/tmp
/home/peter/doc/tmp
/home/peter/tmp/tmp
/home/peter/txt/tmp
```

7.4 Backslash quoting (\) and single quoting (') (advanced)

The file system can let a file name contain **any** character, except the forward slash (/), which is used for directory names. As a result, a file name can contain control characters, spaces, tabs, even newlines and backspaces - any character except slash. It is perfectly legal that a filename contains shell metacharacters, but this usually leads to troubles. For example, we can create a file name containing a star (*), which is not possible in Windows.

```
echo "test" > test.\*
ls -l
```

The shell contains a separate set of metacharacters that remove the special meaning of other shell metacharacters. When a character loses its meta-meaning, it is said to be *quoted*. Shell offers backslash (\) to quote any single character and in **single quotes** A command like `cat file1&2` would result several errors, quoting the ampersand with backslash returns the original file name. As before, the shell removes the quoting metacharacters from the command line before executing it.

```
echo "test" > file1\&2
```

```
cat  file1\&2

echo "invisible" > \ .txt
cat \ .txt
```

Exercise: how would each of these commands fail?

```
cat msg&mail>tom
cat ideas;ch.[1-3]
cat memo.vp(sales)
```

The single quotes are shorthand way of putting a backslash in from of every character. A command line "cat file1\<-\>file2" might work, but you could also write "cat 'file<->file2". The shell does not look inside single quotes, but passes everything between the quotes on to the commands as a single argument, even the spaces inside the argument. Similar quoting appears in inside the range [*] where the star is treated as normal character.

```
ls '\ .txt' 'space in name.txt'
```

7.5 Removing funny files (advanced)

One of the most commonly asked questions by new users is "How do I delete file in my home directory. I've tried everything.". Take for example a file that starts with dash. The problem with the dash is that many programs interpret it as an *option*.There is couple of simple techniques that you can try. You can always refer to the file with relative pathname.

```
echo > \-
rm ./-
```

Another way is to supply an empty option argument, if the `rm(1)` command supports it:

```
$ rm - -
```


or

```
$ rm -- -
```

If you have an file that has control characters in it, you can use the `-i` option and an asterisk, which gives you the option of removing each file in directory - even those that you cannot type:

```
$ rm -i *
rm: remove - (y/n) ?
```

A great way to discover files with control characters in them is to use the `-q` option in the list command. You can alias the ls command to be `ls -q` and files that contain control characters will then appear with question marks. **Note:** the `C-vC-a` notation means that you must press key `Control-v` followed by `Control-a`

```
echo > C-vC-a
ls -q
```

8.0 File archiving commands

8.1 Compress and Uncompress

```
compress [options] file ...
uncompress [options] file ...
```

The old traditional compression method is `compress(1)` with equivalent `uncompress(1)`, but almost nobody uses it. The compression ratio in other programs are much more better. The file extension is *.Z*, note the big Z letter, there also exists little *.z*, but that is handled by seldom used `pack(1)` and `unpack(1)` commands.

> **Note:** This information is "nice to know", but in general you can forget the *.Z* compression. Almost all Open Source projects use either **gzip** or more efficient **bzip2** and **lzma** compression. Despite the popularity of Windows originated **zip**, it has not gained much

popularity in Linux.

```
$ compress file.txt
$ ls file*

file.txt.Z

$ uncompress *.Z
$ ls file*

file.txt
```


8.2 GNU gzip and bzip2

```
gzip [options] file ...
bgzip2 [options] file ...
```

All Linux operating systems now have have `gzip(1)`, which belongs to GNU binutils distribution. The gzip has an option to adjust compression ratio between 1 .. 9, where `-9` is same as `--best` compression.

```
$ gzip --best file.txt
$ ls file*
file.txt.gz
```

There are many options in `gzip(1)`, check the manual pages for full listing. Here are some of the most used ones:

```
-N      Level of compression 1..9. The default compression is 6
-c      Send output to STDOUT
-d      decompress (open the file)
-r      recursive
-t      test integrity of archived file
-v      verbose
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

Exercise: There is also program called **gunzip** in the system. Examine the file size to **gzip**. Do you come any conclusions?

`gunzip(1)` takes a list of files on its command line and replaces each file whose name ends with *.gz*, *.z*, *.Z* and which begins with the correct magic number with an uncompressed file **without** the original extension. It also recognizes the special extensions *.tgz* and *.taz* as shorthands for *.tar.gz* and *.tar.Z* respectively.

There is also a more effective compression program bzip2(1) which uses similar options to `gzip(1)`. See also LZMA compression which under Linux are available in command `xz(1)` (from package XZ utils).

```
$ bzip2 --best package.tar
$ ls package*
package.tar.bz2

$ bzip2 -d package.tar.bz2
$ ls package*
package.tar
```

8.3 Tar, tape archiver

```
tar -x[options] archive [file ...]      # extract
tar -c[options] archive <file|dir> ...  # create
tar -t[options] archive                 # view content
```

The `gzip(1)` can compress only *one file* at a time and it is usually accompanied with the tape archive command `tar(1)`, which simply wraps *multiple* files into one package. Note that this approach differs greatly from the zip compression method where every file in individually compressed, whereas the whole tar package is compressed and you cannot extract single files, unless the whole package is decompressed first. (this is also the reason why *.tar.gz* usually compresses better than *.zip*).

> **Note:** Remember that: **1)** `tar(1)` command does not add extension *.tar* automatically. **2)** In recursive mode, you must not create archive to the **same** directory which is being tarred up, because then the tar would include "itself".


```
$ cd $HOME
$ tar -cvf package.tar tmp/
                  ====
                  You have to specify extension ".tar"

$ gzip --best package.tar
$ ls -l package*
package.tar.gz
```

To see the contents of the compressed tar package, use the `-t` switch. There are several tar commands: one is included in GNU tools, and must be installed separately. and the other is systems default tar command. Which one is active, is easily checked by running tar with the `--help` command line option. If you do have GNU tar, then you have much more powerful archiver with multiple options.

```
$ tar --help
GNU `tar' ...

-c        create new archive
-f NAME   use archive file NAME
-j        bgzip2 compress support (GNU tar only)
-k        keep existing files
-t        list contents of archive
-v        verbose
-x        extract
-z        gzip compress support (GNU tar only)
-O        extract to STDOUT (GNU tar only)
```

Here is an example how to uncompress and unpack (=extract `-x`) a tar file with standard tools in any Operating System. Be very careful that the `-f` and the `stdin` marker dash (-) are the last characters:

```
$ gzip -dc package.tar.gz | tar -xvf -
                                  ====
                                  This here actually says
                                  "Read from FILE -", where
                                  dash(-) file is stdin
```

If you had GNU tar (sometimes compiled with name `gtar(1)`), it would understand compression with command line switch `-z` and you could write simply:

```
$ tar -zxvf package.tar.gz
```


Exercise: Make a complete backup of you *~/tmp* directory. Since the archive may become big, store the archive in system */tmp* directory.

To list contents of the archive use the `-t` option:

```
$ gzip -dc package.tar.gz | tar -tvf -
```

Exercise: How would you do that with GNU tar; in a single tar(1) command alone?

To extract only one file or files from the archive, list them in place of the dash (-) `stdin` marker.

```
$ gzip -dc package.tar.gz | tar -xvf - file.txt
                                      ========
                                      Extract this file only
```

8.4 Other compressing programs: zip

```
zip [options] archive <file|dir> ...
unzip [options] archive
```

The `zip(1)` and `unzip(1)` are *sperate* programs (whereas `gzip(1)` and `bzip2(1)` include both compressor and uncompressor). They are are not necessarily installed by default. The most commonly used command lines options for `zip(1)` and `unzip(1)` are:

```
-N      Compression level 1..9. Defaul is "-6"
-@      Read the list of files from STDIN (can be used as pipe)
-d      Delete files from archive
-l      Translate line endings LF --> CR LF
```

```
-ll     Translate CR LF --> LF
-m      Move the files to archive
-r      recurse to subdirectories
-T      Test integrity of archive
```


Exercise: Make a complete backup of you *~/tmp* directory using zip(1). Since the archive may become big, store the archive in system */tmp* directory.

