8.1 Introduction

A large number of the files in a typical filesystem are *text files*. Text files contain simply text, no formatting features that you might see in a word processing file.

Because there are so many of these files on a typical Linux system, a great number of commands exist to help users manipulate text files. There are commands to both view and modify these files in various ways.

In addition, there are features available for the shell to control the output of commands, so instead of having the output placed in the terminal window, the output can be *redirected* into another file or another command. These redirection features provide users with a much more flexible and powerful environment to work within.

8.2 Linux Essentials Exam Objectives

This chapter will cover the topics for the following Linux Essentials exam objectives:

Topic 3: The Power of the Command Line (weight: 10)

- **3.2 Searching and Extracting Data from Files**
    - Weight: 4
    - Description: Search and extract from files in the home directory.
    - Key Knowledge Areas:
        - Command line pipes
        - I/O re-direction
        - Partial POSIX Regular Expressions (.,[],*,?)
    - The following is a partial list of the used files, terms, and utilities:
        - find
        - grep
        - less
        - head, tail
        - sort
        - cut
        - wc
    - Thing that are nice to know:
        - Partial POSIX Basic Regular Expressions ([^], ^, $)
        - Partial POSIX Extended Regular Expressions (+,(),|)
        - xargs

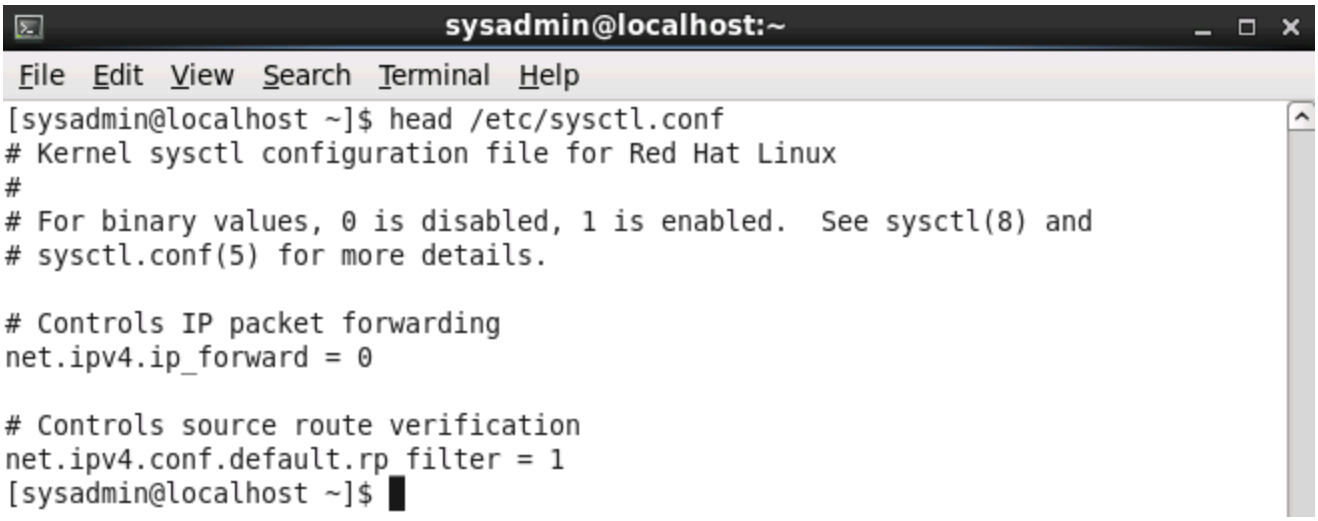
8.3 Command Line Pipes

Previous chapters discussed how to use individual commands to perform actions on the operating system, including how to create/move/delete files and move around the system. Typically, when a command has output or generates an error, the output is displayed to the screen; however, this does not have to be the case.

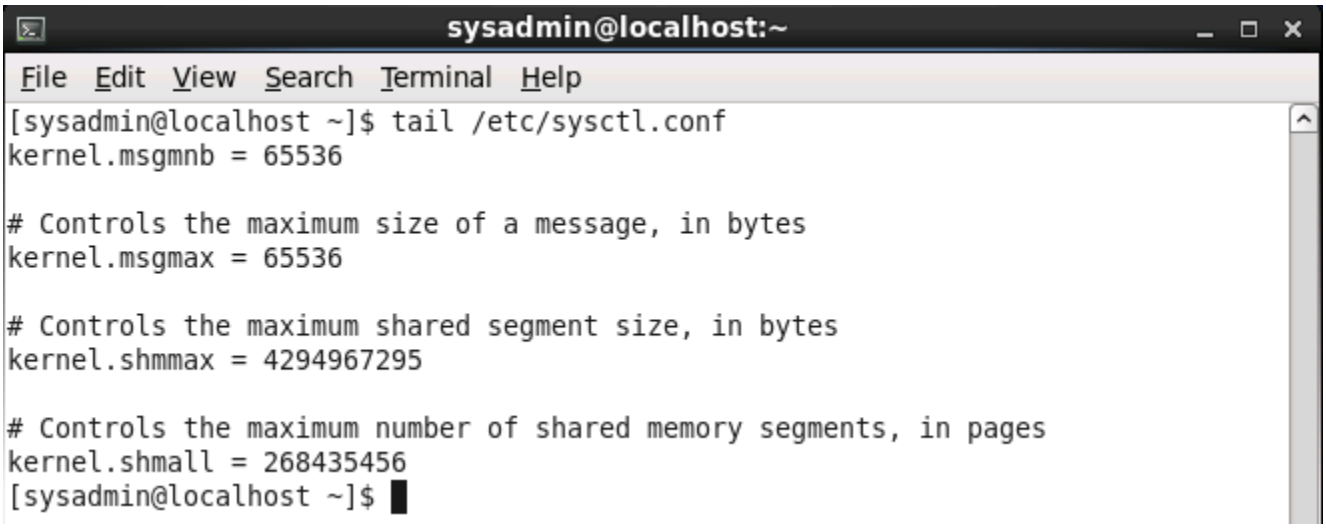
The *pipe* (|) character can be used to send the output of one command to another. Instead of being printed to the screen, the output of one command becomes input for the next command. This can be a powerful tool, especially when looking for specific data; *piping* is often used to refine the results of an initial command.

The `head` and `tail` commands will be used in many examples below to illustrate the use of pipes. These commands can be used to display only the first few or last few lines of a file (or, when used with a pipe, the output of a previous command).

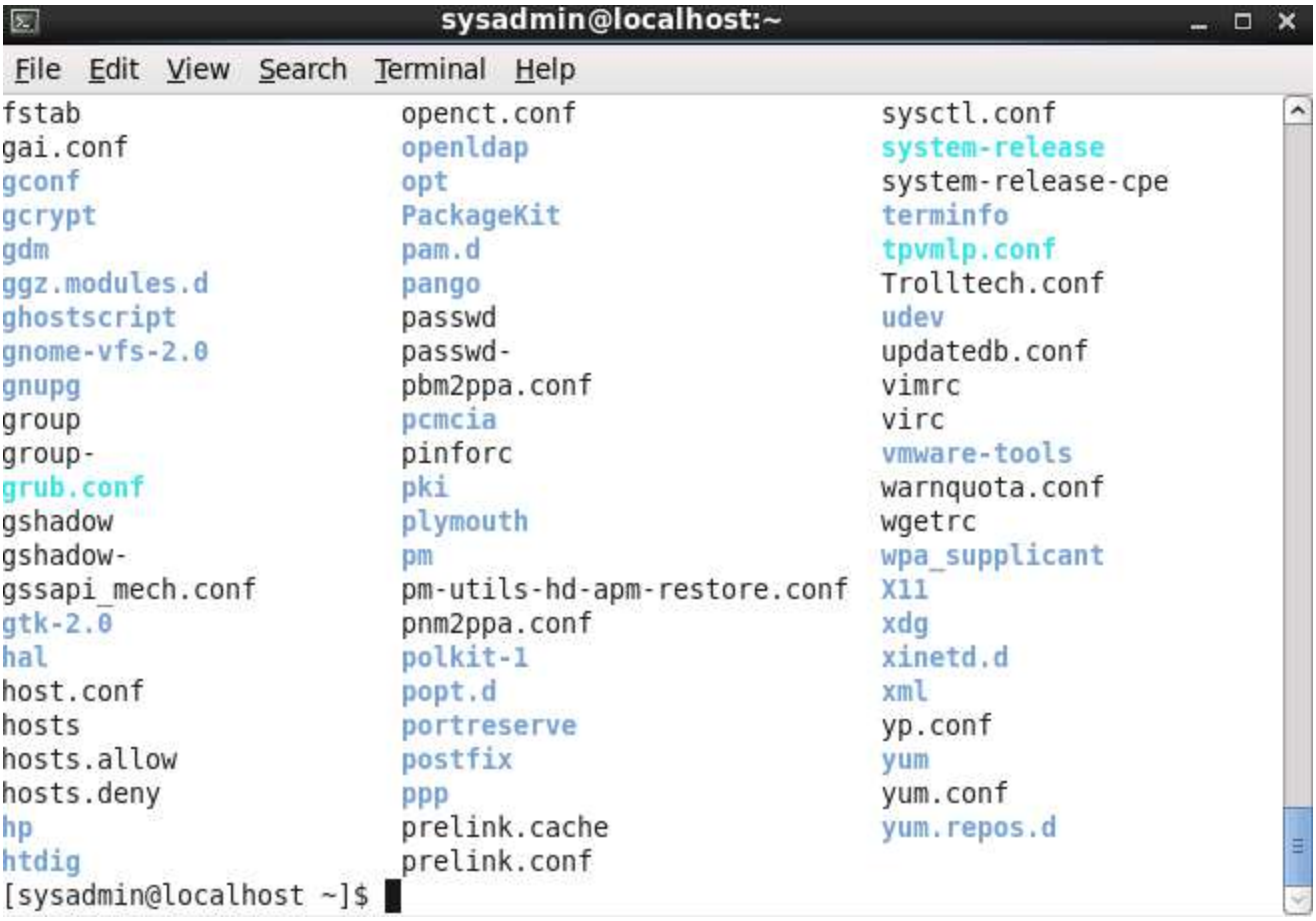
By default the `head` and `tail` commands will display ten lines. For example, the following command will display the first ten lines of the `/etc/sysctl.conf` file:



In the next example, the last ten lines of the file will be displayed:

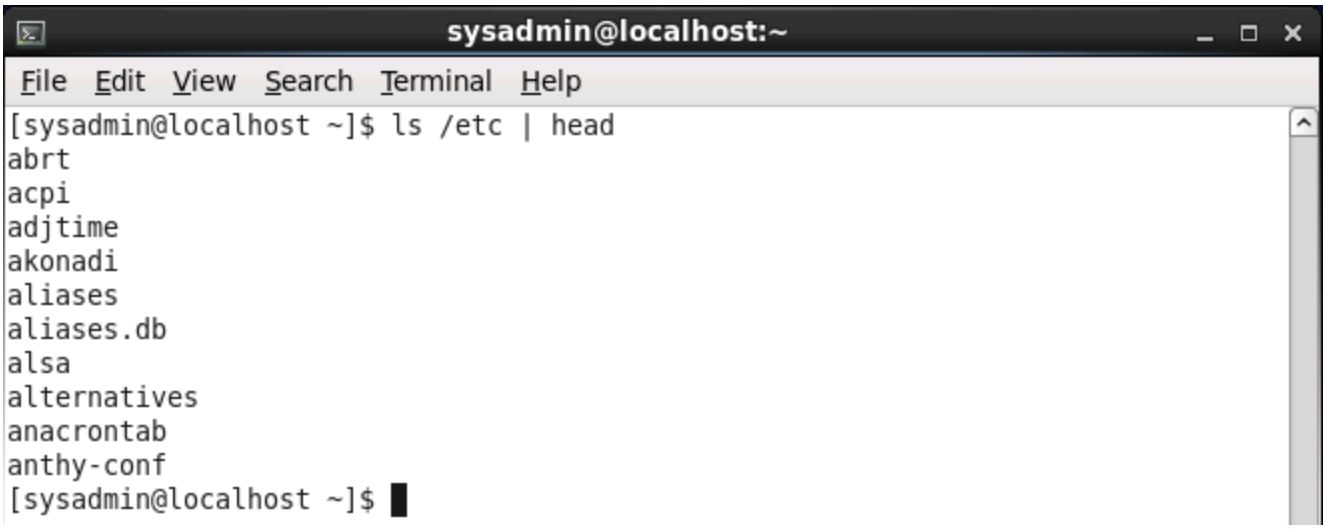


The pipe character will allow users to utilize these commands not only on files, but on the output of other commands. This can be useful when listing a large directory, for example the `/etc` directory:



If you look at the output of the previous command, you will note that first filename is `fstab`. But there are other files listed "above" that can only be viewed if the user uses the scroll bar. What if you just wanted to list the first few files of the `/etc` directory?

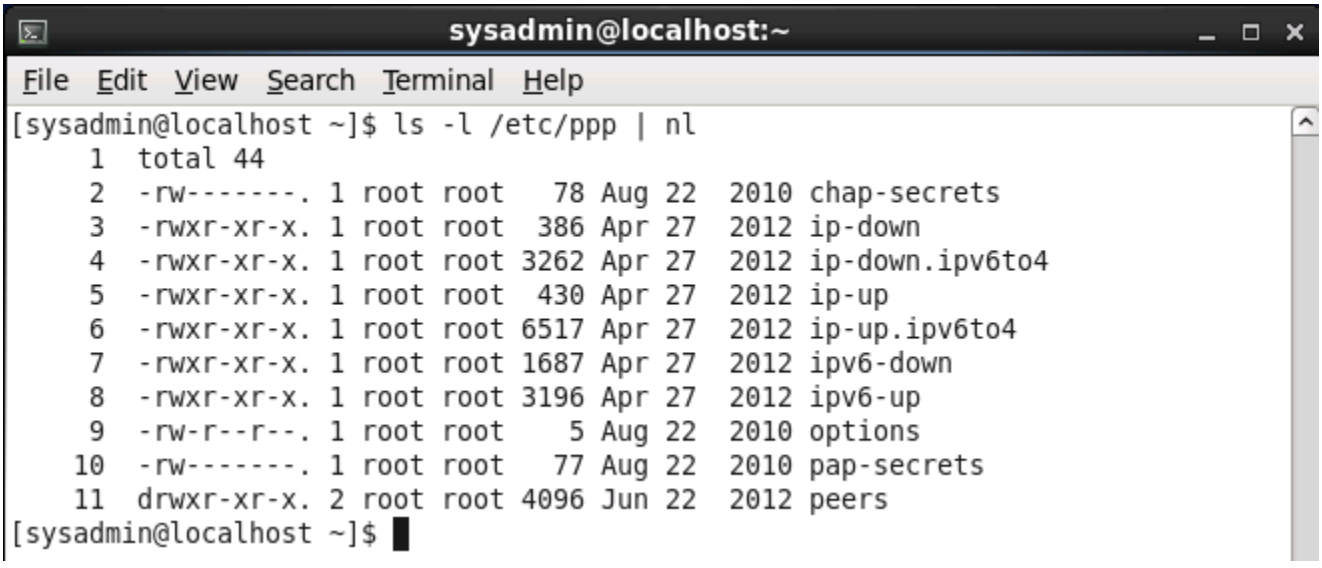
Instead of displaying the full output of the above command, piping it to the `head` command will display only the first ten lines:



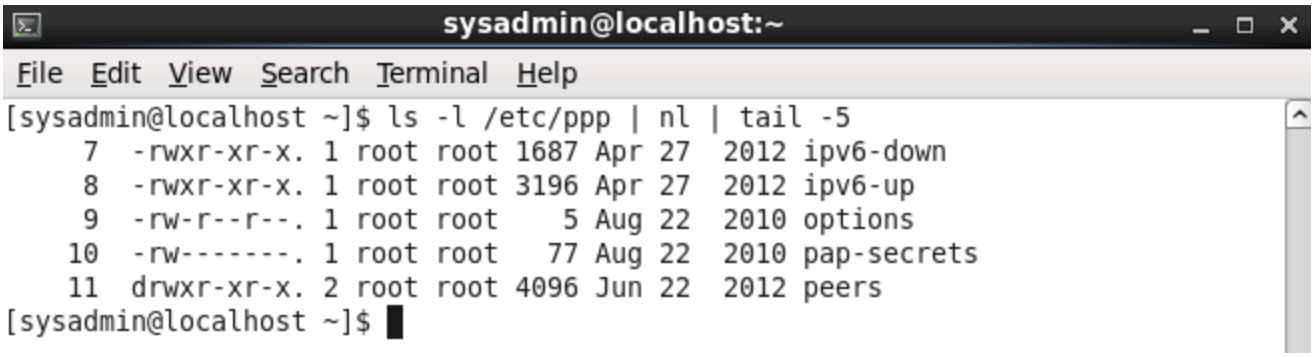
The full output of the `ls` command is passed to the `head` command by the shell instead of being printed to the screen. The `head` command takes this output (from ls) as "input data" and the output of `head` is then printed to the screen.