8.5 Handlind compressed files

When the file is compressed, it cannot be read unless it is decompressed first. There are many utilities that invoke the decompression program and then pass the result to the regular command. Here are some common commands:

> **Note:** The implementation of these z-commands may differ from environment to another. E.g. compared to Linux, the SunOS/Solaris `zcat(1) 'only displays *.Z* files, whereas the Linux version is more effective and can list several archive formats, including *.gz* and *.bz2*.

- `zcmp(1)`, Compare compressed files with `cmp(1)`.
- `zcat(1)`, print the compressed file like `cat(1)`.
- `zdiff(1)`, compare files using `diff(1)`
- `zgrep(1)`, search using regular expressions like `grep(1)`

Exercise: Locate the **zcat** program in your systems and look at the code how they have been written.

Exercise: what is command `zless(1)`?

9.0 The file system and security

9.1 Password security

> **Note:** stand alone systems use `/etc/passwd` or `/etc/shadow`, but networked systems store user information in separate NIS [see ypcat(1)] or LDAP databases [see ldaplist(1)].

There is a file called `/etc/passwd` that contains all the information the system needs to know about each user, including the passwords (unless the passwords are shadowed). The passwords in the file are encrypted and with command `crypt(1)`. A Typical excerpt from the file looks like this:


```
$ head /etc/passwd

...
mark:x:99:10:Foo Bar:/home/bar:/bin/bash
(1) (2)(3)(4) (5)    (6)       (7)
```

The fields are:

- (1) *Account* name. The login name. This is usually restricted to 8 characters on most systems and must consists of alphanumeric or underlines characters.
- (2) *Password*. If the system uses shadow passwords, you might see only a "x", "*" or "!" in place of the password here. Any punctuation makes the password invalid. The string is created using the `passwd(1)` or `makekey(1)` command.
- (3) *User identification* (UID). This number is automatically assigned by the user creation programs, like `useradd(1)` or `adduser(1)`
- (4) *Group identification* (GID). This number is automatically assigned by the group creation programs, like `groupadd(1)` or `addgroup(1)` The names of the groups are listed in file /etc/group
- (5) *Comment* field, usually name, phone, company internal address. a `finger(1)` command uses this information.
- (6) *Home* directory. If this is not found, the user is placed on root directory "/".
- (7) *Startup program*, usually shell name or /bin/false to prevent login

Whenever person logs in, the password typed at the terminal is encrypted and checked against the password stored in this file. If they match, he is allowed to enter on the system and if they don't, a message "Login incorrect" is shown and new try is offered. To change the current password, you can't edit `/etc/passwd` file directly. There is a separate command `passwd(1)` to change active password.

```
$ passwd
Changing password for Leo
Old Password: xxxxxx
New Password: xxxxxx
Re-enter new password: xxxxx
$
```

If you want to keep your login secure, you should choose non-obvious passwords. first and last names, initials, birth dates and the like are poor passwords. Words from dictionaries can be broken in matter of weeks with programs like `crack` or Debian `john(1)` "the ripper". A good password are those that are at least 6 characters long and which contain some non-alphabet characters. One good method for picking in the password is to select a *pattern* from a keyboard, any random pattern or sequence of keys, that don't actually mean anything. It is also advised

that you don't use the same password for logins on different machines. If the password is compromised, then the intruder gains access to all hosts. A good system expires passwords and forces to renew them periodically.

9.2 File system

9.2.1 Files

From the simplest perspective, everything visible to the user can be represented as a file in the file system - including the processes and the network connections. Almost all of these items are represented as files each having at least one name, an owner, access rights and other attributes. This information is actually stored in the file system in an *inode* (index node) the basic file system entry. An inode stores

9.2.2 Directories

One special kind of entry in the file system is the directory. A directory is nothing more than a simple list of names and *inode* numbers. A name can consist of string of characters with the exception of a "/" character and the "null" (0x00) character. There is a limit to the length of these strings, but it is usually a quite long: 1024 or longer on many modern systems. These strings are the names of files, directories and the other objects stored in the file system. Each name can contain control characters, line feeds and other characters.

> *Associated with each name is a numeric pointer that is actually an index on disk for an inode. An inode contains information about an individual entry in the file system.*

Nothing else is contained in the directory other than the names and the inode numbers. No protection information is stored there, nor owner names, nor data. The directory is a very simple relational database that maps names to inode numbers. No restriction on how many names can point to the same inode exists either. A directory may have 2, 5, or 50 names that each have the same inode number.

9.2.3 Inodes

For each object in the file system, the administrative information is stored in a structure known as an inode. Inodes reside on disk and do not have names. Instead they have indices (numbers) indicating their positions in the array of inodes. Each inode generally contains:

- The location of the item's content on the disk
- The item's type (file, directory, symlink)
- The item's size in bytes, if applicable
- last modification time of inode (ctime)
- File content's modification time (mtime)
- Last access time (atime) like read() or exec()
- A reference count: the number of names the file has
- The file's owner (UID)
- The file's group (GID)
- The file's file permissions (mode bits)

The last three pieces of information, stored for each item and coupled with UID/GID information about executing processes are the fundamental data that OS uses for practically all system security.

Other information can also be stored in the inode, depending on the particular version of OS involved. some systems may also have other nodes such as *vnodes* and *cnodes*. These are simply extensions to the inode concept to support foreign files, RAID (Redundant array of inexpensive disks)

9.2.4 Inodes; hard links and soft links

In like manner, several directories may have names that associate to the same inode. These are known as *hard links* to the file. There is no way telling which link was first created. This also means that you do not actually delete a file with commands like `rm(1)`. instead you *unlink* the name - You remove the connection between the filename in a directory and the inode number. If another link still exists, the file will continue to exist on disk. After the last link is removed and the file is closed, the kernel will reclaim the storage because there is no longer a method for user to access it. you can create links with command `ln(1)`

Exercise: Create file and two links, where the fist one is **hard** link and the other is **soft** link. What is the difference? Watch the difference when you remove the **original** file and then access it through link files.

```
echo "123" > test.tmp
ln test.tmp hard.link
ln -s test.tmp soft.link
ls -la *link
cat hard.link
cat soft.link
ls -la *link
rm  test.tmp
ls -la *link
cat hard.link
cat soft.link
```

Every normal directory has two names always present. One entry is for "." (dot) and this is associated with the inode for the directory itself. The second is "..", which points to the parent of this directory; the directory next closest to the root in the tree-structured file system. The exception is the root directory itself, named "/". in the root directory ".." is equal to ".".

9.2.5 Block vs. character devices

Have a look at your system devices in `/dev/` directory with `ls(1)` command.

Most devices are referenced as *character devices*. These are known as *raw devices*. the reason for the name "raw device" is because that is what you get - raw access to the device. You must make your read and write calls to the device file in the natural transfer units. Thus you probably read and write single characters at a time to a terminal device, but you need to read and write sectors to a disk device. Attempts to read fewer (or more) bytes than the natural **block size** results in an error, because the raw device does not work that way.

When accessing the file system, we often want to read or write only the next few bytes of a file at a time. if we used the raw device, it would mean that to write few bytes to a file, we would need to read in the whole sector off the disk containing those bytes, modify the one we want to

write, and then write the whole sector back out. Now consider every user doing as they update each file. That would be lot of disk traffic!

The solution is to make efficient use of caching. *Block devices* are cached versions of character devices. When we make a reference to a few bytes of the block device , the kernel reads the corresponding sector into a buffer in memory, and then copies the characters out of the buffer that we wanted. The next time we referenced the same sector, to read from or write to, the access goes to the cached version in memory. if we have enough memory, most of the files we will access can be kept in buffers, resulting in much better performance.

There is drawback to block devices, however. If the file system crashes before modified buffers get written back out to disk, the changes our programs made won't be there when system reboots. Thus we need to periodically **flush** the modified buffers to disk. This is effectively what system call sync() does: schedule the buffers to be flushed to disk. Most systems have a `syn` or `fsflush` daemon that issues a sync() call every 30 or 60 second to make sure the disk is mostly up to date. if the system goes down between sync() calls, we need to run a programs such as `fsck(1)` or `checkfsys(1)` to make certain that no directories whose buffers were in memory were left in an inconsistent state.

9.2.6 Current directory and path

Every item in the file system with the name can be specified with a *pathname*. The word pathname is appropriate because a pathname represents the path to the entry from the root of the file system. By following this path, the system can find the inode of the referenced entry.

Pathnames can be **absolute** or **relative**. Absolute path names always start at the root and thus always begin with the "/", representing the root directory. A pathname like /home/joe/bin/test.sh represents a pathname to an item starting at the root directory.

A relative pathname always starts interpretation from the current directory of the process referencing the item. This concept implies that every process has associated with it a *current directory* (cwd, see `pwd(1)`). Each process inherits its current directory from a parent process after a fork. The current directory is initialized from the user record in the `/etc/passwd`, the home directory. The current directory is then updated every time the process performs a change directory operation with programmatic chdir() call or with user call `cd`. Relative pathnames also imply that the current directory is at the front of the given pathname. Thus after executing a `cd /usr`, the relative pathname `lib/makekey` would actually be referencing the pathname `/usr/lib/makekey`. Note that any pathname that does not start with "/" must be relative.

9.2.7 Using the ls command

You can use the `ls(1)` command to list all of the files in a directory. This command has many options and the most usually used are `-la` for long listing, `-F` for file type information and if you want tosort by *date*, then add `-t`. See manual page for full details.

```
$ ls -la
total 950
drwxr-xr-x  36 foo     users  3072 Oct 18 11:09 .
lrwxrwxrwx   1 foo     users    28 Aug 12 20:22 test.txt -> t
drwxr-xr-x  19 foo     users  1024 Aug 26 18:55 ..
-rw-r--r--   1 foo     users   118 Oct 15 10:27 t
-rw-------   1 foo     users     0 Jan 22  2000 .pwd.lock
drwxr-xr-x   3 foo     users  1024 Jan 22  2000 CORBA
drwxr-xr-x   2 foo     users  1024 Aug  7 19:17 CVS
-rw-r--r--   1 foo     users  1165 May 31 13:59 ChangeLog
```

```
-----------------------------------------------------------
|           | |       |      |
Permissions  | UID     GID    File size
             |
             The number of hard links
```


9.3 File permissions

The file permissions on each line of the `ls(1)` command tell you what the file is and what kind of file access (that is, ability to read, write or execute) is granted to various users on your system. Here are two examples of file permissions:

```
-rw-------
drwxr-xr-x
 U..G..O..
```

The first character in the mode field indicates the type of file:

- - Plain file
- *d* Directory
- *c* Character device (tty or printer /dev/tty /dev/pty)
- *b* Block device (usually disk /dev/hda)
- *l* Symbolic link
- *s* Socket
- *p* FIFO, see `mknod(1)`

Exercise: To demonstrate the /dev/tty (your terminal) device, run command `tty(1)` and try to write to it with:

```
echo hello there > $(tty)           # See tty(1) command
```

Exercise: Another way to talk to another user, is to find out terminal where he is working on. Run command `who -a | grep <USER>` to see the USER's connections. Next see the most active terminal which the on with dot(.) character. Now, try to write to him with command `write(1)`.

```
who --idle | grep foo     # Where are foo user's terminals?
mesg y                    # allow messages

write <terminal>          # send message; see "who -a"

  Hello there             # type the message
  C-d                     # <end>
```


The rest of the characters are taken in groups to indicate *who* on your computer can *do what* with the file. There are three kind of permissions.

- *r* Read. You are allowed to **open**, **read** and **copy** the file.
- *w* Write. You can **overwrite** the file or **modify** the content.
- *x* Execute. The file can be **run** by typing it's name. The first two bytes are assumed to be MAGIC NUMBERS indicating the nature of the file. The special two byte **shebang**, #! means that the it is an executable script of some kind. Anything with an unknown value is assumed to be a `sh(1)` shell script.

These values can be specified to three groups

```
-rw-------
drwxr-xr-x
 ---   ---
 U  G  O
```

- *user* The file's owner /etc/passwd
- *group* Users who are in the group file /etc/group
- *other* Everybody else on the system

File permissions apply to devices, named sockets, and FIFOSs exactly as they do for regular files. If you have write access, you can write information to the file or other objects. If you have read access, you can read from it and if you don't, you are out of luck. **NOTE:** file permissions do *not* apply to symbolic links. Whether or not you can read the file pointed by the link depends on the file's permissions, not the link's. In fact a symbolic link is almost always created with same sets of file permissions *lrwxrwxrwx* (or mode 0777) and are ignored by the operating system.

```
$ ln -s test.tmp link-test.tmp
$ ls -la
```

```
total 6
drwxrwxr-x   2 foo   users       1024 Nov 19 19:12 .
drwxr-xr-x   4 foo   users       1024 Nov 19 18:42 ..
-rw-r--r--   1 foo   users       2738 Nov 19 18:46 ddir.pl
lrwxrwxrwx   1 foo   users          8 Nov 19 19:12 link-test.tmp -> test.tmp
-rw-rw-r--   1 foo   users          1 Nov 19 18:46 test.tmp
```


- You can have an *execute* access without having a read access. In such case you can run a program without reading it. This ability is useful in case you wish to hide the function of a program. Another use is to allow people to execute a program without letting them make a copy of it.
- If you have a *read* access but not execute access, you can then make a copy of the file and run it yourself. The file will then run with your UID, not the original author's.
- An executable shell script must have both its *read* and *execute* bits set to allow people to run it. This is natural, because shell must read and interpret the lines before they are executed. For binary files this is not needed and plain execute but will allow running the file.

9.4 Changing permission (chmod)

```
chmod [-Rfv] {[augo][+-=][rwxst] ...} <file|dir> ...
```

Many systems have had security breaches because their file permissions are not properly set. The permissions determine who can read and modify the information stored in the files, they are the primary method of protecting the data. The file's permissions are changed with chmod(1) command. You can change the file's permissions only if you are the file's owner or superuser (also called "root" user). In its simplest form, the chmod command lets you specify which of the file's permissions you wish to change. This usage is called *symbolic mode*.

This command changes the permissions of many files or only a single file, possibly recursing to all directories below if `-R` switch is used. The letters augo are the primary tools to specify whose prviledges are to be modified:

```
Access groups                Change mode
-------------------------    ---------------------------------
a   Change all priviledges   +   Add
u   Change user              -   remove
g   Change group             =   Set all  (clear others)
o   Change other

Priviledge                   Options
-------------------------    ---------------------------------
r   Read                     -R  Recursive
```

```
w   Write                       -f  force, ignore errors
x   Execute                     -v  verbose
s   SUID or SGID
t   Sticky bit (old)
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

Exercise: What do following commands do?

```
$ chmod u+rw    myfile.txt
$ chmod og-rwx  myfile.txt
$ chmod a=r     myfile.txt
```

The file permissions can also expressed in octal modes where the values are calculated from right to left as shown below:

```
USER    GROUP   OTHER           Higher bits (x)
---------------------           --------------------------------
r w x   r w x   r w x           4000    SUID    chmod u+s
4       4       4               2000    SGID    chmod g+s
  2       2       2             1000    sticky  chmod +t
    1       1       1
----------------------
    7       7       7
```

Exercise: The most used file permissions can be set using the numerical values. Find out what permissions are set with following commands:

```
$ chmod 777 file
$ chmod 755 file
$ chmod 700 file
$ chmod 600 file
$ chmod 644 file
```

9.5 Umask

```
umask [<octal value>]
```

The *umask* (User file creation mode mask) is a four-digit octal number that is used to determine the file permissions for newly created files. Every process has its own umask, inherited from its parent process. The umask specifies the permissions you do not want to give by default to newly created files and directories. umask works doing the bitwise AND with the bitwise complement of the umask. Bits that are set in the umask correspond to permissions that are not assigned to newly created files. Normally the default umask value is set in `~/.login` or `~/.profile` or in system wide `/etc/profile`. It is usually the first commands in the file, because anything prior it will have the system default, possibly unsafe value. To see the current value, run

```
$ umask
```

The explanation of various values

```
0   All permissions granted
1   no execute
2   no write
3   no write or execute
4   no read
5   no read or execute
6   no read or write
7   no permissions at all
```

The most common umask values are presented below. This setting is usually stored in variable UMASK at `~/.bash_profile`. (see environ(7) manual page).