Multiple pipes can be used consecutively to link multiple commands together. If three commands are piped together, the first command's output is passed to the second command. The output of the second command is then passed to the third command. The output of the third command would then be printed to the screen.

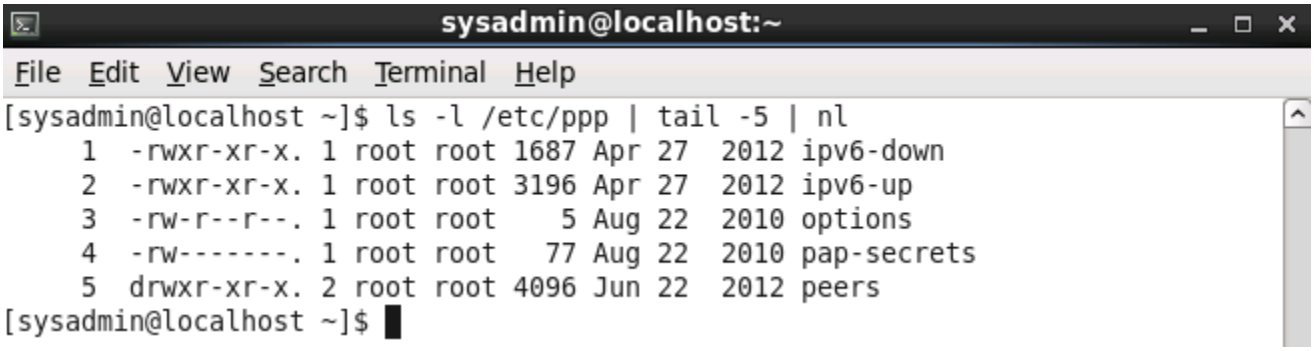
It is important to carefully choose the order in which commands are piped, as the third command will only see input from the output of the second. The examples below illustrate this using the `nl` command. In the first example, the `nl` command is used to number the lines of the output of a previous command:



In the next example, note that the `ls` command is executed first and its output is sent to the `nl` command, numbering all of the lines from the output of the `ls` command. Then the `tail` command is executed, displaying the last five lines from the output of the `nl` command:



Compare the output above with the next example:



Notice how the line numbers are different. Why is this?

In the second example, the output of the `ls` command is first sent to the `tail` command which "grabs" only the last five lines of the output. Then the `tail` command sends those five lines to the `nl` command, which numbers them 1-5.

Pipes can be powerful, but it is important to consider how commands are piped to ensure that the desired output is displayed.

8.4 I/O Redirection

Input/Output (I/O) redirection allows for command line information to be passed to different *streams*. Before discussing redirection, it is important to understand standard streams.

8.4.1 STDIN

Standard input, or STDIN, is information entered normally by the user via the keyboard. When a command prompts the shell for data, the shell provides the user with the ability to type commands that, in turn, are sent to the command as STDIN.

8.4.2 STDOUT

Standard output, or STDOUT, is the normal output of commands. When a command functions correctly (without errors) the output it produces is called STDOUT. By default, STDOUT is displayed in the terminal window (screen) where the command is executing.

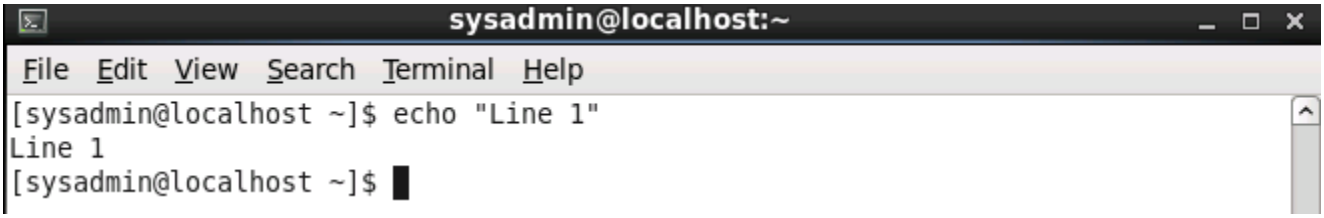
8.4.3 STDERR

Standard error, or STDERR, are error messages generated by commands. By default, STDERR is displayed in the terminal window (screen) where the command is executing.

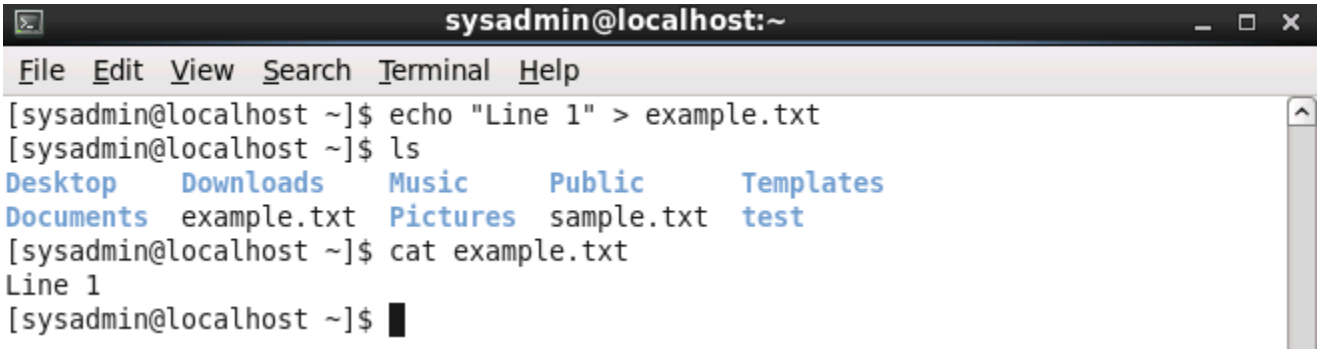
I/O redirection allows the user to redirect STDIN so data comes from a file and STDOUT/STDERR so output goes to a file. Redirection is achieved by using the arrow characters: (<) and (>).

8.4.4 Redirecting STDOUT

STDOUT can be directed to files. To begin, observe the output of the following command which will display to the screen:

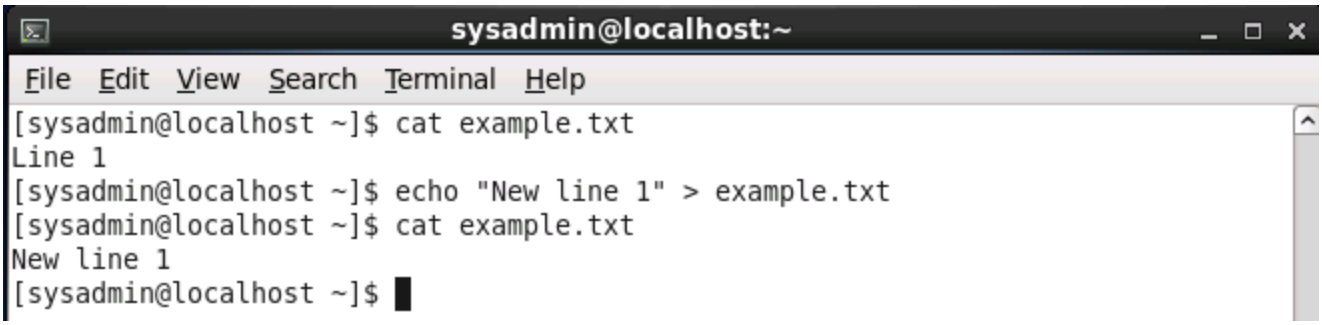


Using the > character the output can be redirected to a file:



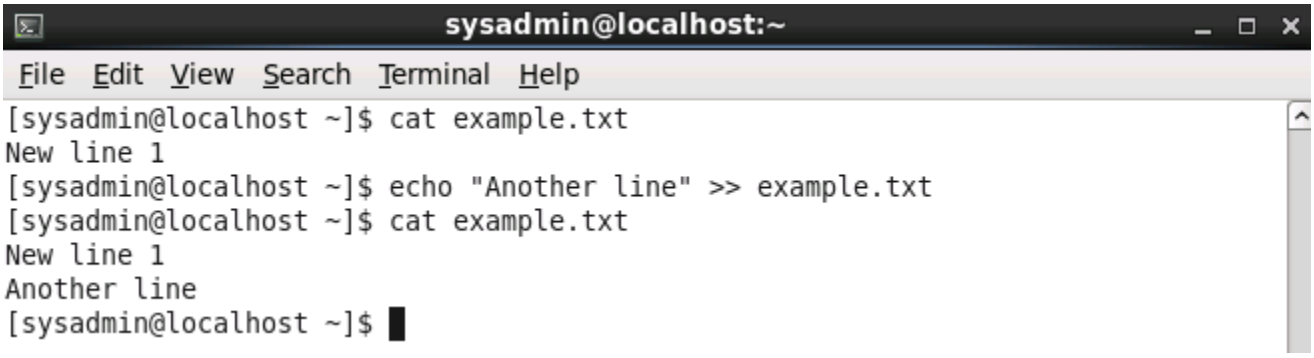
This command displays no output, because STDOUT was sent to the file `example.txt` instead of the screen. You can see the new file with the output of the `ls` command. The newly-created file contains the output of the `echo` command when the file is viewed with the `cat` command.

It is important to realize that the single arrow will overwrite any contents of an existing file:



The original contents of the file are gone, replaced with the output of the new `echo` command.

It is also possible to preserve the contents of an existing file by appending to it. Use "double arrow" (>>) to append to a file instead of overwriting it:



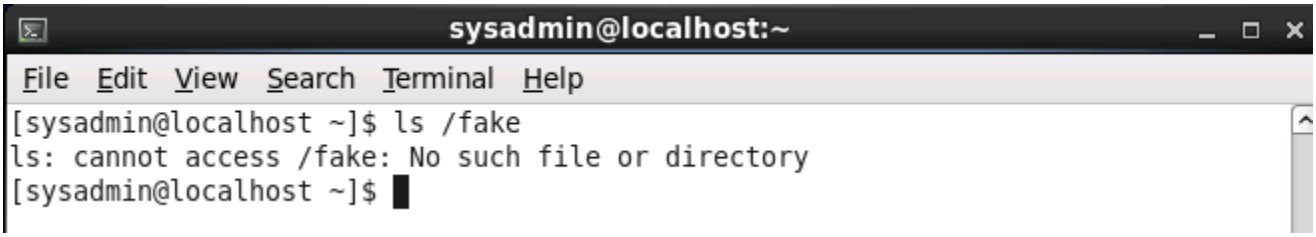
Instead of being overwritten, the output of the most recent `echo` command is added to the bottom of the file.

8.4.5 Redirecting STDERR

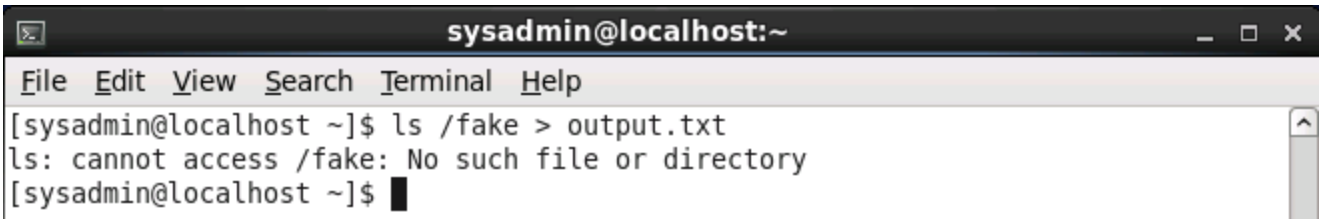
STDERR can be redirected in a similar fashion to STDOUT. STDOUT is also known as *stream* (or*channel*) #1. STDERR is assigned stream #2.

When using arrows to redirect, stream #1 is assumed unless another stream is specified. Thus, stream #2 must be specified when redirecting STDERR.

To demonstrate redirecting STDERR, first observe the following command which will produce an error because the specified directory does not exist:

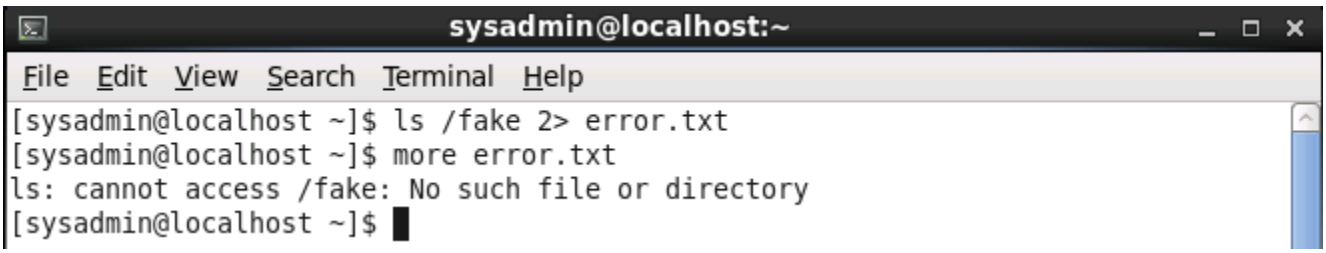


Note that there is nothing in the example above that implies that the output is STDERR. The output is clearly an error message, but how could you tell that it is being sent to STDERR? One easy way to determine this is to redirect STDOUT:



In the example above, STDOUT was redirected to the `output.txt` file. So, the output that is displayed can't be STDOUT because it would have been placed in the `output.txt` file. Because all command output goes either to STDOUT or STDERR, the output displayed above must be STDERR.