```
002     others: no write
022     group + others: no write
027     group: no write; others: none
077     group + others: none
```

9.6 Directory permissions

Unlike many other operating systems, Linux stores the contents of directories in ordinary files. These files are similar to other files, but they are specifically marked so that they can only be modified by the operating system. As with other files, directories have full complement of security attributes: ownership, group and permissions. But because the directories are interpreted is a special way by the file system, the permissions have special meanings.

- *r* Read. E.g. ls(1) can **list** the content
- *w* Write. You can **add**, **rename** and **remove** files If user has write permission to directory, he overrides any permissions set for individual files.
- *x* Execute. You can search the directory with `find(1)` or `cd` to it. This gives `stat(1)` rights, e.g. long listing of `ls(1)` Pure *+x* permission allows user to list the file only if he knows the exact filename.

Notes on the directory permissions

- You can't access the files, if the *+x* flag is not set
- Directory permissions override file permissions. You can "lock" the directory from other by removing "ug-rwx" and nobody except you can access the files.
- *-r* but *+x* gives access to individual files, only if you know the full name. (You can't list the directory)
- To *remove/unlink* file from directory you need *+wx*, regardless of the file permissions

9.7 SUID and SGID bits (advanced)

Sometimes, unprivileged users must be able to accomplish tasks that require privileges. An example is the `passwd(1)` program. Changing user's password requires modifying the password file /etc/passwd. Likewise, the `mail(1)` program requires rights to add mail messages to the mailbox in /var/mail/$USER. To get around these problems, programs are allowed to be endowed with privileges. Processes executing these programs can assume another UID or GID when they are running. A program that changes its UID is called SUID program (set UID), a program that modifies its GID, is called SGID. when SUID program is run, its effective UID becomes that of the owner of the file, rather than the user who is running it.The *x* flag in the file listing will contan the marker *S*:

```
---S------      SUID    Effectve ID is set to program owner
------S---      SGID    The group is changed to program's GID
---------t      sticky  Obsolete for files. In directory, allow
                        creating each file as "user" file.
```

A typical program which has SUID bit set is the `su(1)`, substitute user.

```
$ ls -l /bin/su
-rwsr-xr-x   1 root     root        13208 Apr 13  1999 /bin/su
```


9.7.1 SGID and Sticky bits on directories

Although the sticky bit were originally intended for use only with programs, Berkeley, Sun OS, Solaris and other operating systems also use these bits to change the behavior of directories. For example to set the mode of the `/tmp` on a system so any user can create or delete her own files, but cannot delete other's file, you set the sticky bit as root. On older systems setting the SGID bit resulted the same behavior.

```
$ chmod 1777 /tmp
```

9.7.2 The origin of sticky

A very long time ago, Linu ran on machines with much less memory than today; 64 kilobytes, for instance. This amount of memory was expected to contain a copy of the operating system, I/O buffers and running programs. This memory often was not sufficient when there were several large programs running at the same time. To make most of the limited memory, the OS *swapped* process from to and from secondary storage (disk) as their turns at the CPU ended. When a program was started, the system would determine the amount of storage that might ultimately be needed for te program. Its stack and all its data. It then allocated a set of blocks on the swap portion of the disk or drum attached to the system. (Many systems still have a `/dev/swap` or a `swapper` process that is a holdover from these times). Each time the process got a turn from the scheduler, the OS would *swap in* the program and data, if needed, execute for a while, and then *swap out* the memory copy if the space was needed for the next process. When the process exited or *exec* another program, the swap space was reclaimed for use elsewhere. If there was not enough swap space to hold the process's memory image, the user got "No memory error".

Obviously, this is great deal of I/O traffic that could slow computation, so one of the eventual steps was development of compiler technology that constructed executable files with two parts: **pure code** that would not change and everything else. These were indicated with a special magic number in the header inside the file. When the program was first executed, the program and data were copied to their swap space on disk first, then brought into memory to execute. However, when time comes to *swap out*, the code portions were not written to disk - they would not have changed from what was already on the disk! This change was a big savings.

The next obvious step was to stop some of that extra disk-to-disk copying at start-up time. Programs that were run frequently - such as `cc(1)` could share the same program pages. Furthermore, if no copy was currently running, we could expect another one to be run soon. Therefore keeping the pages in the memory and on the swap partition, even while we weren't using them made sense. The *sticky bit* was added to mark those programs as worth saving. since those times, larger memories and better memory management methods have largely removed the original need for the sticky bit.

9.8 File encryption (advanced)

Encryption programs available are usually `crypt(1)`, *des* (data encrption standard) and *pgp* (pretty good privacy). Each of these programs offer increasing amounts of security. The `crypt(1)` is an encryption program that is included as a standard part of the operating system. It is a very simple encryption program that is easily broken. The `crypt(1)` uses a simplified simulation of the Enigma encryption machine. For several years messages have been able to break thanks to program developed in 1986 by Robert Baldwin at the MIT laboratory for computer science. Crypt Breakers Workbench (CBW) decrypts text files within matters of minutes with minimal help from the user. CBW has been widely distributed and as a result files encrypted with crypt should not be considered secure.

9.8.1 Ways of improving the security of crypt

You may not have other encryption programs available to you, but you can take a few simple precautions that will increase the chance that your files will be harder to break. the following activities will defeat the CBW's automatic crypt-breaking activities:

- Encrypt the file multiple times, using different keys at each stage.
- Compress your files before encrypting them. compressing a file alters the information - the plain text ascii programs such as CBW use to determine when they have correctly assembled part of the encryption key. If your message does not encrypt into plain text, CBW will not determined when it has correctly decrypted your message.
- Do not use the `compress(1)` or `pack(1)` which leave 3-byte detectable header to tell that the file was compressed.
- Pick extension that only you know will indicate encryption and possible compressions. Alternatively, remove extensions altogether.

An Example: traditional crypting (which can be easily opened with crack tools)

```
$ crypt mykey < file.txt > file.txt.crypt   # crypt
$ crypt mykey < file.txt.crypt > file.txt   # decrypt
```

An example: more secure crypting

```
$ gzip file.txt
$ crypt mykey < file.txt.gz > file.txt.gc
```

9.8.2 PGP, Pretty good privacy

Note: GPG (and GPG) are very effective encryption programs that are far superior to the old crypt(1) program. Choose GPG with at least 2048 bit key size if need strong cryptography and value crossplatform Open Source solution.

In 1991, Phil Zimmerman wrote a program called PGP which performs both private key and public key cryptography. That program was subsequently released on the Internet and improved by numerous programmers, mostly outside of the United States. The original program uses version numbers 2.6.x and is not compatible with the new PGP keys generated by the Windows versions of 5.x or later. The original PGP is at <http://www.pgpi.org> however and commercial implementation at <http://www.pgp.com>.

For cross platform use, similar GNU project program that implements the PGP algorhitm is called `gpg(1)` (GNU privacy guard).

10.0 Grep command

```
egrep [OPTIONS] 'REGEXP' file [file ...]
ls | egrep [OPTIONS] 'REGEXP'
```

The grep(1) command and it's derivates, `egrep(1)` extended grep, `fgrep(1)` file grep, `zgrep(1)` for compressed files, is a powerful filter that searches input for lines that contain *pattern*, regular expression, and sends them to the *stdout*. Like most utilities, `grep(1)` recognizes options that further specifies what the command does. The `egrep(1)` is preferred, because it has more regular expression power than the older `grep(1)`. The regular expression syntax is explained later.

The basic options of egrep are:

- `-c` Print a count of matching lines
- `-e "REGEXP"` Use regular expression pattern
- `-i` Ignore case sensitivity during matching pattern.
- `-l` Print only files that contain matching lines.
- `-n` Add line number to show where match is
- `-v` Print all lines that do not match.

10.1 Egrep Regular expression tokens

10.1.1 The basic atoms

```
.       Any character, except newline character \n.
[a-z]   Character class, but only if it's between a-z. Another
        example could be [0-9] or combination of characters
        [a-z0-9!.?*] or for vowels [aeiouy]
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