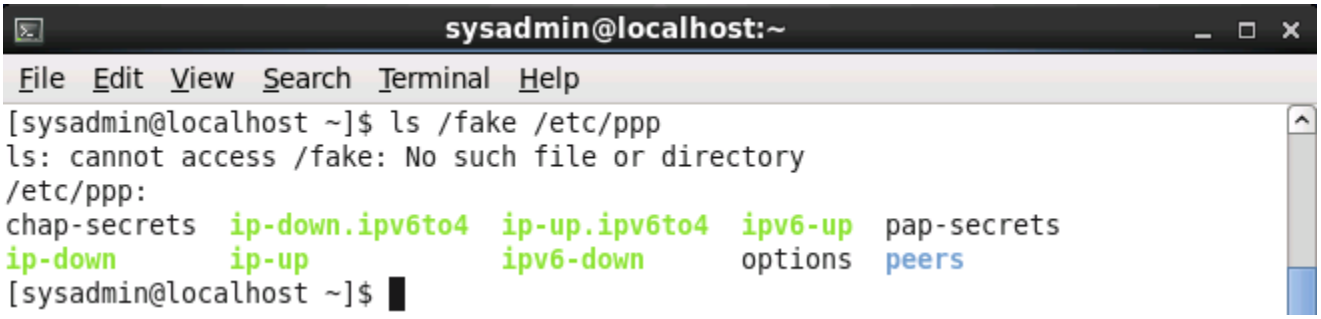
The STDERR output of a command can be sent to a file:



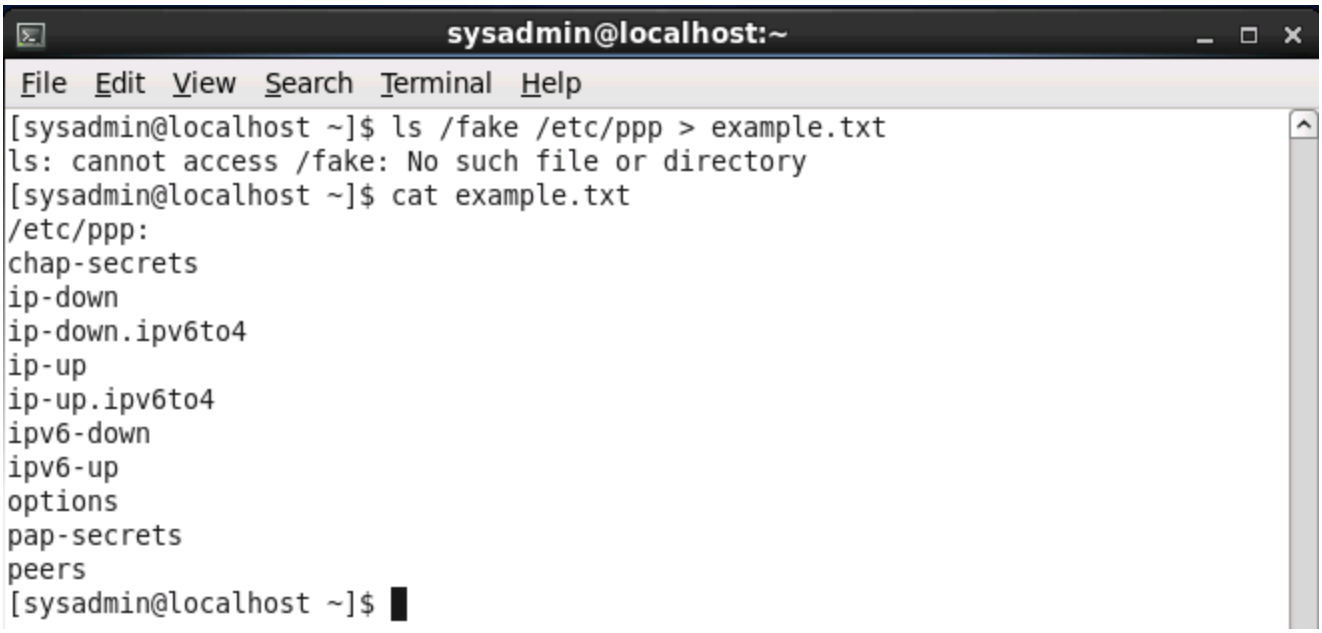
In the command above, the `2>` indicates that all error messages should be sent to the file `error.txt`.

8.4.6 Redirecting Multiple Streams

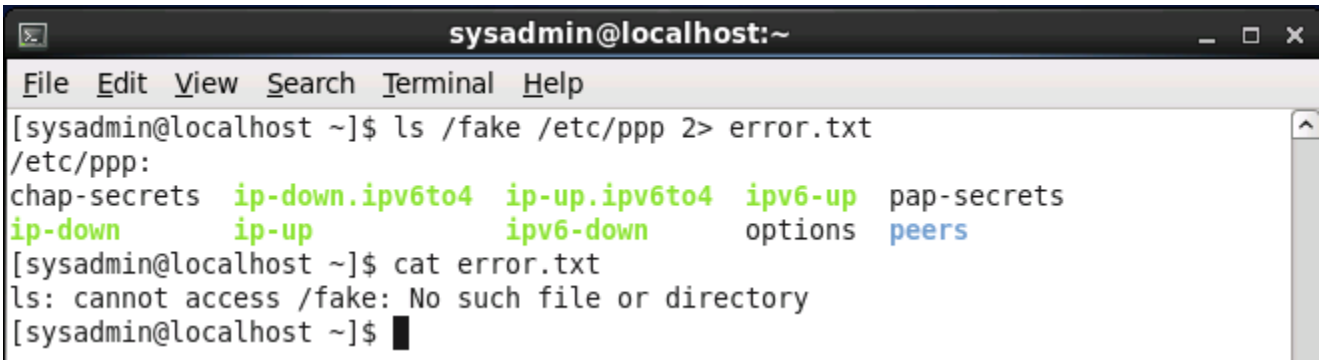
It is possible to direct both the STDOUT and STDERR of a command at the same time. The following command will produce both STDOUT and STDERR because one of the specified directories exists and the other does not:



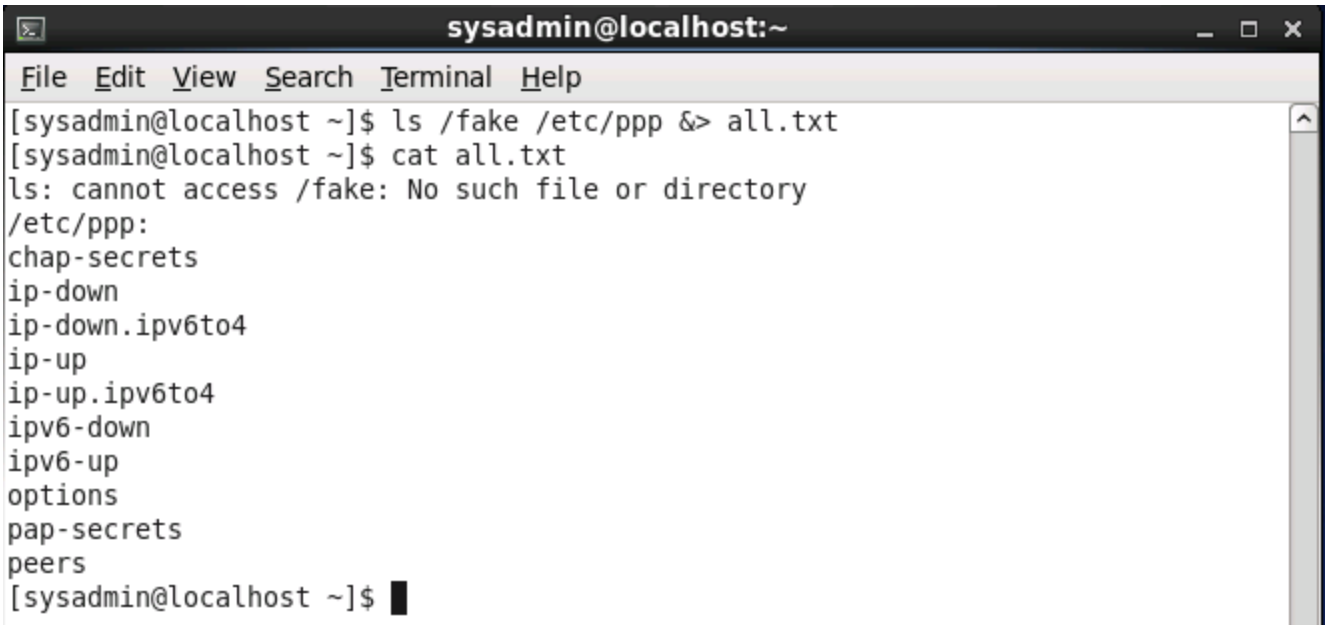
If only the STDOUT is sent to a file, STDERR will still be printed to the screen:



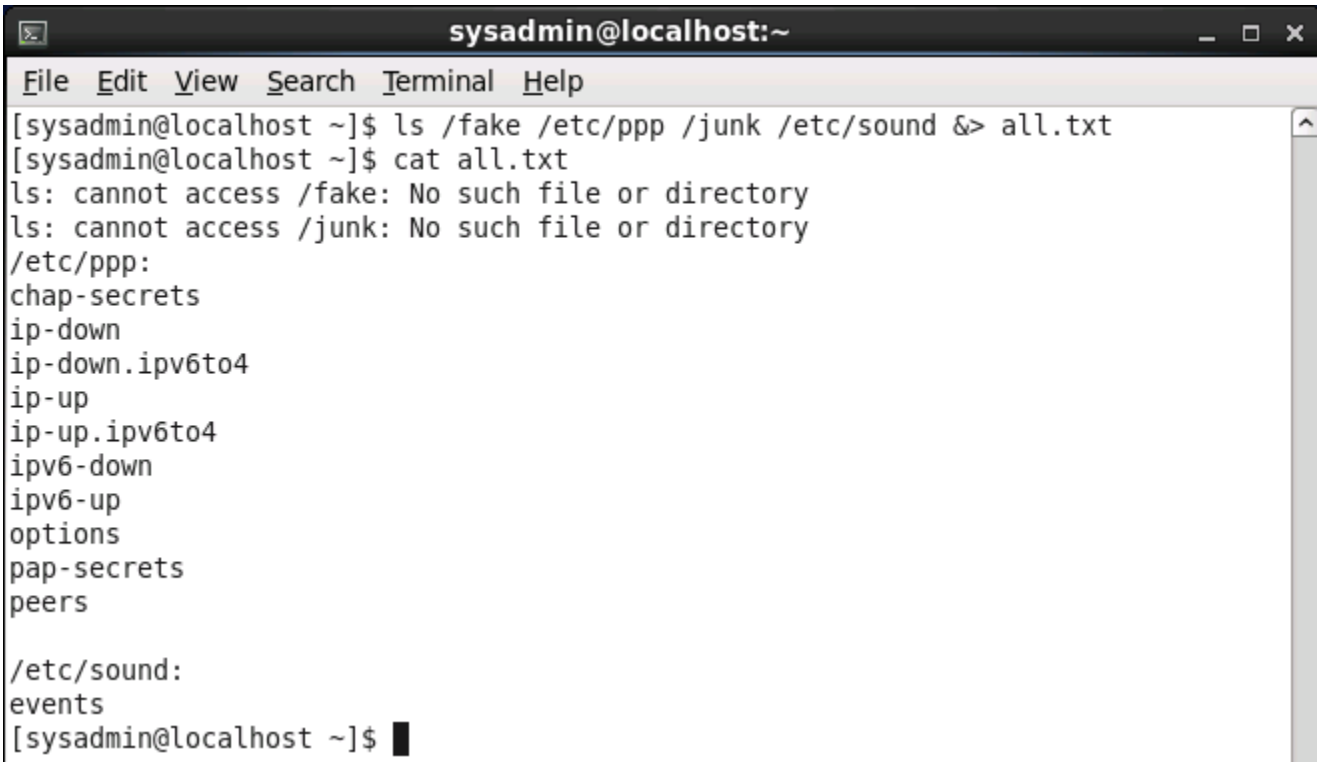
If only the STDERR is sent to a file, STDOUT will still be printed to the screen:



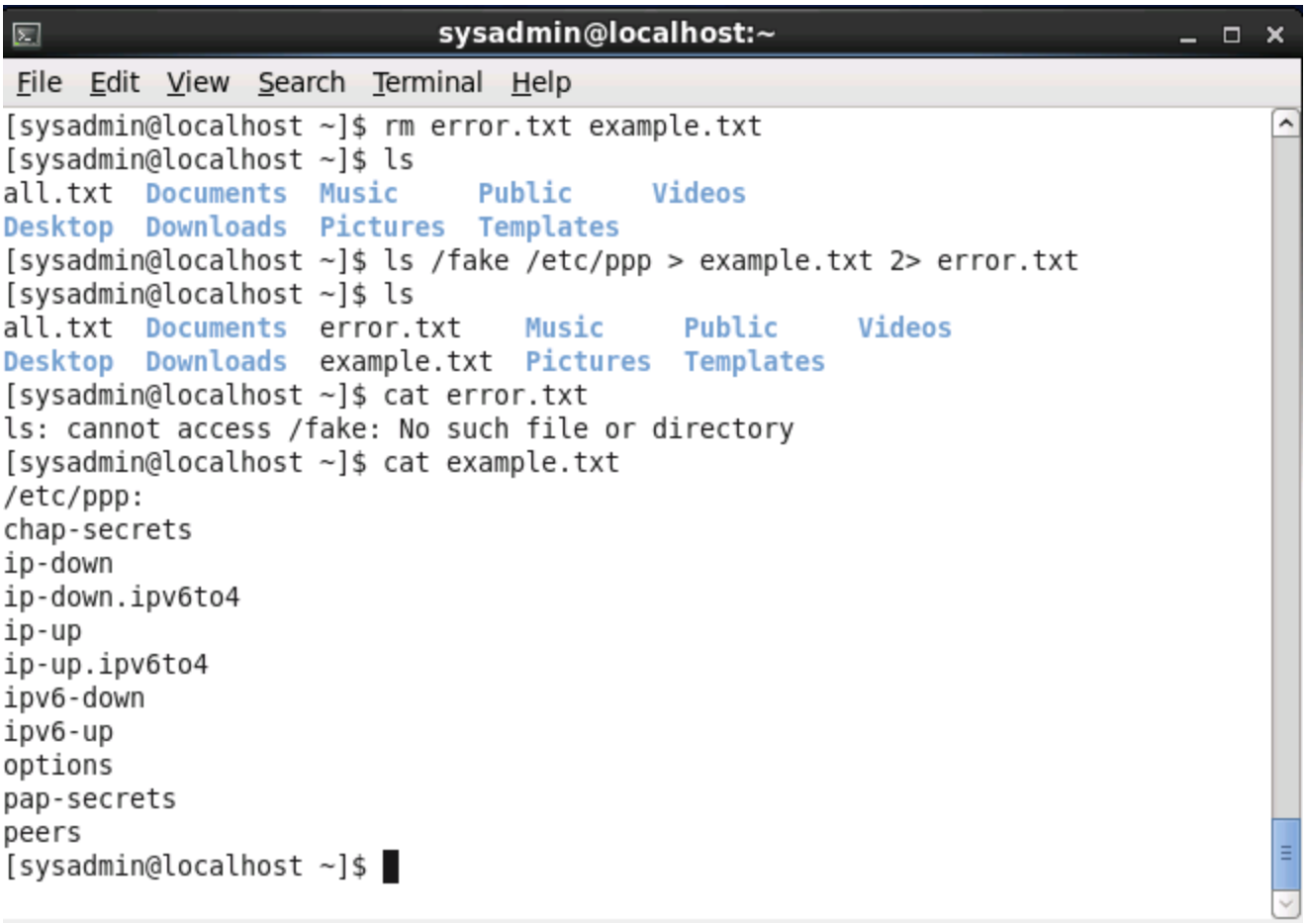
Both STDOUT and STDERR can be sent to a file by using &>, a character set that means "both 1> and 2>":



Note that when you use &>, the output appears in the file with all of the STDERR messages at the top and all of the STDOUT messages below all STDERR messages:



If you don't want STDERR and STDOUT to both go to the same file, they can be redirected to different files by using both > and 2>. For example:



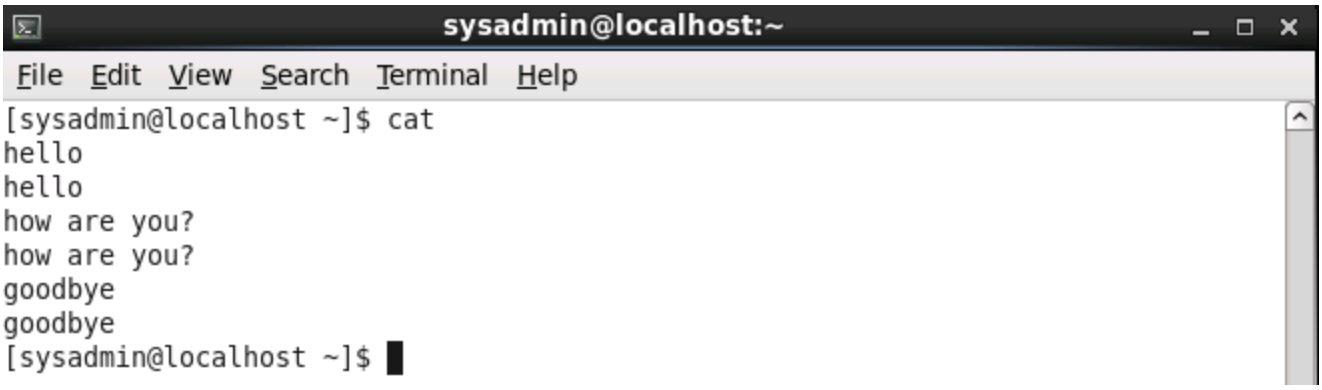
The order the streams are specified in does not matter.

8.4.7 Redirecting STDIN

The concept of redirecting STDIN is a difficult one because it is more difficult to understand *why*you would want to redirect STDIN. With STDOUT and STDERR, the answer to *why* is fairly easy: because sometimes you want to store the output into a file for future use.

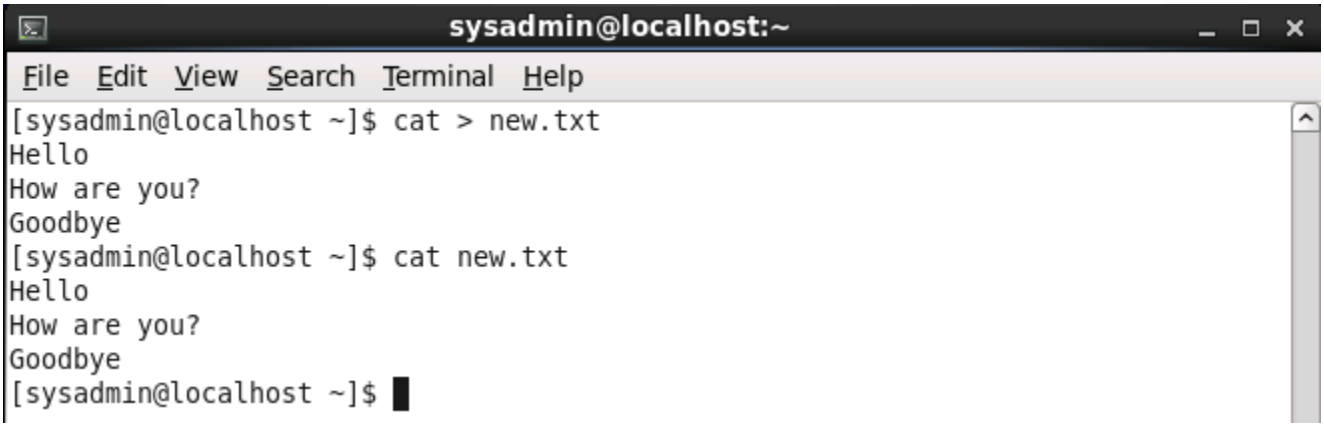
Most Linux users end up redirecting STDOUT routinely, STDERR on occasion and STDIN...well, very rarely. There are very few commands that require you to redirect STDIN because with most commands if you want to read data from a file into a command, you can just specify the filename as an argument to the command. The command will then look into the file.

For some commands, if you don't specify a filename as an argument, they will revert to using STDIN to get data. For example, consider the following `cat` command:



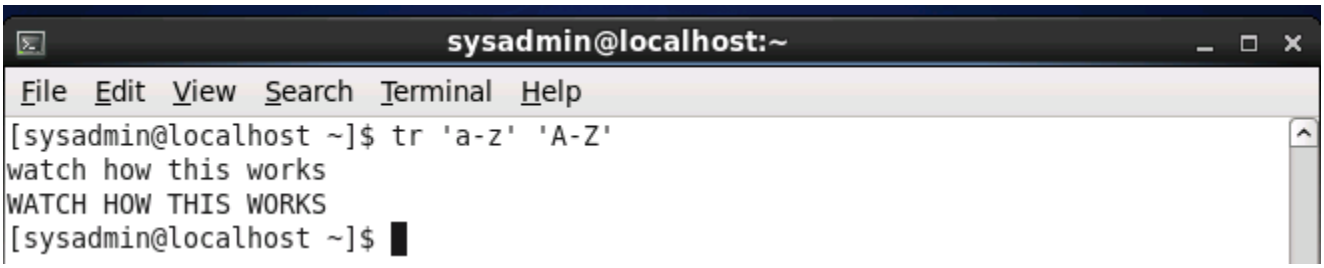
In the example above, the `cat` command wasn't provided a filename as an argument. So, it asked for the data to display on the screen from SDTIN. The user typed "hello" and then the `cat` command displayed "hello" on the screen. Perhaps this is useful for lonely people, but not really a good use of the `cat` command.

However, perhaps if the output of the `cat` command were redirected to a file, then this method could be used either to add to an existing file or to place text into a new file:



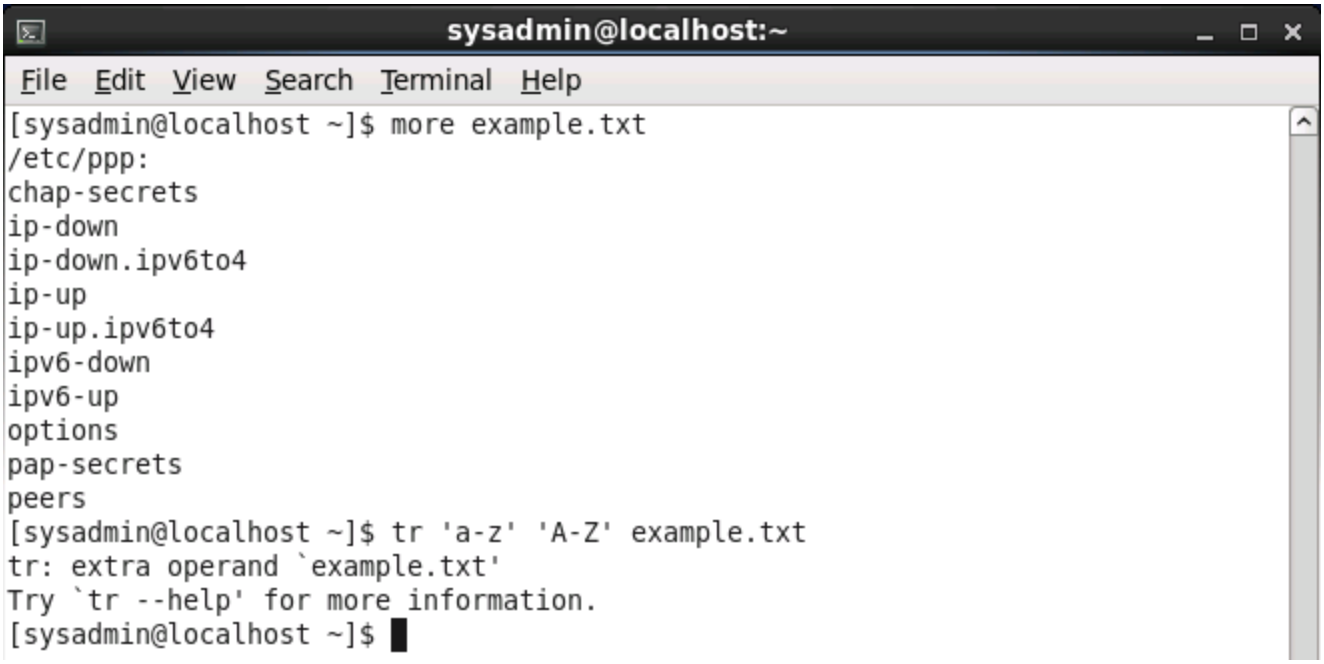
While the previous example demonstrates another advantage of redirecting STDOUT, it doesn't address why or how STDIN can be directed. To understand this, first consider a new command called `tr`. This command will take a set of characters and translate them into another set of characters.

For example, suppose you wanted to capitalize a line of text. You could use the `tr` command as follows:

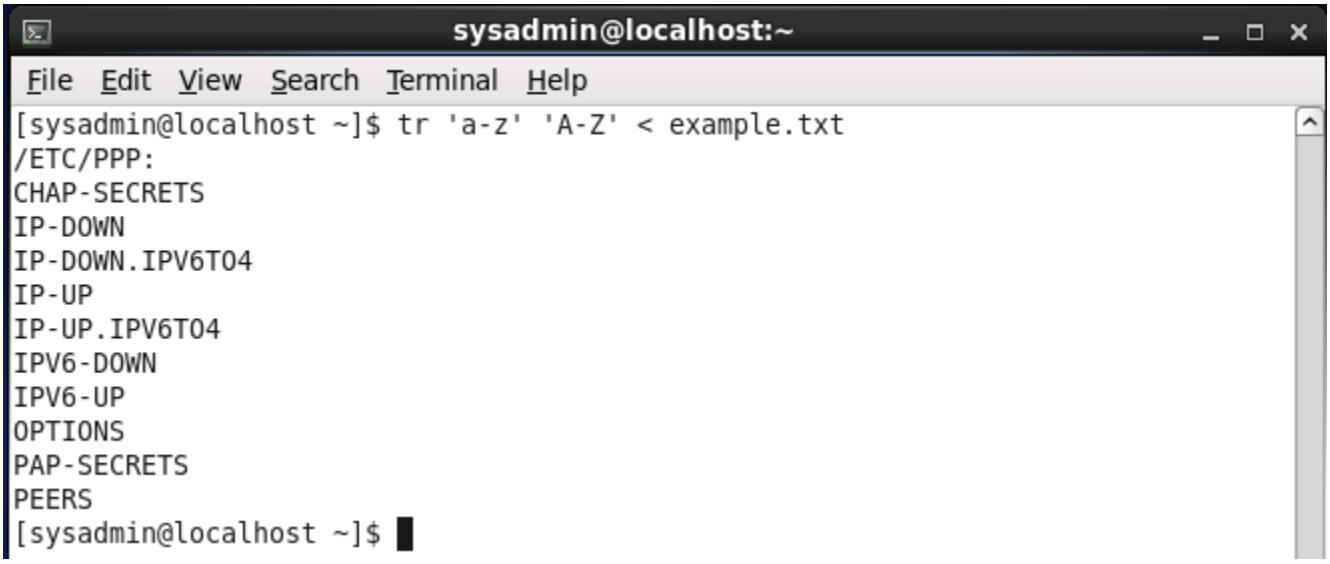


The `tr` command took the STDIN from the keyboard ("watch how this works") and converted all lower case letters before sending STDOUT to the screen ("WATCH HOW THIS WORKS").

It would seem that a better use of the `tr` command would be to perform translation on a file, not keyboard input. However, the `tr` command does not support filename arguments:

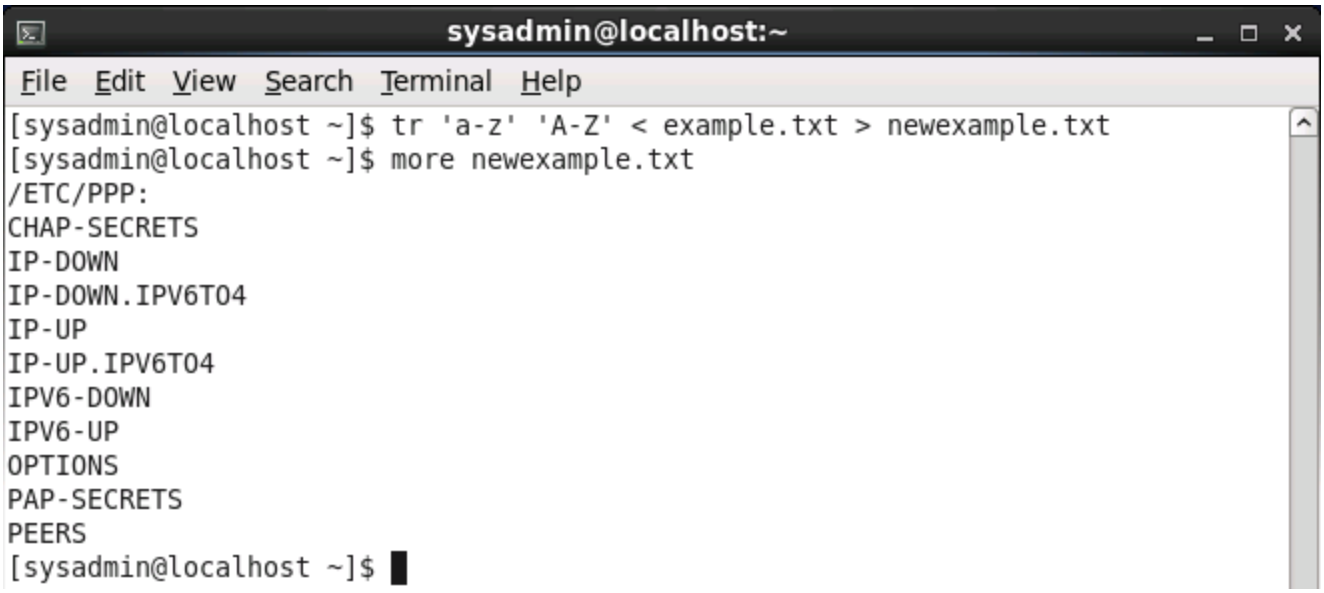


You can, however, tell the shell to get STDIN from a file instead of from the keyboard by using the < character:



This is fairly rare because most commands do accept filenames as arguments. But, for those that do not, this method could be used to have the shell read from the file instead of relying on the command to have this ability.

One last note: In most cases you probably want to take the resulting output and place it back into another file:



8.5 Searching for Files Using the Find Command

One of the challenges that users face when working with the filesystem, is trying to recall the location where files are stored. There are thousands of files and hundreds of directories on a typical Linux filesystem, so recalling where these files are located can pose challenges.

Keep in mind that most of the files that you will work with are ones that you create. As a result, you often will be looking in your own home directory to find files. However, sometimes you may need to search in other places on the filesystem to find files created by other users.

The `find` command is a very powerful tool that you can use to search for files on the filesystem. This command can search for files by name, including using wildcard characters for when you are not certain of the exact filename. Additionally, you can search for files based on file metadata, such as file type, file size and file ownership.