[advanced] GNU grep also support "Named character classes" that are shorthand for range of charatcer. E.g.

```
a-zA-Z    =   [:alpha:]
```

But this not all there is to it. The *Named Classes* work even if the text being matched it not plain ASCII supposed that the `locale(1)` is set correctly in the operating system environment:

```
echo "äö" | grep "[[:alpha:]]"
```

Named Classes can be combined:

```
echo "abc123" | grep "[a-zA-Z0-9]"
                       ======
                       |
                       ========
echo "abc123" | grep "[[:alpha:][:digit:]]"
                      |                 |
                      start             end
                      of class          of class
```

See `grep(1)` manual page and topic "REGULAR EXPRESSIONS" for more information. The GNU grep known classes are:

```
Named class     Traditional character class (range)
----------      -----------------------------------
[:alnum:]       [a-zA-Z0-9]. A word. Also known as \w
[:alpha:]       [a-zA-Z]
[:cntrl:]       Control" characters. ASCII table: [\x00-\x1F\x7F]
[:digit:]       [0-9]
[:graph:]       Visible cracters. ASCII table: [\x21-\x7E]
[:lower:]       [a-z]
[:print:]       Visible characters. ASCII table: [\x20-\x7E]
[:punct:]       Punctuation: [!"#$%&'()*+,\./:;<=>?@[\\\]^_`{|}~-]
[:space:]       whitespace = space and tab characters
```

```
[:upper:]       [A-Z]
[:xdigit:]      [a-fA-F0-9]
```

10.1.2 Additional tokens

```
|       OR clause. "a|b"  "word-here|word-there"
()      Grouping. "(this-word|or-this)that-are-before-this"
```

10.1.3 Quantifiers

```
?       0 or 1, That is, a element is NOT required or EXACTLY 1.
*       0 or more. That is, a element is NOT required or MORE
+       1 or more. That is, a element is REQUIRED
{N}     N times exactly. Example: "[0-9]{4}", like in year 2000.
{N,}    N times or more.
{N,M}   N to M times exactly.
```

10.1.4 Anchors

```
^       Force match to start at the BEGINNING of line
$       Force match to start at the END of line
\<      Word boundary LEFT. Like "\<[a-z]" => "(!)this.(!)word"
\>      Word boundary RIGHT. Like "[a-z]\>" => "this(!).word(!)"
```

10.1.5 Examples

```
Joe     Find word "Joe" anywhere in the line.

^Joe    Match name "Joe" only at the beginning.
        "Joe is tall and Joe plays tennis", only the Joe at
        the beginning is found.
```

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing


```
car$    Match only at the end. Like "Mike's car and Joe's car".
        Only the last "car" is found. Without the dollar, the
        Mike's car would have been found.

        Mike's car          ok
        This car            ok
        By a car now!       not matched

.*      Match `any'(.) char 0 or more (eat everything on the line)
        THIS IS THE "WILDCARD" IN REGULAR EXPRESSION SYNTAX.
        Like if you would search prices for collowing
        products: pen, paper, chalk.

        egrep '(pen|paper|eraser).*[$][0-9]+

        pen's price is $1.24                    ok
        paper in the other hand is $0.25        ok
        For $0.50 you can buy a good eraser     nok

ab?c    match combination "abc" "ac", because `b' is not
        necessarily required (0 or 1 times)

ab+c    Match "abc" "abbc" "abbbc", as long as there is b's

ab.*c   Match anything between "ab" and character "c".
        Examples: "abc", "ab word c", "abbbc".

gr[ea]y

        Match "grey" or "gray". The [ea] lists characters that
        can exist in a position

[0-9]+  Match multiple number2 in a line.

        $200 dollars in cas                 ok
        The area is 3000 cubic inches       ok
        two times (2) two (2) = 2 x 2       ok
        There is no number here             nok

Joe +(Smith|Jones)
```

```
Match First name "Joe" followed by any number of spaces
and last name that can be either "Smith" or "Jones".

"Joe Smith has a bicycle"          ok
"Joe Jones has a motorcycle"       ok
"Joe Davis has a boat"             not matched
```


11.0 Sed command

```
sed [OPTIONS] [-e] 'INSTRUCTION' [-e 'INSTRUCTION' ...] file ...
sed [OPTIONS] -f SCRIPT.sed file ...
```

Command sed(1) is a (s)tream (ed)itor, which manipulates the data according given rules. The `ed(1)` was the most primitive and original editor and sed offered "batch" mode for all `ed(1)` commands.

Some most used ptions:

- `-f FILE` Read commands from file; typically ends in extension *.sed.
- `-e` "SED-EXPRESSION" Expression follows immediately. You can give this option *multiple* times.
- `-n` Do not print lines unless *p* command used.

The INSTRUCTION choices can be in format

```
SEARCH/INSTRUCTION/
|      |
|      What to do:
|      p = print
|      d = delete
|      s = substitute
|
Can be a regular expression /REGEXP/
or an ADDRESS notation: FROM_ADDRESS [, TO_ADDRESS]
```

11.1 Sed commands and options

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

11.1.1 The delete line command

(d)elete lines containing word "this-word"

```
/this-word/d
```

11.1.2 The print lines options

(p)rint all lines that match the regular expression. In fact, this where the `grep(1)` got it's name. The grep was followed by egrep, an extended and better grep with better regular expression syntax.

```
/RE/p
/RE/,/RE/p                  # Between RE..RE
2p                          # Print line 2
```

11.1.3 The substitution of text in the line

The command that sed uses most is the (s)ubstitute and the syntax is bit different:

```
[address]s/RE/replacement/[flag]
         |
         substitute command
```

For example if we delete (substitute with nothing) all words matching "this-word". The (g)lobal flag causes regular expression search to continue to the end of line, so all words on the line will be deleted.

```
s/this-word//g
```

Here we make couple of simple substitutions for all lines. Notice the (g) option.

```
s/Duke/Royal/g      Convert every word "Duke" to "Royal"
s/CA/California/g   Expand CA to full word
```

11.1.4 Address in substitute command

The [address] says, where the command does its work. It can be a) numeric address or b) special marker; like `$` which denotes the end of file c) regular expression to delimit the the commands to certain lines:

We refer to explicit lines by a number here:

```
 1s/CA/California/g      Do substitution only at line 1.
10s/CA/California/g      Do substitution only at line 10.
10,20s/CA/California/g   RANGE: lines from 10 to 20.
```

We can mix the number with the other address markers:

```
50,$s/CA/California/g    from line 50 to the end of file.
1,/^$/s/CA/California/g  from line 1 to next empty line.
```

We can also use only regular expression to delimit the line area. Here the commands is applied only between lines BEGIN and END. We don't know the line numbers, where this code is:

```
$ sed -e '/BEGIN/,/END/s/variable1/variable2/g' some.code

BEGIN
    line1;
    line2 variable1 = variable1 + variableX;
END
```

11.2 Sed and regular expressions

Sed knows the same regular expressions as `egrep(1)`:

```
sed -e '/[0-9]/d' ...            delete all lines that have numbers
sed -d '/Joe.*pobox.*23[56]/'... Delete all Poboxes of Joe with
                                 range 235 and 236
```


NOTE: *The GNU sed supports "+" qualifier, which must be preceded by a backslash.*

```
sed -e '/[0-9][0-9]\+/d' FILE    Delete lines (at least 2 nbrs)
                      |
                     Notice the extra backslash for quantifier +
```

Sed also has a *memory*, where it can store the matched text. The memory is taken into use by surrounding the regular expression between special grouping markers:

```
\(REGULAR-EXPRESSION\)      Anything between \( .. \) is remembered
  |
  The contents of this is available from memory \1
```

Like if you would change all integer numbers to decimal values:


```
$ cat data.txt
  The price is 200
  The price is 199

$ sed -e 's/\([0-9]\+\)/\1.00/g' data.txt
              |         |
              |         take the match from memory 1 and add ".00"
              remember all numbers, that are repeated, and put to
              memory 1

-->

"The price is 200.00"
"The price is 199.00"
```