The syntax of the `find` command is:

```
find [starting directory] [search option] [ search criteria] [result option]
```

A description of all of these components:

Component	Description
[starting directory]	This is where the user specifies where to start searching. The `find` command will search this directory and all of its subdirectories. If no starting directory is

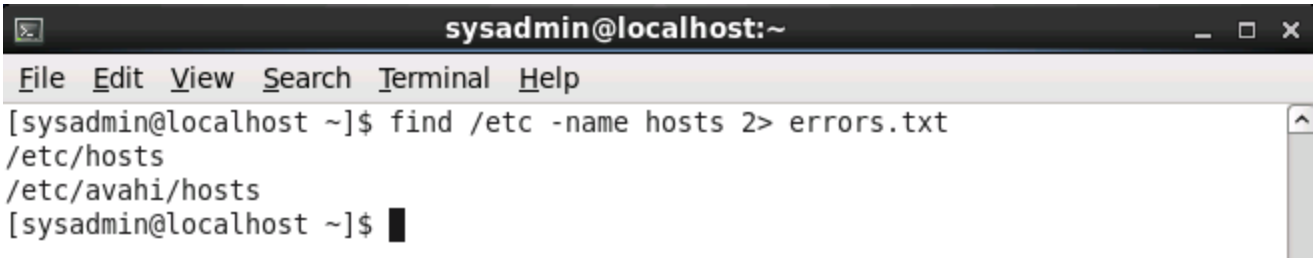
Component	Description
	provided, then the current directory is used for the starting point.
[search option]	This is where the user specifies an option to determine what sort of metadata to search for; there are options for file name, file size and many other file attributes.
[search criteria]	This is an argument that compliments the search option. For example, if the user uses the option to search for a file name, the search criteria would be the filename.
[result option]	This option is used to specify what action should be taken once the file is found. If no option is provided, the file name will be printed to STDOUT.

8.5.1 Search by File Name

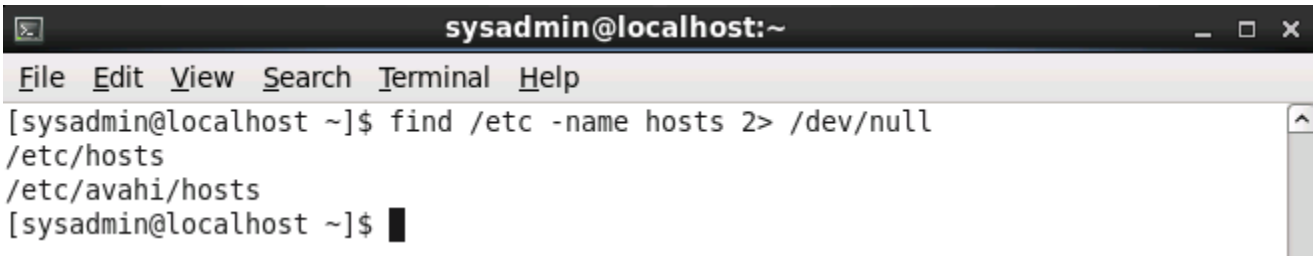
To search for a file by name, use the `-name` option to the `find` command:

Note that two files were found: `/etc/hosts` and `/etc/avahi/hosts`. The rest of the output was STDERR messages because the user who ran the command didn't have the permission to access certain subdirectories.

Recall that you can redirect STDERR to a file so you don't need to see these error messages on the screen:



While the output is easier to read, there really is no purpose to storing the error messages in the `error.txt` file. The developers of Linux realized that it would be good to have a "junk file" to send unnecessary data; any file that you send to the `/dev/null` file is discarded:

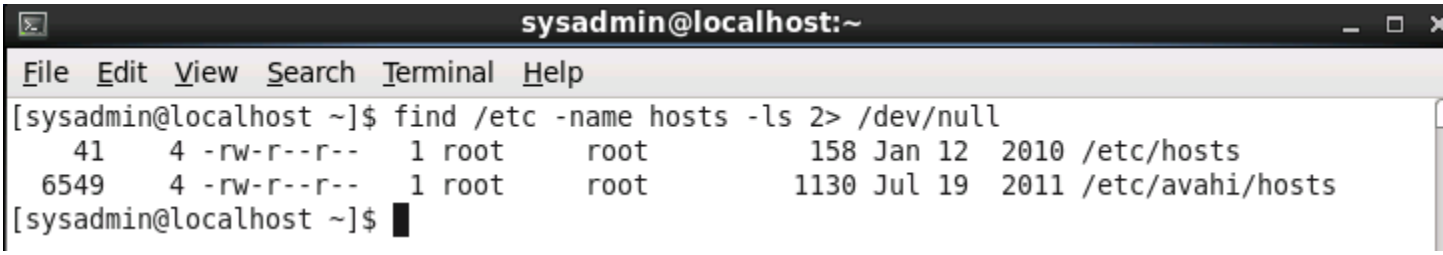


8.5.2 Displaying File Detail

It can be useful to obtain file details when using the `find` command because just the file name itself might not be enough information for you to find the correct file.

For example, there might be seven files named hosts; if you knew that the host file that you needed had been modified recently, then the modification timestamp of the file would be useful to see.

To see these file details, use the `-ls` option to the `find` command:

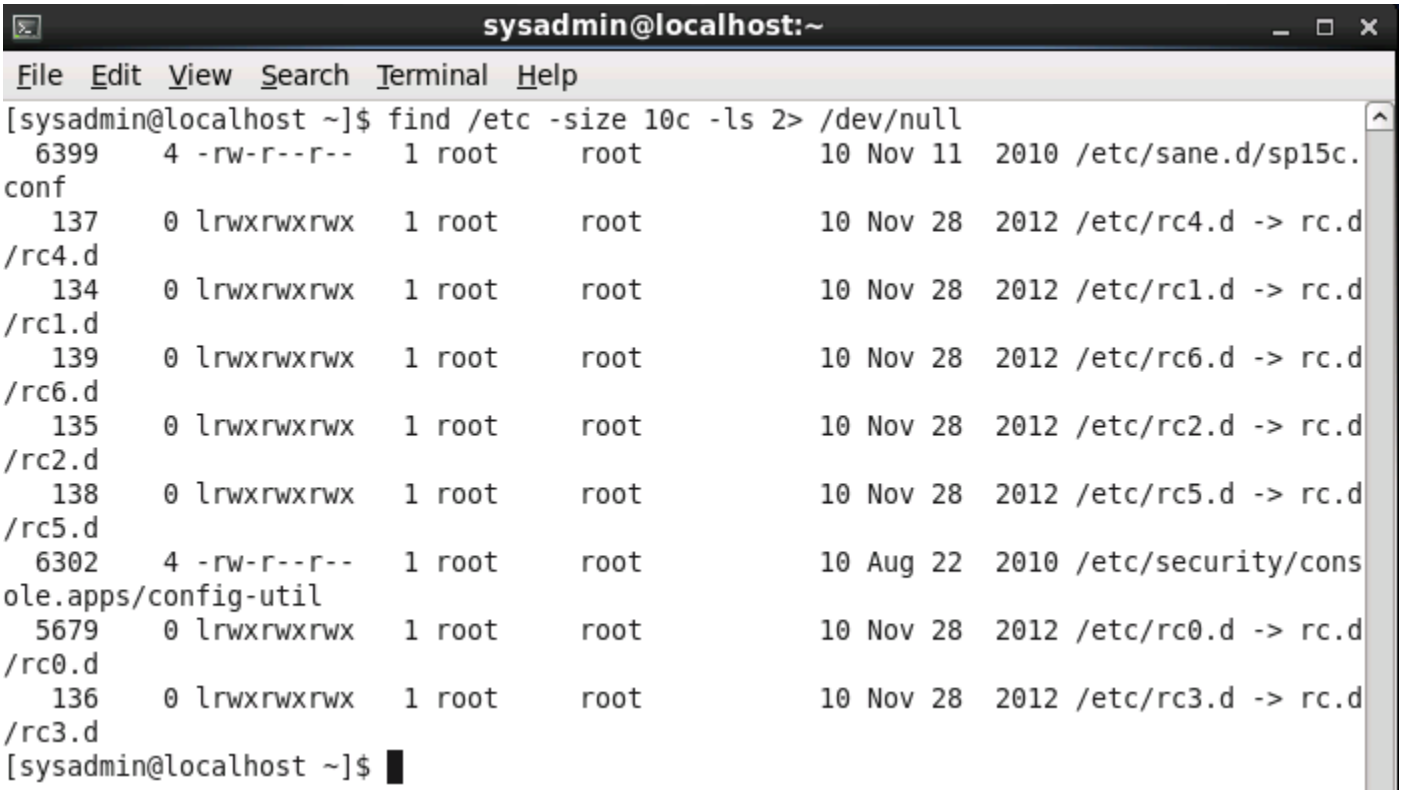


The first two columns of the output above are the *inode number* of the file and the number of *blocks* that the file is using for storage. Both of these are beyond the scope of the topic at hand. The rest of the columns are typical output of the `ls -l` command: file type, permissions, hard link count, user owner, group owner, file size, modification timestamp and file name.

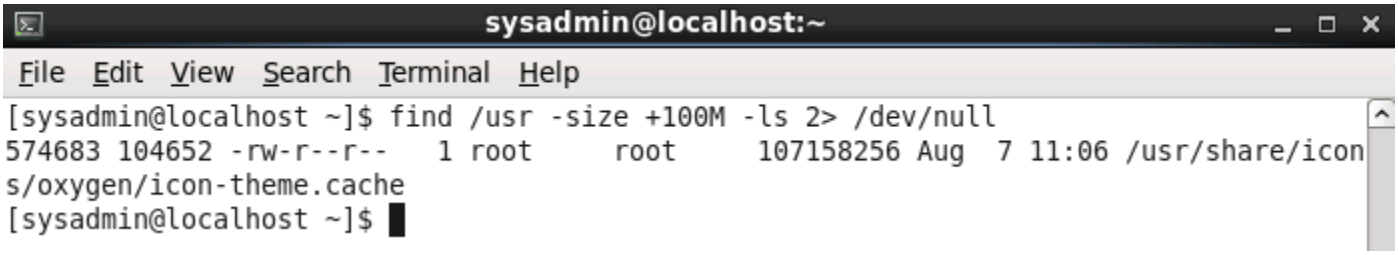
8.5.3 Searching for Files by Size

One of the many useful searching options is the option that allows you to search for files by size. The `-size` option allows you to search for files that are either larger than or smaller then a specified size as well as search for an exact file size.

When you specify a file size, you can give the size in bytes (c), kilobytes (k), megabytes (M) or gigabytes (G). For example, the following will search for files in the /etc directory structure that are exactly 10 bytes large:



If you want to search for files that are larger than a specified size, you place a + character before the size. For example, the following will look for all files in the `/usr` directory structure that are over 100 megabytes in size:



To search for files that are smaller than a specified size, place a - character before the file size.

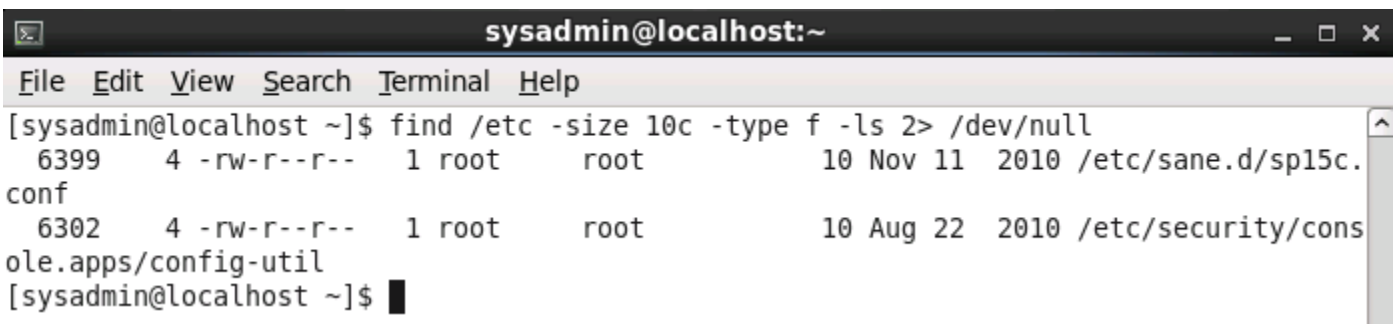
8.5.4 Additional Useful Search Options

There are many search options. The following table illustrates a few of these options:

Option	Meaning
-maxdepth	Allows the user to specify how deep in the directory structure to search. For example, -maxdepth 1 would mean only search the specified directory and its immediate subdirectories.
-group	Returns files owned by a specified group. For example, -group payroll would return files owned by the payroll group.
-iname	Returns files that match specified filename, but unlike -name, -iname is case insensitive. For example, -iname hosts would match files named hosts, Hosts, HOSTS, etc.
-mmin	Returns files that were modified based on modification time in minutes. For example, -mmin 10 would match files that were modified 10 minutes ago.
-type	Returns files that match file type. For example, -type f would return files that are regular files.
-user	Returns files owned by a specified user. For example, -user bob would return files owned by the bob user.

8.5.5 Using Multiple Options

If you use multiple options, they act as an "and", meaning for a match to occur, all of the criteria must match, not just one. For example, the following command will display all files in the `/etc` directory structure that are 10 bytes in size *and* are plain files:



8.6 Viewing Files Using the less Command

While viewing small files with the `cat` command poses no problems, it is not an ideal choice for large files. The `cat` command doesn't provide any way to easily pause and restart the display, so the entire file contents are dumped to the screen.

For larger files, you will want to use a *pager* command to view the contents. Pager commands will display one page of data at a time, allowing you to move forward and backwards in the file by using movement keys.

There are two commonly used pager commands:

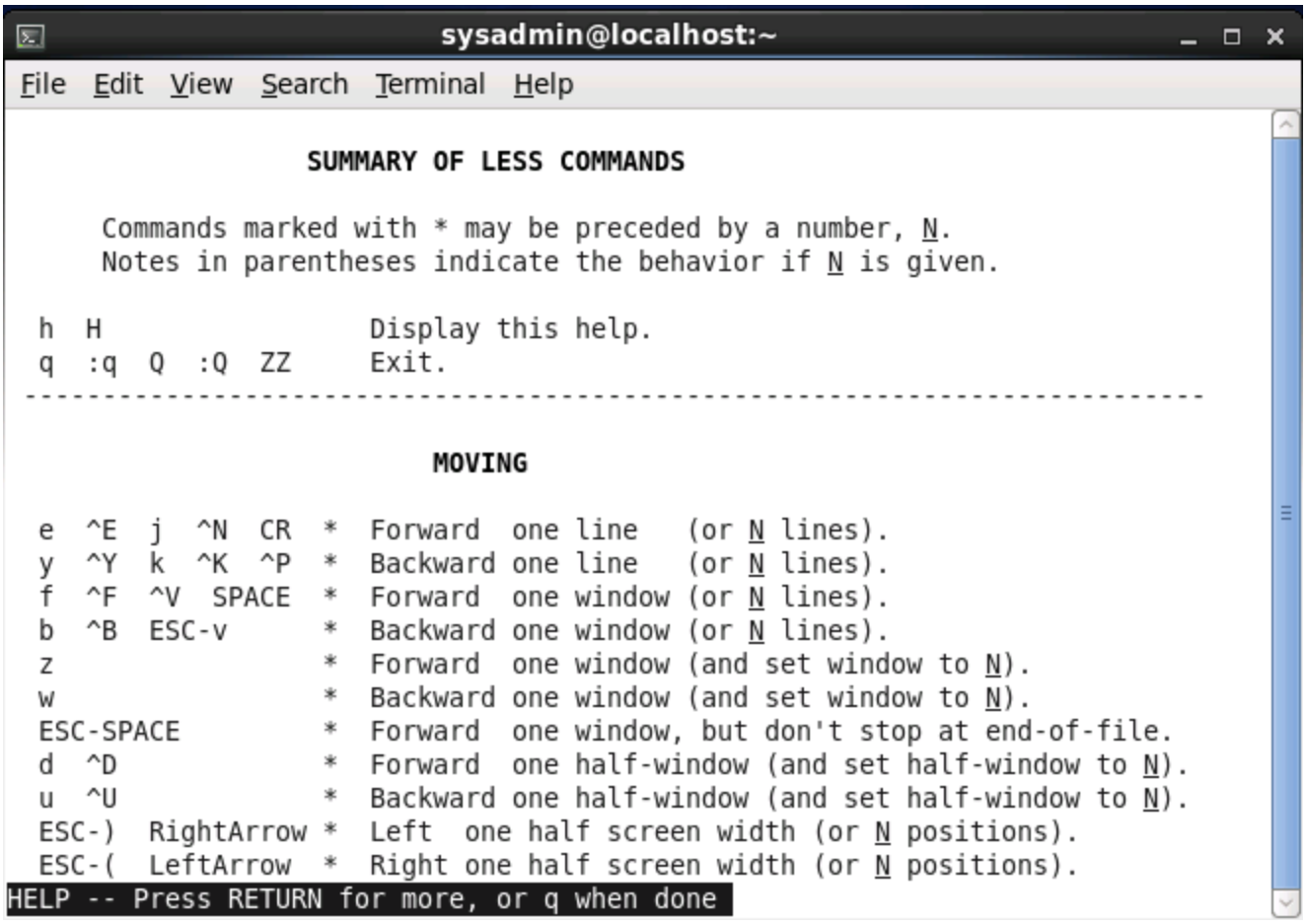
- The `less` command: This command provides a very advanced paging capability. It is normally the default pager used by commands like the man command.
- The `more` command: This command has been around since the early days of UNIX. While it has fewer features than the `less` command, it does have one important advantage: The `less` command isn't always included with all Linux distributions (and on some distributions, it isn't installed by default). The `more` command is always available.

When you use the `more` or `less` commands, they will allow you to "move around" a document by using *keystroke commands*. Because the developers of the `less` command based the command from the functionality of the `more` command, all of the keystroke commands available in the`more` command also work in the `less` command.

For the purpose of this manual, the focus will be on the more advanced command (`less`). The`more` command is still useful to remember for times when the `less` command isn't available. Remember that most of the keystroke commands provided work for both commands.

8.6.1 Help Screen in less

When you view a file with the `less` command, you can use the **h** key to display a help screen. The help screen allows you to see which other commands are available. In the following example, the `less /usr/share/dict/words` command is executed. Once the document is displayed, the**h** key was pressed, displaying the help screen:



8.6.2 less Movement Commands

There are many movement commands for the `less` command, each with multiple possible keys or key combinations. While this may seem intimidating, remember you don't need to memorize all of these movement commands; you can always use the **h** key whenever you need to get help.

The first group of movement commands that you may want to focus upon are the ones that are most commonly used. To make this even easier to learn, the keys that are identical in `more` and `less` will be summarized. In this way, you will be learning how to move in `more` and `less` at the same time:

Movement	Key
Window forward	Spacebar
Window backward	b

Movement	Key
Line forward	Enter
Exit	q
Help	h

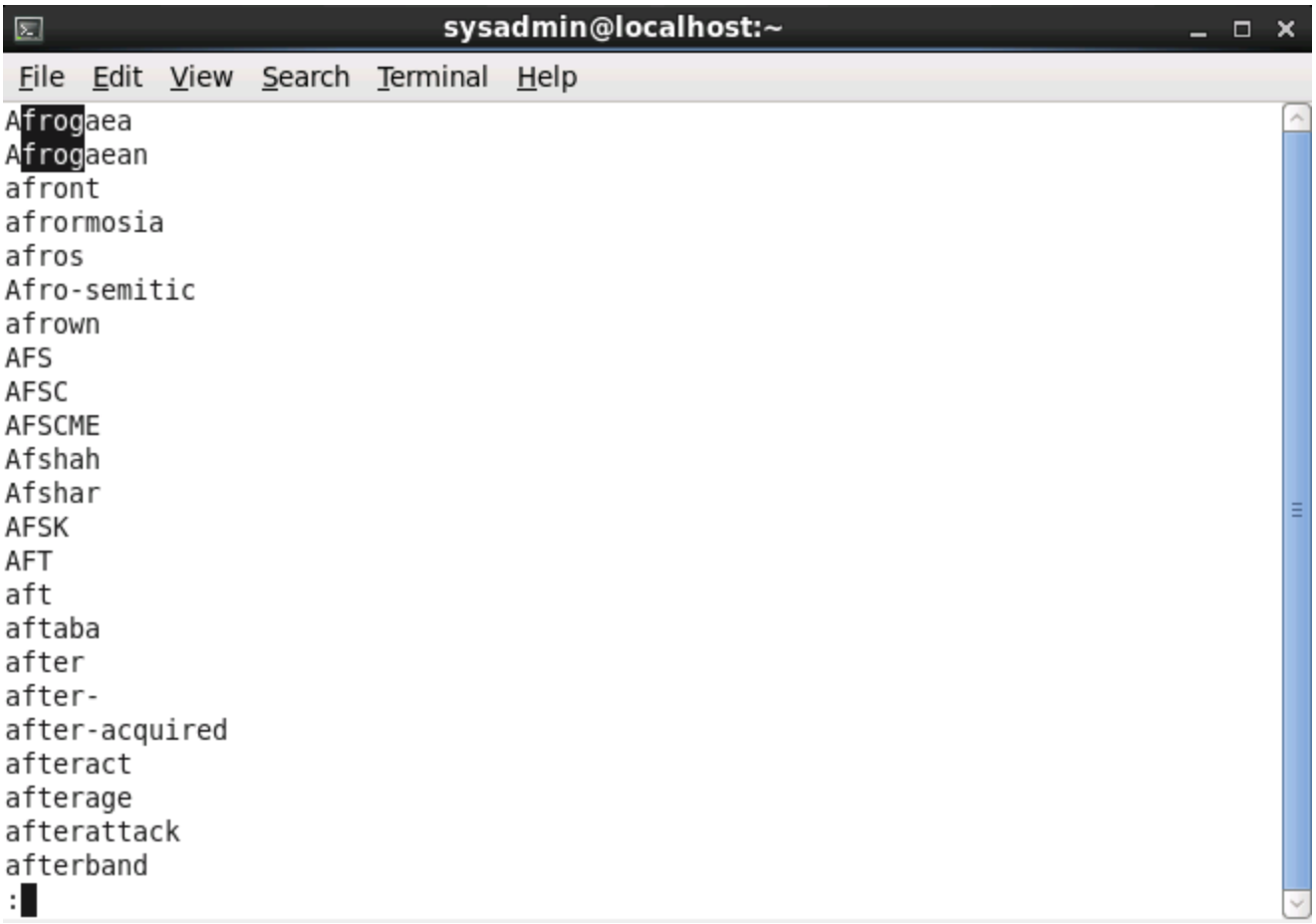
When simply using `less` as a pager, the easiest way to advance forward a page is to press the spacebar.

8.6.3 less Searching Commands

There are two ways to search in the `less` command: you can either search forward or backwards from your current position using patterns called regular expressions. More details regarding regular expressions are provided later in this chapter.

To start a search to look forward from your current position, use the ***/*** key. Then, type the text or pattern to match and press the **Enter** key.

If a match can be found, then your cursor will move in the document to the match. For example, in the following graphic the expression "frog" was searched for in the `/usr/share/dict/words` file:



Notice that "frog" didn't have to be a word by itself. Also notice that while the `less` command took you to the first match from the current position, all matches were highlighted.

If no matches forward from your current position can be found, then the last line of the screen will report "Pattern not found":

To start a search to look backwards from your current position, press the **?** key, then type the text or pattern to match and press the **Enter** key. Your cursor will move backward to the first match it can find or report that the pattern cannot be found.

If more than one match can be found by a search, then using the **n** key will allow you to move to the next match and using the **N** key will allow you to go to a previous match.

8.7 Revisiting the head and tail Commands

Recall that the `head` and `tail` commands are used to filter files to show a limited number of lines. If you want to view a select number of lines from the top of the file, you use the `head` command and if you want to view a select number of lines at the bottom of a file, then you use the `tail` command.

By default, both commands display ten lines from the file. The following table provides some examples:

Command Example	Explanation of Displayed Text
head /etc/passwd	First ten lines of /etc/passwd
head -3 /etc/group	First three lines of /etc/group
head -n 3 /etc/group	First three lines of /etc/group
help | head	First ten lines of output piped from the help command
tail /etc/group	Last ten lines of /etc/group
tail -5 /etc/passwd	Last five lines of /etc/passwd
tail -n 5 /etc/passwd	Last five lines of /etc/passwd
help | tail	Last ten lines of output piped from the help command

As seen from the above examples, both commands will output text from either a regular file or from the output of any command sent through a pipe. They both use the `-n` option to indicate how many lines to output.

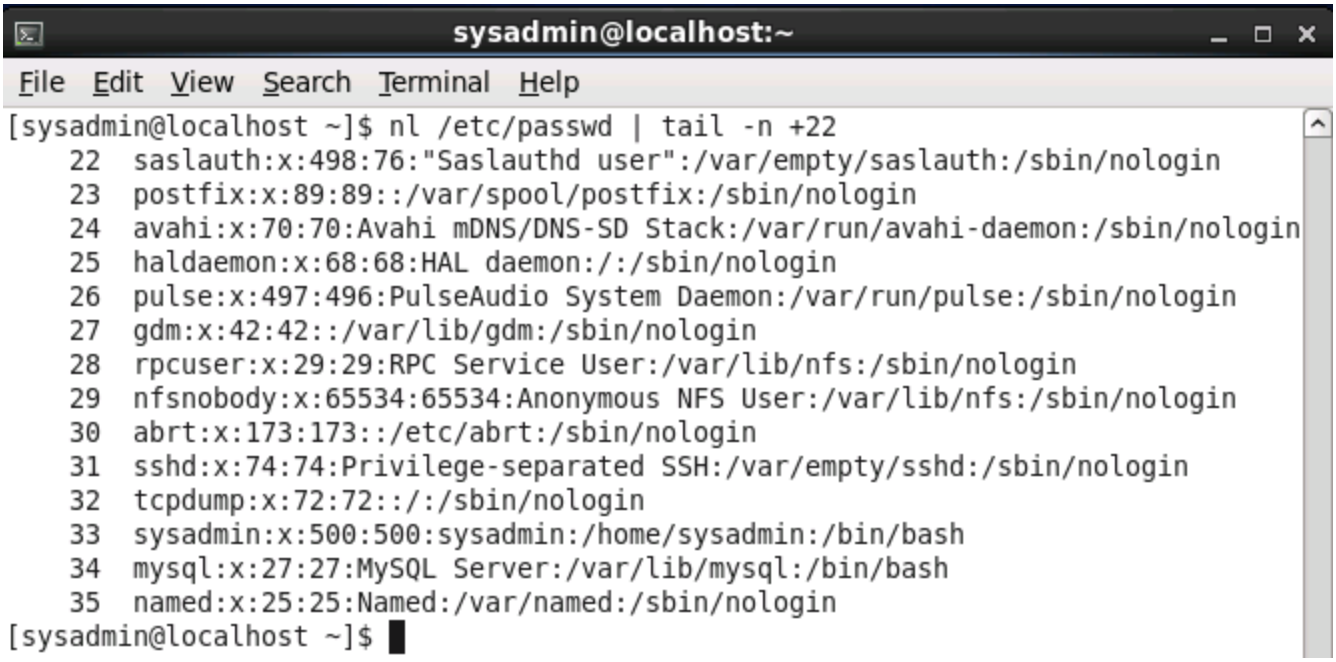
8.7.1 Negative Value with the -n Option

Traditionally in UNIX, the number of lines to output would be specified as an option with either command, so `-3` meant show three lines. For the `tail` command, either `-3` or `-n -3` still means show three lines. However, the GNU version of the `head` command recognizes `-n -3` as show **all but the first three lines**, and yet the `head` command still recognizes the option `-3` as show the first three lines.

8.7.2 Positive Value With the tail Command

The GNU version of the `tail` command allows for a variation of how to specify the number of lines to be printed. If you use the `-n` option with a number prefixed by the plus sign, then the `tail` command recognizes this to mean to display the contents starting at the specified line and continuing all the way to the end.

For example, the following will display line #22 to the end of the output of the `nl` command:



8.7.3 Following Changes to a File

You can view live file changes by using the `-f` option to the `tail` command. This is useful when you want to see changes to a file as they are happening.

A good example of this would be when viewing log files as a system administrator. Log files can be used to troubleshoot problems and administrators will often view them "interactively" with the `tail` command as they are performing the commands they are trying to troubleshoot in a separate window.

For example, if you were to log in as the root user, you could troubleshoot issues with the email server by viewing live changes to its log file with the following command: `tail -f /var/log/mail.log`

8.8 Sorting Files or Input

The `sort` command can be used to rearrange the lines of files or input in either dictionary or numeric order based upon the contents of one or more fields. Fields are determined by a field separator contained on each line, which defaults to whitespace (spaces and tabs).

The following example creates a small file, using the `head` command to grab the first 5 lines of the `/etc/passwd` file and send the output to a file called `mypasswd`.

```
sysadmin@localhost:~$ head -5 /etc/passwd > mypasswd
sysadmin@localhost:~$
```

```
sysadmin@localhost:~$ cat mypasswd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
sysadmin@localhost:~$
```

Now we will sort the `mypasswd` file:

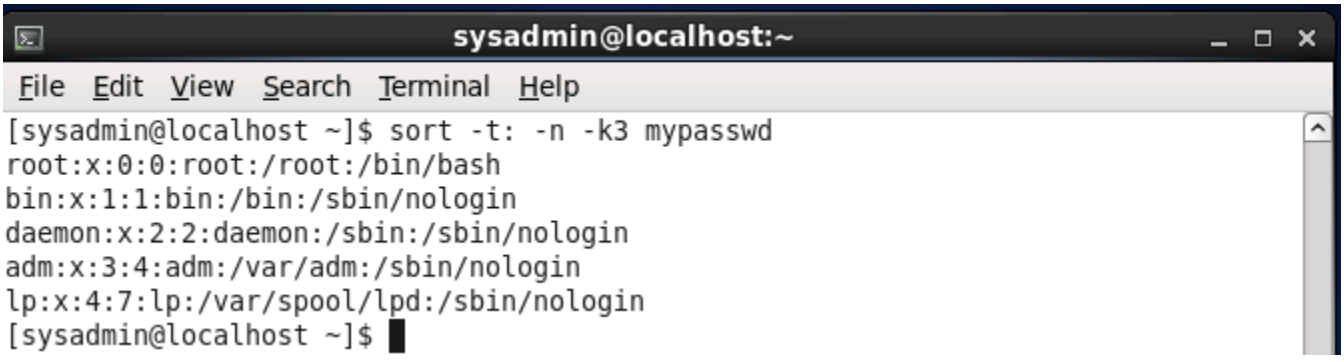
```
sysadmin@localhost:~$ sort mypasswd
bin:x:2:2:bin:/bin:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
root:x:0:0:root:/root:/bin/bash
sync:x:4:65534:sync:/bin:/bin/sync
sys:x:3:3:sys:/dev:/bin/sh
sysadmin@localhost:~$
```

8.8.1 Fields and Sort Options

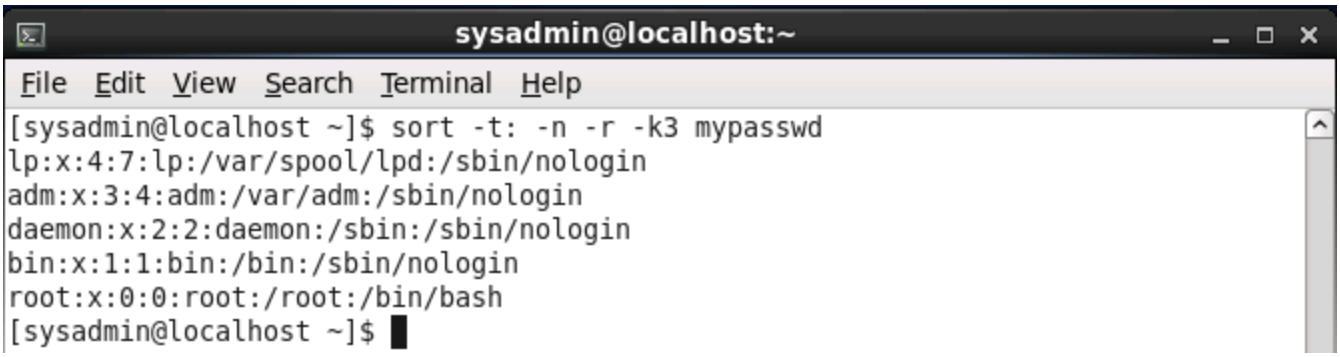
In the event that the file or input might be separated by another delimiter like a comma or colon, the `-t` option will allow for another field separator to be specified. To specify fields to sort by, use the `-k` option with an argument to indicate the field number (starting with 1 for the first field).