The memory is not limited to 1, but you can use up to 9 memory positions. The memory positions runs from left to right in order 1,2,3 .. 9. E.g. here we use two to swap the words:

```
Joe Michaels
Ken Jordan

$ sed -e 's/\([a-zA-Z]*\) \([a-zA-Z]*\)/\2, \1/'
             |            |            |   |
             |            |            |   extract memory 1
             |            |            extract memory 2
             |            Put "Michaels" to memory 2
             put "Joe" to memory 1

-->
Michaels, Joe
Jordan, Ken
```

11.3 SED examples

To print lines:


```
$ echo -e "1\n2222\n3\n4\n5\n" | sed -n -e "1,2p"
1
2222

Enclosed in single quotes due to dollar($) shell variable sign
                                                 |
$ echo -e "1\n2222\n3\n4\n5\n" | sed -n -e '3,$p'
3
4
5
```

To delete lines:

```
$ echo -e "1\n2222\n3\n4\n5\n" | sed -e "1,2d"
3
4
5

$ echo -e "1\n2222\n3\n4\n2\n5\n" | sed -e "/2/d"
1
3
4
5
```

To change lines Pay attention to use of *double* quotes(") in this statement.

```
$ from="2" ; to="changed "
$ echo -e "1\n2222\n3\n4\n5\n" | sed -e "s/$from/$to/g"
1
changed changed changed changed
3
4
5
```

To change lines. Pay attention to use of *single* quotes(') in this statement. Multiple commands can be chained using ";" or by repeating option `-e`.

```
$ echo -e "1\n2222\n3\n4\n5\n" |
  sed -e 's/[23]$/new/g" \
      -e 's/2/word /g'
      |
      multiple "-e" options for the sed(1) command

$ echo -e "1\n2222\n3\n4\n5\n" |
  sed -e 's/[23]$/new/g ; s/2/word /g'
                        |
                        semicolon separates commands

SAME RESULT FROM BOTH COMMANDS =>
1
word word word new
new
4
5
```


12.0 Find command

12.1 Basics of find command

```
find dir [dir ...] [option ...]
```

The find command is similar to Windows graphical search, except that it is much more powerful. In other systems there may be commands like `locate(1)` and `glimpse(1)` to help searching files faster by using pre-gnerated index database files. However the find(1) command is the only one that comes standard in all systems.

Find all text files recursively starting from current directory:

```
$ find . -name "*.txt" -print
       |
```

```
Where to start finding.
"." = From current directory
```

Find files that haven't been accessed/modified for more than 10 days:


```
$ find $HOME -atime +10 -print
$ find $HOME -mtime +10 -print
```

By default the find command scans the whole file system, regardless of the files, devices or directories. You can limit the search to certain types. See more types in find(1) manual page.

Find all *directories* that contain word "tmp"

```
$ find . -type d -name "*temp*" -print
```

Find all *files* that start with big letters

```
$ find . -type f -name "[A-Z]*" -print
```

Note: More options are available in the GNU version of `find(1)`, like extended print formatting option `-printf`, long listing `-ls` and absolute path matching with `-path`. See also sort(1) command for more information about ordering items.

```
$ find $HOME -size +10k -printf "%p %s\n"
                        |
                        Check the manual pages if this option
                        is available.

$ find $HOME -size +100k -ls | sort -k 7
                    |     |          |
                    |     |          sort by column 7 (size)
                    |     print long listing (ls -l)
                    find bigger than 100k files
```

```
$ find . -type f -path "*/tmp/*.txt" -print
                        |
                        Matches /home/jdoe/tmp/this.txt
                        but also /home/jdoe/Mail/tmp/dir/more.txt
```


Exercise: Use GNU find(1) and print all files below a directory so that the ISO 8601 date and time is printed at the beginning for easy sorting by date. Stydy option `-printf` and its description.

Note: [Cygwin] The syntax of the directories is a little different, because Windows uses drive letters and Linux does not. For example, to search all of `C:/temp`, the Cygwin syntax would be:

```
$ find /cygdrive/c/temp -name "*tmp*" -print
       |
       Notice virtual dir prefix: /cygdrive
```

12.2 Find command with AND/OR options

Find also has relational operators AND OR NOT and GROUPING Find all `txt` or `doc` files. **NOTE**: put a backslash in front of the shell metacharacters like \(\) and semicolon \;

```
$ find . -type f \( -name "*.txt" -o -name "*.doc*" \) -print
```

Find all `*.doc` files but NOT those that contain numbers

```
$ find . -type f -name "*.doc" -a ! -name "*[0-9]*" -print
```

You can drop search items with `-prune`. The following command descends into directories whose name is `temp` or `tmp`.

```
$ find . -type d \( -name "temp" -o -name "tmp" \) -prune \
         -a ! -name "tmp"       \
         -a ! -name "temp"      \
         -o                     \
```

```
-type f                \
-name "*.conf"
```

12.3 Using the "launch" option (-exec)

It is possible to run a specific command on file that has been found. The option `-exec` uses a peculiar syntax where the items are below. The `COMMAND` can be any command or combination of commands:

```
$ find [options] -exec COMMAND {}  \;
                  |     ======= |   |
                  |             |   END marker
                  |             |
                  |             "found" file
                  |
                  BEGINNING marker
```

Display long listing for every file:

```
$ find . -type f -name "*.txt" -exec ls -l {} \;
                                       =====
```

Usually the `-exec` should be avoided, because it launches a process for **each** found item. Here is equivalent command which is far more resource effective. See manual page of xargs(1) for more information. Note that options `--null` and `--no-run-if-empty` are found in GNU xargs(1) and not included in the POSIX version of xargs(1).

```
find . -type f -name "*.tmp" -print0 | xargs --null --no-run-if-empty ls -l
                                                                     =====
```

Exercise: Use `time(1)` command to test just how much more effective the `xargs(1)` is compared to `-exec` version.

Process each found file with `egrep` and search word "find-this".


```
$ find . -type f -name "*.txt" \
  -exec egrep -n "find-this" {} /dev/null \;
        ====================|==|==========|
                            |  |          EOF marker for -exec which
                            |  |          must be shell "escaped"
                            |  Extra parameter for egrep(1)
                            The found file appears HERE
```

Notice that a null file `/dev/null` was used with egrep(1). This is due to `egrep(1)`, which only displays *full path* if there is not at least **two** files to search. See the difference here:

```
$ egrep -n "abc" test.txt
 1:test line abc
 5:more abc lines here

                1.       2.
                |        |
$ egrep -n "abc" test.txt /dev/null
 test.txt:1:test line abc
 test.txt:5:more abc lines here
 ========
 |
 Because two files (test.txt /dev/null) were searched,
 the filename is displayed in front of every line.
```

[advanced] A more complicated example. This one replaces file contents. Find all *.txt files that contain email addess *foo@example.com* and update the address to new *foo.bar@examnple.com*. The special construct "{}" in `find(1)` command is substituted by the current file name

```
$ find . -name Root \
    -exec \
    sh -c "sed 's/foo@example.com/foo.bar@example.com/g' < {} > {}.bak \
    && mv {}.bak {}" \;
          |
          The braces denote FILENAME
```

FURTHER READING

- Greg's Wiki, Using Find

13.0 Awk Programming

Home
Coding Style
Software

» 4A00EK44 Java
» 4A00EZ65 Java OO
» 4A00EZ62 backend vc
» 4A00EX44 SQL
» 5K00DM65 C++ (kone)
» 5G00DM05 backend
» 5G00DL95 server
» 5G00EU63 API
» 5G00DM06 API en
» 5G00DM62 testing

13.1 Awk command foreword

```
awk '{ CODE }' [var="value" ..] file ...
awk -f program.awk [var="value" ..] file ...
```

awk(1) is a "Pattern Scanning and Processing Language" which was developed 1978. The AWK was named after its developers Aho, Weinberger and Kernighan. Later, 1985 AT&T improved the language and named it `nawk(1)`, new awk. Since then the AWK has continued to improve in small steps and the Windows Cygwin distribution's `awk(1)` is in fact GNU version. Awk is a mini-programming language that makes it superior to `sed(1)` or bunch of commands put together. It also uses regular expressions to manipulate and detect text. *Perl* programming language's heritage is in `awk(1)`, `sed(1)` and many other similar utilities.

> **Note:** In Linux and Cygwin environments the awk is GNU `gawk(1)`. In other Unix operating systems, it may be one of the older and limited versions `awk(1)`.

The basic options of awk are:

```
-f PROGRAM      Read the program from file PROGRAM

-Fc             Use (c)harcter as the field separator (FS
                variable in Awk). The default is
                whitespace. This affects contents of $1 .. $N
                variables. Like in command
                "date | awk -F: '{print $2}'"
```

If you're doing very simple things, you can write the program directly inside quotes. Each word is known to Awk by variable `$N` e.g. `$1` refers to first word. The so called *main()* program is enclosed in the braces `{}`. for more infmation about formatted printing, see GNU Awk manual page `gawk(1)` and `printf(3C)`.

```
                                        Words 5 and 9 from "ls -l"
                                        |
$ ls -l | awk '{ printf "%6d %s\n", $5, $9  }'
                | |                   |
                | Print (f)ormatted   |
                |                     |
                Program BLOCK start   BLOCK end
```


13.2 Awk programming language basics

Note: The full programming language syntax is explained in the manual page `awk(1)`. See also GNU Awk page `gawk(1)`.

The program's overall structure is:

- **IMPORTANT:** The syntax requires that you put leading "{" at the same line with the BEGIN, END and /REGEXP/ condition.
- The words BEGIN and END are special and must be written in capital letters.
- The main() program is the one that has block "{}". In Awk there can be several main programs. Each "{}" block is run in the order of appearance for every matching line.
- It is not necessary to add the statement terminator ";" at the end of lines but editors may not indent lines properly if the semicolon is left out.

```
#!/usr/bin/awk -f

################################################################
#
# This block is run BEFORE anything is read.
# a) You can set initial values for variables.
# b) if program doe snot read anything; this is the main()
#
################################################################

BEGIN {
    outFile = "out.tmp"
}
```


```
##############################################################
#
# Functions are created with "function" keyword
#
##############################################################

#   Define function "SomeFunction"
#
#   input parameters:  input1 input2
#   local variables:   local1 local2
#
#   NOTE: Awk does not actually have build-in concept of
#   "local variables". By defining "extra unused function
#   arguments" it is possible to cheat Awk to create
#   "local" variables that are only visible to a function.

function SomeFunction(input1, input2,    local1, local2)
{
    local1 = input1 + input2    # sum
    local2 = local1 / 2         # average
    return( local1 )
}

##############################################################
#
# The blocks that are executed for matched lines. The last
# one matches any line.
#
##############################################################

$1 ~ /REGEX/ {
    #   Block is executed ONLY if the FIELD ($1) match regexp
}

/REGEX/ {
    #   Block is executed ONLY if WHOLE LINE match regexp
}

{
    #   Block is executed for EVERY LINE
    status = SomeFunction(1, 2)
```

```
    sum   += $5
}

#################################################################
#
#   This block is run AFTER all the lines have been processed.
#   It is called just before the program exits.
#
#################################################################

END {

    #   After all lines have been read, terminated, this
    #   final block is executed.

    printf "Total: %s", sum
}

# End of file
```


Basic character escape codes

```
\f   form feed       (not used usually)
\n   newline         (In Linux the end of line is \n   )
\r   carriage return (In Windows the end of line is \r\n )
\t   tab character
\xHH Use a HH hex code to represent a character in ASCII table
```

Special variables

```
$0              The whole line that has been read
$1, $2 .. $(N)  Individual fields. Whole line is in $0
NF              Number of fields in the line
$(NF)           The last word (= field) in the line
$(NF - 1)       Second last word in the line
```

Variables, arrays and hashes (associative arrays)

```
var = 1;            # Set VAR to 1
var = "string";     # Set var to "string"
var = x + y;        # Add two variables together

array[1]  = 1;      # Set array's first element to "1"
array[10] = "str";  # Set array's 10th element to "str";
```

Using array whose subscript is a string. In Awk all the arrays are in fact hashes, even the numerically subscripted ones.

```
hash["mike"] = "1st street, New York";
hash["joe"]  = "34st street, New Jersey";
```

The HASH is like an array, but indexed with a string, instead of NUMBER. The variable KEY contains the hash index.

```
key         = "Mike";
hash[key]   = "value";
```

Common built-in variables (see more in the gawk manual)

- `FILENAME` String. The current filename.
- `FS` Regexp. Field separator. Like that in `-F`
- `NF` Number of fields, I.e. how many $N are there
- `NR` Number of current record, I.e the line number
- `ARGC` Number of arguments in command line
- `ARGV` Array of command line arguments
- `ENVIRON` Hash containing all environment variables Try "print ENVIRON`LOGNAME`;"
- `IGNORECASE` Flag. Same as '-i' in egrep. Matches are not case sensitive

13.3 Awk code examples

Add line numbers to each line:


```
$ ls -l | awk '{print NR ": " $0}'
```

Print only few fields from input:

```
$ ls -l | awk '{ printf "%-20s %s\n", $9, $5 }'
```

Count size of files in a directory. Notice the use of END block after input.

```
$ ls -l | awk '{size += $5} END{print "Total: " size}'
```

Print only directories

```
$ ls -F | awk '/\/$/ {print}'
```

Print only lines that have 3 fields:

```
$ cat file | awk 'NR == 3 {print}'
```

Using several characters as field separator FS. Remember that `date(1)` command input is separated by spaces and for time it is separated with colons(:).

```
$ date | gawk 'BEGIN{ FS="[ \t:]+"} END{ print "fields: " NF } '
         |
         Notice: GNU awk is used here. The antique
         awk(1) may not understand this code snippet.
```

Count how many empty lines are in `file.txt`:

```
$ awk '/^[ \t]*$/ {i++} END{print "Total: " i} ' file.txt
                        |
                        The "END" prints accumulated "i"
```


Count how much space files take in some directory. The AWK is used for parsing the output of `ls`, whose line indicates the file size. **Note:** the `ls -l` listing may be different in various Linux/Unix/Cygwin operating systems, so the exact position of file size field varies.

```
$ chmod +x sum.awk          # Make it "executable"
$ ls -l | ./sum.awk
          |
          Run program "sum.awk" from current "." directory
```

The program `sum.awk` could be written like this. Notice the *first* line that indicated the command interpreter (Try `type -a gawk`):

```
#!/usr/local/bin/gawk -f

BEGIN {
    sum = 0
}

{
    line = $0
    pos  = match(line, "^[-]")

    if ( pos > 0 )
    {
        print pos ": " line
        sum += $5
    }
}

END {
    print "total is: " sum
}
```

```
# End of file
```

Here is an example how to print all the environment variables. The AWK's global variable `ENVIRON` is traversed using a hash `for` loop construct:


```
BEGIN {

    for (var in ENVIRON)
    {
        printf "%-20s %s\n\n", var, ENVIRON[var]
    }

    exit 0
}
```

13.3.1 Advanced example

An AWK program can even examine command like arguments like any other program. Here is a simple calculator:

```
$ chmod +x calc.awk
$ ./calc.awk 1 2 3
Argument 0 = gawk
Argument 1 = 1
Argument 2 = 2
Argument 3 = 3
Sum: 6
```

The program must run all code inside `BEGIN`, because no lines are read from `stdin`. Notice the little command `exit` when the code has finished. To understand this code, you need to be familiar with data type `hash` (in awk parlance: associative array) and how it relates to environment variable ARGV.

```
BEGIN {
    sum = 0
    for ( i=0; i < ARGC; i++ )
```

```
    {
        arg = ARGV[i]
        printf "Argument %d = %s\n", i, arg
        sum += arg
    }
    print "Sum: " sum
    exit 0
}
```


14.0 Appendix A - essential commands briefly

14.1 The manual pages

14.1.1 Finding the help information

The shell provides `man(1)` which can assist you to find the help about any command. For example to find command syntax for the `ls(1)`, you would give command:

```
$ man ls        # read section (1) for command "ls"
$ man -k list   # Search by keywords
```

The `man(1)` command itself has many options that you can use. For example, the apropos is usually behind switch `-k`. See listing and also option `-s`.

```
$ man man
```

There is also more search commands, like `whereis(1)` and `whatis(1)` which tells the command locations and the associated manual page location. The `apropos(1)` is like running man command with the keyword `-k` option. If you are only interested in finding where is the command in your system, use the `which(1)` command that tells the first location that the command occurs in a path

```
whereis ls
=> ls: /bin/ls /opt/./bin/ls /usr/man/man1/ls.1

apropos kill
```

```
which ls              # Where is the ls(1) command located?
```

Linux may include the GNU `info(1)` page viewer for command manuals. These pages are usually located in /usr/share/info or /usr/local/info.

14.1.2 Manual page sections

The manual pages can be in different section, check your `$MANPATH` environment variable where all the system manual pages are located. The most typical sections are:

```
echo $MANPATH $MANPAGER
less /etc/man*            # The manual page configuration
ls /usr/man
whatis mount
```

To request specific page, say mount (8):

```
man 8 mount               Linux accepts number directly
man -s8 mount             SunOS/Solaris requires -s option
```

The pages are typically grouped in sections:

```
1       User commands
2       Systems calls and system error messages
3       C language subroutines or library functions
4       File formats or devices
5       Miscellaneous files and typesetting macros
6       Games
7       Miscellaneus. Special files (device file naming)
8       System maintenance
```

14.1.3 Configuring the manual page viewer

The environment variable `PAGER` controls which program is used to paginate the input. The default pager is usually `more(1)`, but you can change it to more feature rich `less(1)` with:

```
$ PAGER="less"; export PAGER          # old Bourne Shell syntax
$ export PAGER="less"                 # POSIX syntax
```


14.1.4 Converting manual pages to text

The manual pages are not usually in human readable format, but written using a `nroff(1)` text formatting syntax. you can't just save the pages that you see with output redirection, because they contain terminal controlling codes, like, bold `^H` that looks nice in the terminal, but not when outputted to printed.

```
$ whereis ls
ls: /bin/ls /opt/./bin/ls /usr/man/man1/ls.1

$ head /usr/man/man1/ls.1

. Do NOT MODIFY THIS FILE! It was generated by help2man 1.5.1.2
. TH LS 1 "November 1998" "GNU Fileutils 4.0" "FSF"
. SH NAME
ls \- list directory contents
.SH SYNOPSIS
.B ls
[\fIOPTIONS\fR]...[\fIFILE\fR]...
.SH DESCRITION
.PP
```

14.1.5 Formatting the manual page to plain text

As you can see, the file content is not pure text. The man command interprets this to the terminal printable format, but look at the the contents and you will see extra control characters still in the file:

```
# don't do "man ls > ls.txt"

man ls | cat -v
man ls | col -bx | cat -v
```

```
=======
Filter out "bold" and "strike"
```

The file contains lot of gibberish characters that must be removed before a pure text page can be printed. There is command `col(1)` that includes handy character cleanup options. This command line produces a clean text-only manual page to the file *ls.man*, which can be printed


```
man ls | col -bx > ls.txt 2> /dev/null
```

14.2 Dealing with the line endings

If you move a Linux file to Windows, the line endings are different and you won't see the file correctly e.g in *notepad* (You could use *wordpad*). similarly, if you move a Windows file to Linux, you won't see a clean file, but an extra `^M` character at the end of each line. This is due to differences in **end-of-line** markers in the operating system. A handy tool `tr(1)`, short name for "translate", can easily convert a Windows file to Linux by deleting the extra **\r** character. A more universal is the `perl(1)` command that can convert any regular expression in very simple way to another.

```
    example line\r\n        Windows
    example line\n          Linux/Unix/Mac OS

$ tr -d "\r" < Windows.txt > unix.txt                   # Windows => Linux
$ perl -pi.bak -e "s/\n/\r\n/"  unix.txt > Windows.txt  # Unix => Windows
```

14.3 Using the shell and terminal

The choice of the shells depends on the operating system flavor, but you can always peek directory /bin to find out what *sh* commands exist there. There is also command `which(1)` that you can use to locate any given command in the file system:

```
sh        # change to bourne shell
bash      # change to Bourne Again Shell
tcsh      # change to extended C-shell
csh       # change to C-shell
ksh       # change to Korn-shell
zsh       # change to Z-shell
```

The default shell has been set by your system administrator in /etc/passwd (unless this file is in yellow pages). The command to /change the default shell permanently is `chsh(1)`. The most used interactive shell is the `tcsh(1)` and `bash(1)` and possibly `zsh(1)`. **NOTE:** Not all shells come standard with the operating system. To change the shell permanently use command:

```
chsh -l

    /bin/bash
    /bin/sh
    /bin/ash
    /bin/tcsh
    /bin/csh

chsh -s /bin/bash <your login name>
... log out and log in ...

echo $SHELL
    /bin/bash
```


15.0 Appendix B - Commands briefly

15.1 File commands

```
cat     print content of file(s)
cd      change directory
chmod   Change modes  (files and directories)
comm    Report common and uncommon lines
cp      copy files
diff    Report difference between files.
ls      List files
mv      move files (you can think also as rename)
od      Dump the content of file ("od -h -c"  for hex dump)
pwd     print working firectory
rm      Remove files
split   Split file
tee     Genrate output to files inside pipeline
touch   Modify File's time stamp
```

```
tar     Group files into TAR packet. Usually with gzip or bzip2
bzip2   Compress single file with bzip2
gzip    Compress single file with gzip
compress Old .Z compression
```


15.2 Disk commands

```
du      Disk usage
df      Disk free, sometimes as bdf(1)
find    find files under all directories below
```

15.3 Process commands

```
bg      Put process to background
fg      put process to foreground
jobs    Print started jobs from current tty
ps      Print running processes
```

15.4 Manipulation commands

```
awk         A mini-programming language
cut         Cut portion from input
egrep       Grep lines. See also grep(1) fgrep(1)
head        print only from the beginning of file
join        Join lines of two files
paste       Paste input to lines
sed         Stream editor: Manipulate lines
sort        Sort input by field
tail        Print only from the end of file
tr          Translate words
uniq        Print only unique lines
```

15.5 Text formatting commands

```
adjust      for filling, centering, ..justifying
a2ps        ASCII to postscript converter
banner      big letters
col -bx     remove control codes from man pages
fold        fold long lines for finite width output device
fmt         simple formatter
groff       text manipulating command (for manual pages)
nl          number lines
pr          paginate text: Add header and footer
```

15.6 Printing commands

```
cancel      Terminate the print queue
enscript    SunOS Multi-page printing
lpr         Line printer (the most basic printing command)
lpstat      Show printer status (queue)
pps         pretty printer for PostScript, see psutils(1)
```

15.7 Miscellaneous commands

```
echo        Write to the stdout
date        Print date. See srftime(3C)
time        Time the execution of command
wc          Word count ("wc -l" for line count)
sleep       Sleep seconds (wait command)
bc, dc      Desk calculator. See http://www.bgw.org/tutorials/utilities/
```

15.8 Packaging commands


```
gzip        .gz    Per-file basis compression
bzip2       .bz2   Per-file basis compression
compress    .Z     Per-file basis compression
uncompress  .Z     Per-file basis decompression
pack        .z     Per-file basis compression
unpack      .z     Per-file basis decompression
zip         .zip   Multiple file compression
unzip       .zip   Multiple file decompression

tar         .tar   Tar archiver
zip         .zip   Multiple files compression and archiving
cpio       Copy input-output (Older, not so much used)
```

15.9 Windows to Linux translation

```
                            Windows DOS             Linux
-------------------------------------------------------------
display list of files       dir /w                  ls
                            dir                     ls -l
display contents of file    type                    cat
display file with pauses    type filename | more    less file

copy file                   copy                    cp
find string in file         find                    egrep
compare files               comp                    diff
rename file                 rename OR ren           mv
delete file                 erase OR del            rm
delete directory            rmdir OR rd             rmdir
delete everything           deltree                 rm -rf *
change file protection      attrib                  chmod
create directory            mkdir OR md             mkdir
change working directory    chdir OR cd             cd
get help                    help                    man
                                                    apropos
                                                    which
display date and time       date, time              date
display free disk space     chkdsk                  df
```

```
print file                print                lpr
display print queue       print                lpq
write to screen           echo                 echo
```