The other commonly used options for the `sort` command are the `-n` to perform a numeric sort and `-r` to perform a reverse sort.

In the next example, the `-t` option is used to separate fields by a colon character and performs a numeric sort using the third field of each line:



Note that the `-r` option could have been used to reverse the sort, making the higher numbers in the third field appear at the top of the output:



Lastly, you may want to perform more complex sorts, such as sort by a primary field and then by a secondary field. For example, consider the following data:

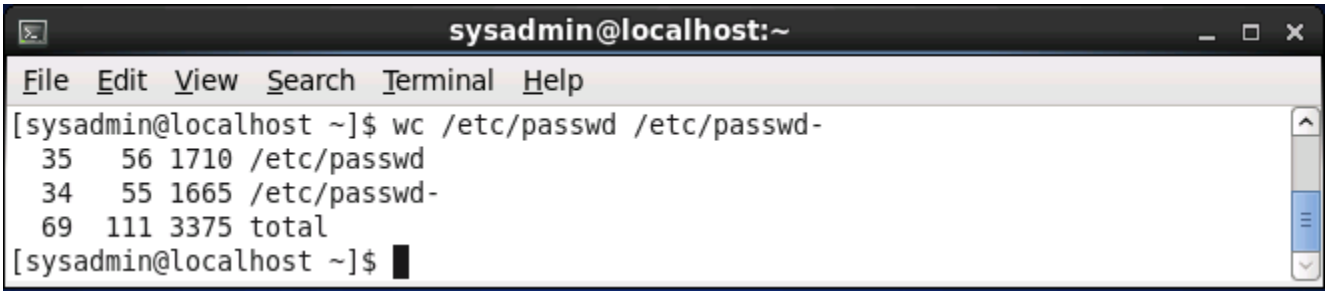
```
bob:smith:23
nick:jones:56
sue:smith:67
```

You might want to sort first by the last name (field #2) and then first name (field #1) and then by age (field #3). This can be done with the following command:

```
sort -t: -k2 -k1 -k3n filename
```

8.9 Viewing File Statistics With the wc Command

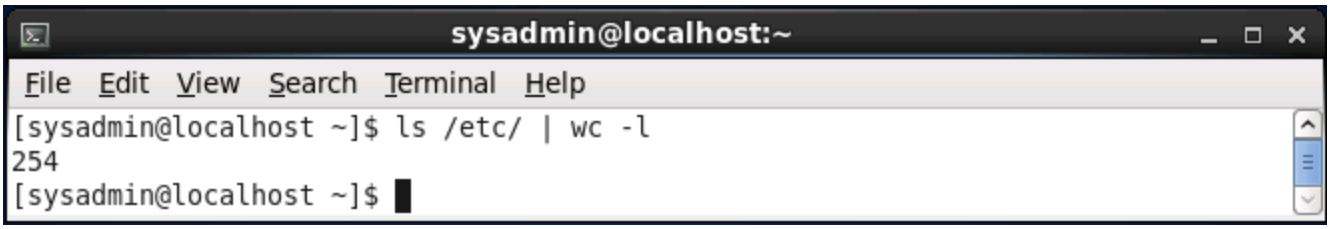
The `wc` command allows for up to three statistics to be printed for each file provided, as well as the total of these statistics if more than one filename is provided. By default, the wc command provides the number of lines, words and bytes (1 byte = 1 character in a text file):



The above example shows the output from executing: `wc /etc/passwd /etc/passwd-`. The output has four columns: number of lines in the file, number of words in the file, number of bytes in the file and the file name or "total".

If you are interested in viewing just specific statistics, then you can use `-l` to show just the number of lines, `-w` to show just the number of words and `-c` to show just the number of bytes.

The `wc` command can be useful for counting the number of lines output by some other command through a pipe. For example, if you wanted to know the total number of files in the `/etc` directory, you could execute `ls /etc | wc -l`:



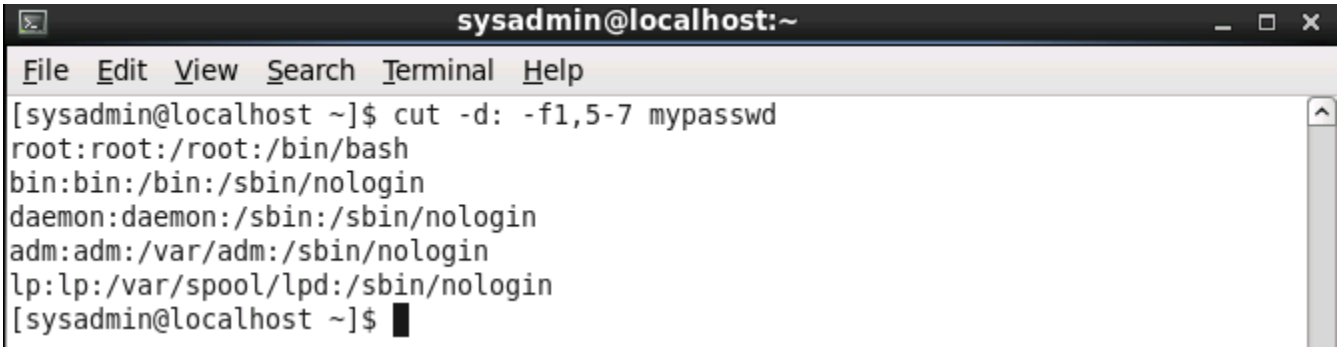
8.10 Using the cut Command to Filter File Contents

The `cut` command can extract columns of text from a file or standard input. A primary use of the `cut` command is for working with delimited database files. These files are very common on Linux systems.

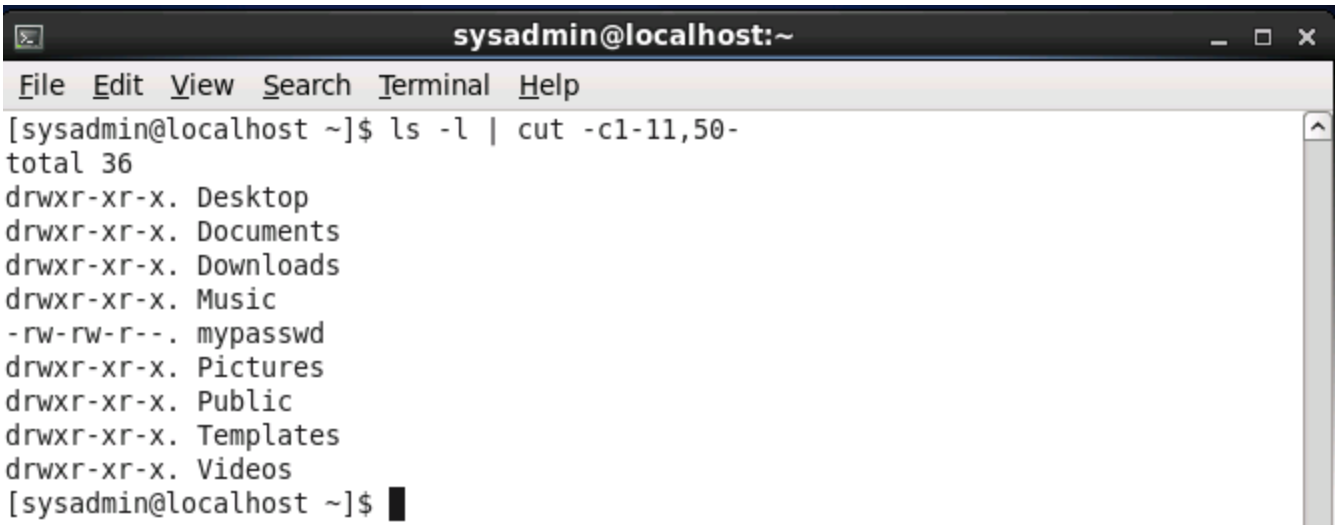
By default, it considers its input to be separated by the **Tab** character, but the `-d` option can specify alternative delimiters such as the colon or comma.

Using the `-f` option, you can specify which fields to display, either as a hyphenated range or a comma separated list.

In the following example, the first, fifth, sixth and seventh fields from `mypasswd` database file are displayed:



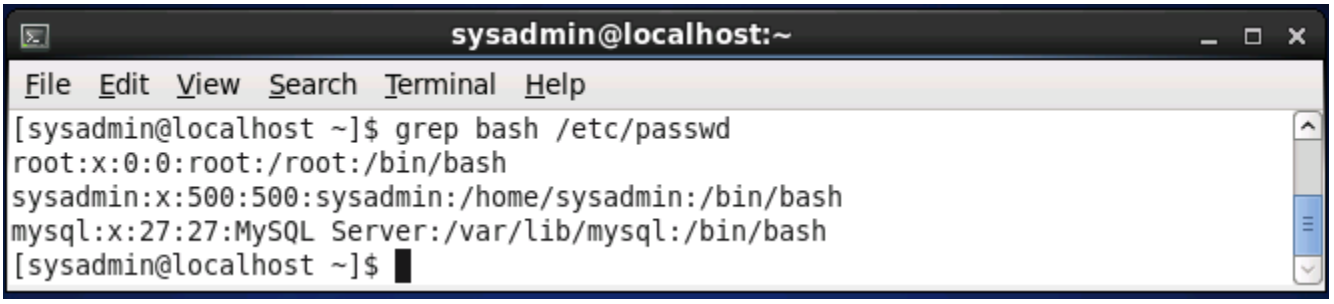
Using the `cut` command, you can also extract columns of text based upon character position with the `-c` option. This can be useful for extracting fields from fixed-width database files. For example, the following will display just the file type (character #1), permissions (characters #2-10) and filename (characters #50+) of the output of the `ls -l` command:



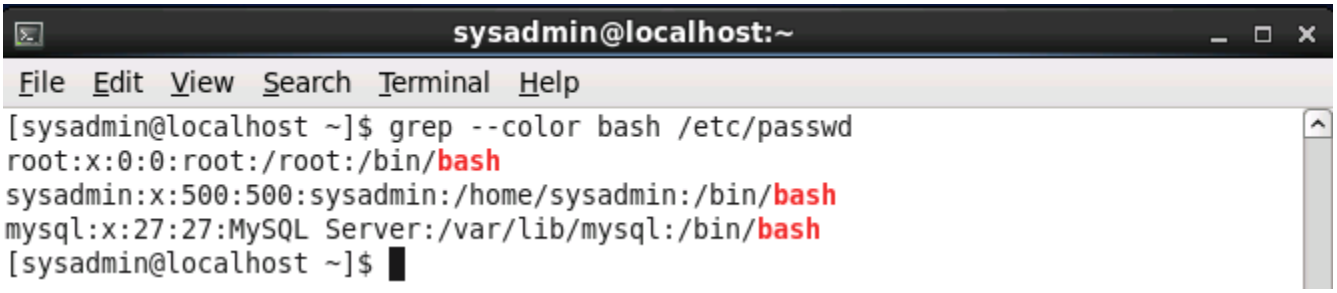
8.11 Using the grep Command to Filter File Contents

The `grep` command can be used to filter lines in a file or the output of another command based on matching a pattern. That pattern can be as simple as the exact text that you want to match or it can be much more advanced through the use of regular expressions (discussed later in this chapter).

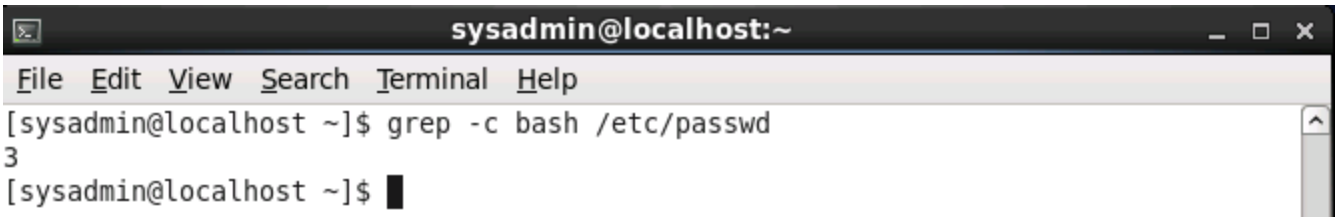
For example, you may want to find all the users who can login to the system with the BASH shell, so you could use the `grep` command to filter the lines from the `/etc/passwd` file for the lines containing the characters "bash":



To make it easier to see what exactly is matched, use the `--color` option. This option will highlight the matched items in red:

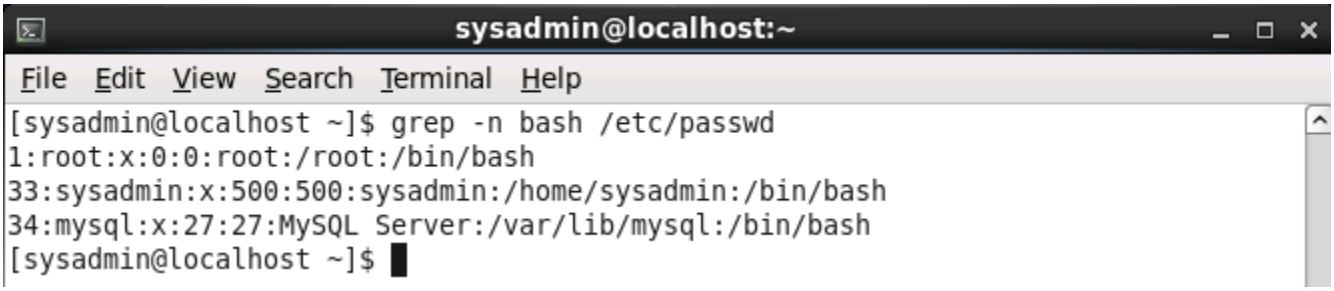


In some cases you don't care about the specific lines that match the pattern, but rather how many lines match the pattern. With the `-c` option, you can get a count of how many lines that match:



When you are viewing the output from the `grep` command, it can be hard to determine the original line numbers. This information can be useful when you go back into the file (perhaps to edit the file) as you can use this information to quickly find one of the matched lines.

The `-n` option to the `grep` command will display original line numbers:



Some additional useful `grep` options:

Examples	Output
grep -v nologin /etc/passwd	All lines not containing "nologin" in the /etc/passwd file
grep -l linux /etc/*	List of files in the /etc directory containing "linux"
grep -i linux /etc/*	Listing of lines from files in the /etc directory containing any case (capital or lower) of the character pattern "linux"
grep -w linux /etc/*	Listing of lines from files in the /etc directory containing the word pattern "linux"

8.12 Basic Regular Expressions

A *Regular Expression* is a collection of "normal" and "special" characters that are used to match simple or complex patterns. Normal characters are alphanumeric characters which match themselves. For example, an "a" would match an "a".

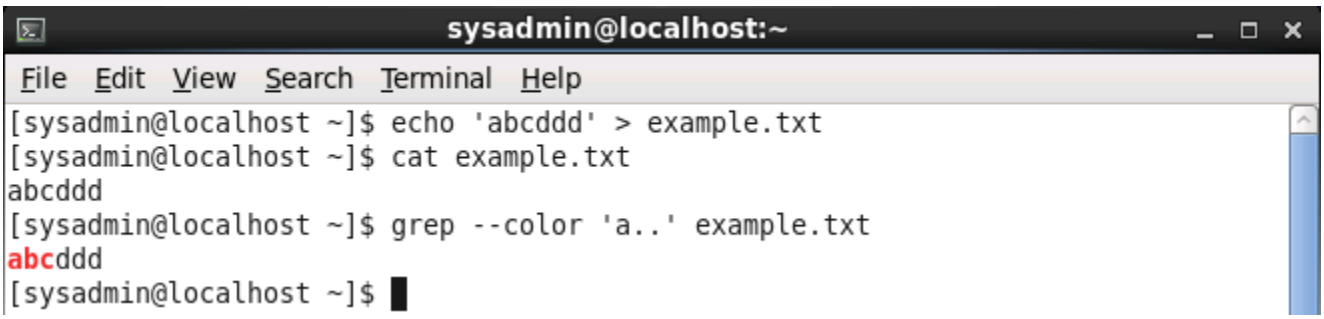
Some characters have special meanings when used within patterns by commands like the `grep` command. There are both *Basic Regular Expressions* (available to a wide variety of Linux commands) and *Extended Regular Expressions* (available to more advanced Linux commands). Basic Regular Expressions include the following:

Regular Expression	Matches
.	Any single character
[ ]	A list or range of characters to match one character, unless the first character is the caret "^", and then it means any character not in the list
*	Previous character repeated zero or more times
^	Following text must appear at beginning of line
$	Preceding text must appear at the end of the line

The `grep` command is just one of many commands that support regular expressions. Some other commands include the `more` and `less` commands. While some of the regular expressions are unnecessarily quoted with single quotes, it is a good practice to use single quotes around your regular expressions to prevent the shell from trying to interpret special meaning from them.

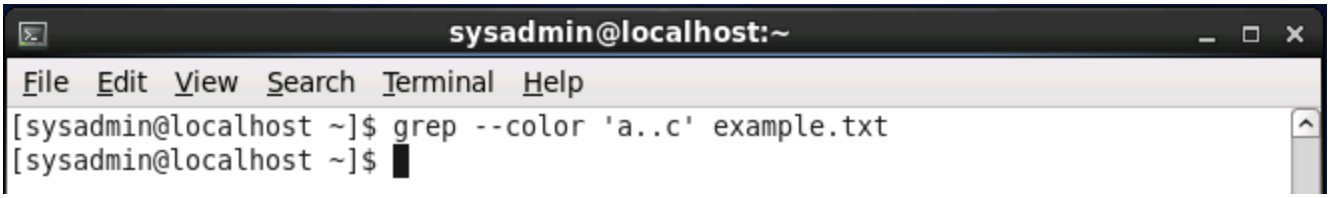
8.12.1 Basic Regular Expressions - the . Character

In the example below, a simple file is first created using redirection. Then the `grep` command is used to demonstrate a simple pattern match:



In the previous example, you can see that the pattern "a.." matched "abc". The first . character matched the "b" and the second matched the "c".

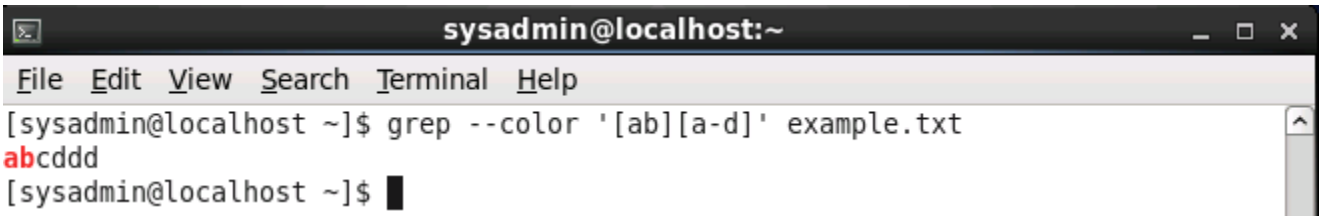
In the next example, the pattern "a..c" won't match anything, so the `grep` command will not product any output. For the match to be successful, there would need to be two characters between the "a" and the "c":



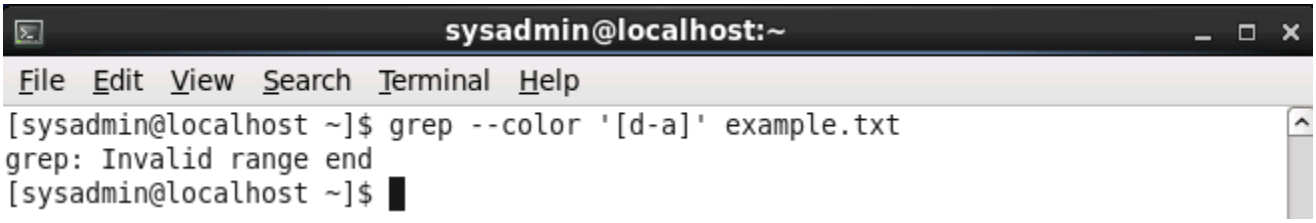
8.12.2 Basic Regular Expressions - the [ ] Characters

If you use the . character, then any possible character could match. In some cases you want to specify exactly which characters you want to match. For example, maybe you just want to match a lower-case alpha character or a number character. For this, you can use the [ ] Regular Expression characters and specify the valid characters inside the [ ] characters.

For example, the following command matches two characters, the first is either an "a" or a "b" while the second is either an "a", "b", "c" or "d":



Note that you can either list out each possible character ([abcd]) or provide a range ([a-d]) as long as the range is in the correct order. For example, [d-a] wouldn't work because it isn't a valid range:



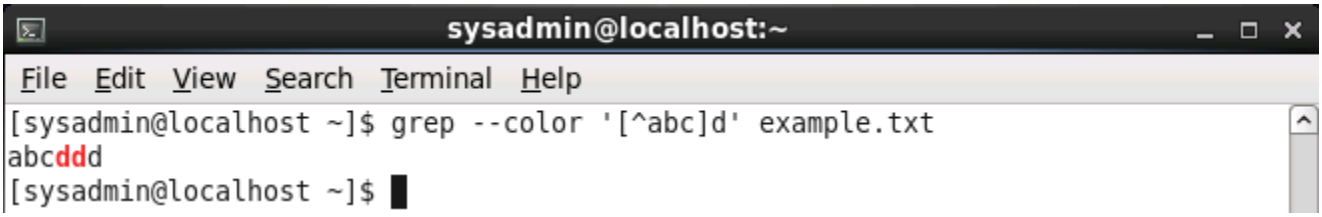
The range is specified by a standard called the ASCII table. This table is a collection of all printable characters in a specific order. You can see the ASCII table with the `man ascii` command. A small example:

```
041   33    21    !                141   97    61    a
042   34    22    "                142   98    62    b
043   35    23    #                143   99    63    c
044   36    24    $                144   100   64    d
045   37    25    %                145   101   65    e
046   38    26    &                146   102   66    f
```

Since "a" has a smaller numeric value (141) then "d" (144), the range a-d includes all characters from "a" to "d".

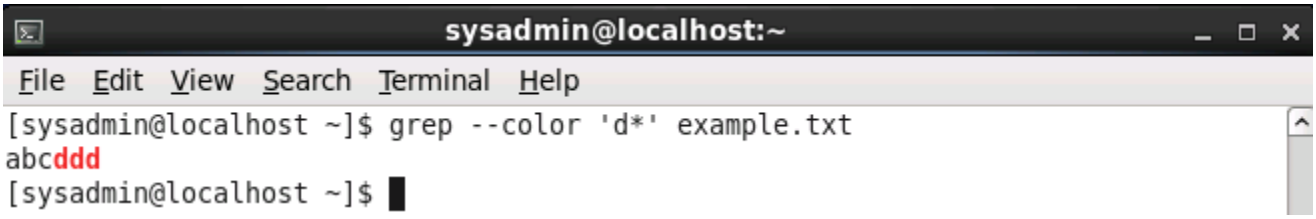
What if you want to match a character that can be anything but an "x", "y" or "z"? You wouldn't want to have to provide a [ ] set with all of the characters except "x", "y" or "z".

To indicate that you want to match a character that is not one of the listed characters, start your [ ] set with a ^ symbol. For example, the following will demonstrate matching a pattern that includes a character that isn't an "a", "b" or "c" followed by a "d":



8.12.3 Basic Regular Expressions - the * Character

The * character can be used to match "zero or more of the previous character". For example, the following will match zero or more "d" characters:



8.12.4 Basic Regular Expressions - the ^ and $ Characters

When you perform a pattern match, the match could occur anywhere on the line. You may want to specify that the match occurs at the beginning of the line or the end of the line. To match at the beginning of the line, begin the pattern with a ^ symbol.

In the following example, another line is added to the example.txt file to demonstrate the use of the ^ symbol:

```
sysadmin@localhost:~$ echo "xyzabc" >> example.txt
sysadmin@localhost:~$ cat example.txt
abcddd
xyzabc
sysadmin@localhost:~$ grep --color "a" example.txt
abcddd
xyzabc
sysadmin@localhost:~$ grep --color "^a" example.txt
abcddd
sysadmin@localhost:~$
```

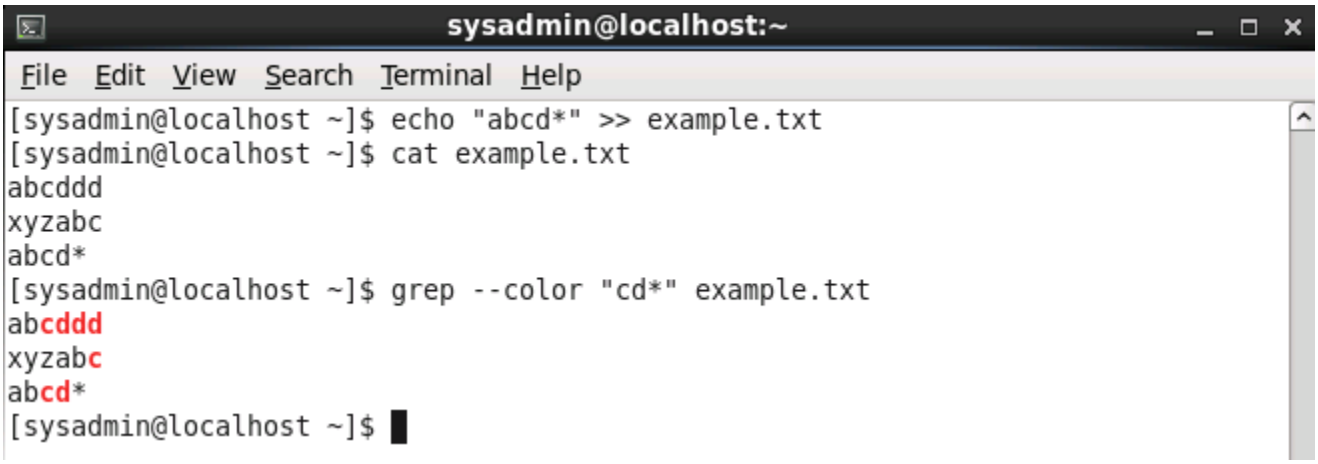
Note that in the first grep output, both lines match because they both contain the letter "a". In the second grep output, only the line that began with the letter "a" matched.

In order to specify the match occurs at the end of line, end the pattern with the $ character. For example, in order to only find lines which end with the letter "c":

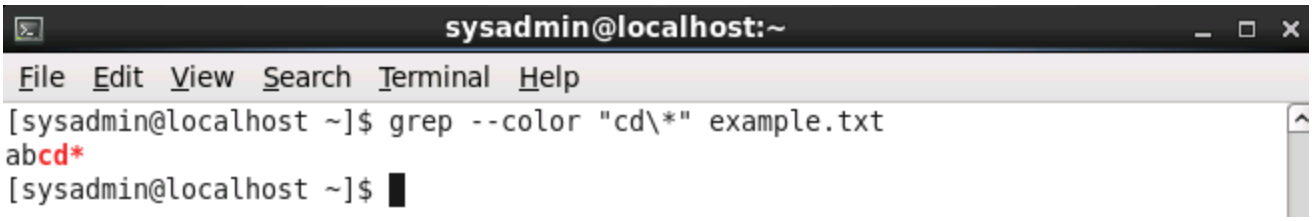
```
sysadmin@localhost:~$ grep "c$" example.txt
xyzabc
sysadmin@localhost:~$
```

8.12.5 Basic Regular Expressions - the \ Character

In some cases you may want to match a character that happens to be a special Regular Expression character. For example, consider the following:



In the output of the `grep` command above, you will see that every line matches because you are looking for a 'c' character followed by zero or more 'd' characters". If you want to look for an actual * character, place a \ character before the * character:



8.13 Extended Regular Expressions

The use of Extended Regular Expressions often requires a special option be provided to the command to recognize them. Historically, there is a command called `egrep`, which is similar to grep, but is able to understand their usage. Now, the `egrep` command is deprecated in favor of using grep with the `-E` option.

The following regular expressions are considered "extended":

RE	Meaning
?	Matches previous character zero or one time, so it is an optional character
+	Matches previous character repeated one or more times
|	Alternation or like a logical or operator

Some extended regular expressions examples:

Command	Meaning	Matches
grep -E 'colou?r' 2.txt	Match 'colo' following by zero or one 'u' character	color colour
grep -E 'd+' 2.txt	Match one or more 'd' characters	d dd ddd dddd
grep -E 'gray|grey' 2.txt	Match either 'gray' or 'grey'	gray grey

8.14 xargs Command

If you receive an error about an argument list being too long when trying to execute a command, then it's probably time to think about using `xargs` with that command. The `xargs` command also has a useful option `-0` that helps to eliminate problems with files that have spaces or tabs in their names.

The `xargs` command is useful for allowing commands to be executed more efficiently. Its goal is to build the command line for a command to execute as few as times as possible with as many arguments as possible, rather than to execute the command many times with one argument each time.

The following example shows a scenario where the `xargs` command allowed for many files to be removed, where using a normal wildcard (glob) character failed: