Unit -IV

Prepared by

Dr.N.Gobi

Syllabus- Unit IV

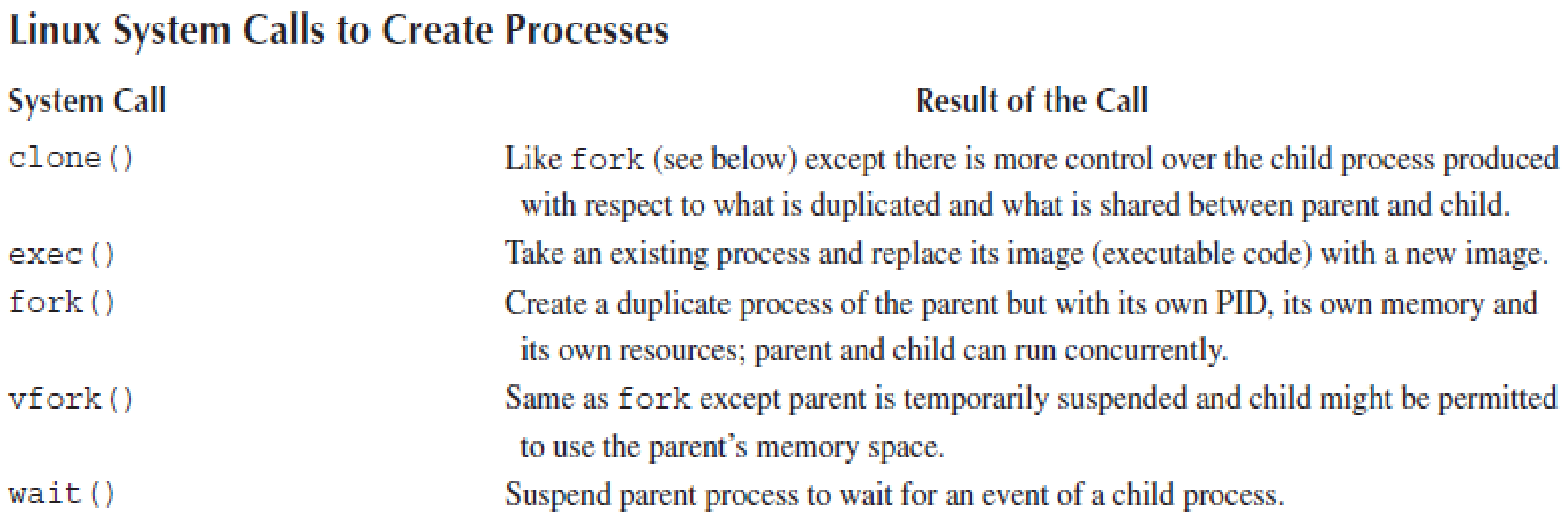
Linux Process Management:

Forms of Process Management-Starting, Pausing, and Resuming Processes-Monitoring Processes-Managing Linux Processes-Linux Applications: Text EditorsLatex-Email Programs- Network Software-Regular Expressions: Grep-Sed-Awk.

Introduction

- *Process* is a running program.
- Program is static whereas Process is dynamic which changes its states over time.
- In Linux, processes can be started from the GUI or the command line or by other running processes.
- Whenever a process runs, the Linux kernel keeps track of it through a process ID (*PID*).
- Linux kernel is loaded and running, the first process it launches is called system[PID is 1].
- systemd is responsible for starting the run-time environment and then monitoring the environment while in use.

- In Linux, a process can only be created by another process (with the exception of systemd). We refer to the creating process as the *parent* and the created process as the *child.*



Linux System Calls to Create Processes

System Call	Result of the Call
`clone()`	Like `fork` (see below) except there is more control over the child process produced with respect to what is duplicated and what is shared between parent and child.
`exec()`	Take an existing process and replace its image (executable code) with a new image.
`fork()`	Create a duplicate process of the parent but with its own PID, its own memory and its own resources; parent and child can run concurrently.
`vfork()`	Same as `fork` except parent is temporarily suspended and child might be permitted to use the parent's memory space.
`wait()`	Suspend parent process to wait for an event of a child process.

- **fork** is best for independent processes.
- **vfork** is faster than fork for processes that immediately exec.
- **exec** is for transforming a process into a new program.
- **clone** is ideal for creating threads with controlled resource sharing.

FORMS OF PROCESS MANAGEMENT

- Consider that a program is a piece of software. It contains code and has memory space reserved for data storage.
- Most of this storage is dedicated to variables that are declared and used in the program.
- Program exists in one of two states: the original source code and the executable code. In either case, the program is a *static* entity.

- Process is an *active* entity.
- As the process executes, the values it stores in memory (variables) change.
- As the process executes, it may request resources from the operating system which it then holds, uses and returns to the operating system.
- CPU stores information about the running process via its registers.
- Program Counter (PC) stores the address of the next instruction to fetch from memory while the Instruction register (IR) stores the current instruction.

- Results of the most recent operation are indicated using status flags in the Status Flag (SF) register .
- A run-time stack is maintained for this process, indicating subroutine invocation and parameter passing. The top of the stack is pointed to by the Stack Pointer (SP).

Process Control Block Information

Item	Use
Process state	The run-time state of the process; usually one of `new`, `ready`, `running`, `suspended`, `terminating`, `waiting`.
Process ID	Usually a numeric designator to differentiate the process from others.
Other process data	Parent process (if there is one), process owner, priority and/or scheduling information, accounting data such as amount of CPU time elapsed.
Process location	Which queue the process currently resides in (ready queue, wait queue, job queue).
Process privilege/state	What mode the process runs in (user, privileged, some other).
Hardware-stored values	PC, IR, SF, SP (and possibly data register values) and interrupt masks.
Resource allocation	I/O devices currently allocated to the process; memory in use; page table.

Process Management.

- Handling the startup and termination of, scheduling and movement of, and interprocess communication of processes.

Single-process execution

- *single tasking*, the operating system starts a process and then the CPU’s attention is entirely held by that one running process.
- CPU executes this one process until either the process terminates or requests some operation by the operating system.
- Among them, the user is limited to running one task at a time and so there is no way to switch back and forth between multiple tasks.
- if the process has to perform some time-consuming input or output, the CPU remains idle while the input/output (I/O) takes place.

Batch Processing

- *Batch processing* is much like single tasking .Batch processing, the computer is shared among multiple users such that users can submit processes to run at any time.
- Scheduling may be performed by some offline system
- Job may run any time from immediately to hours or days later, the job cannot be expected to run interactively.
- Scheduling algorithm can be used to schedule the processes.

Concurrent processing

- One of the great inefficiencies of both single tasking and batch processing is that of I/O.
- One process/job may wait for the i/o – time consuming if read from magnetic tapes.
- Idea of switching from a waiting process to a process ready for execution leads us to an improved form of process management called *multiprogramming*.
- In multiprogramming, the CPU executes a single process, but if that process needs to perform time-consuming I/O then that process is set aside and another process is executed.
- Concurrent Processing- By switching between processes in this way, it appears to the user that the computer is executing multiple processes at a time
- Context switching plays a major role.

For context switching, OS follows

- Pre-emptive
- Non-Preemptive
- Time sharing

Interrupt handling

- Process of interrupting the CPU as an *interrupt* and the process of requesting an interrupt as an *interrupt request* (IRQ).
- IRQ may originate from hardware or software. For hardware, the IRQ is carried over a reserved line on the bus connecting the hardware device to the CPU (or to an interrupt controller device). For software, an IRQ is submitted as an interrupt signal.
- Upon receiving an interrupt request, the CPU finishes the current fetch-execute cycle and then decides how to respond to the interrupt request.
- For every type of interrupt, the operating system contains an interrupt handler.
- Each *interrupt handler* is a piece of code written to handle a specific type of interrupting situation.

STARTING, PAUSING AND RESUMING PROCESSES

- Depending on the version of Linux and the GUI being used, there are menu selections and icons.
- Parent process making the request might be the GUI, Bash, another running application or a part of the operating system (e.g., a service).
- kernel’s process manager creates the process and prepares it for execution.
- When a process is launched by the user, the process runs under access rights based on its *Effective User ID* (EUID) and *Effective Group ID* (EGID).

Launching processes from the command line

- Launching processes from within a shell is accomplished by specifying the program name. → Absolute path or relative path.

Suspending and resuming processes from the command line

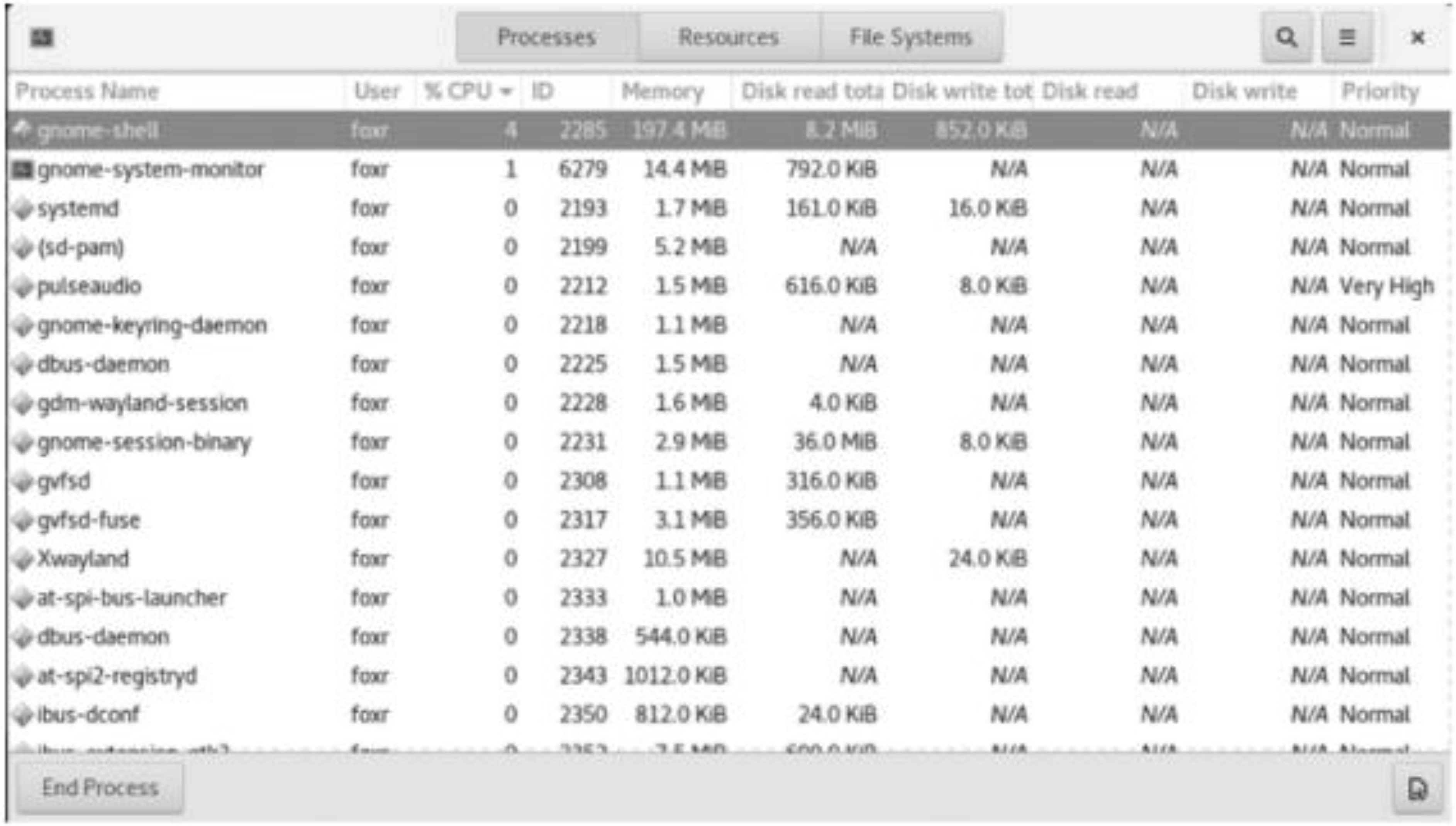
- Run as many processes in the background.
- A terminal window can only have one process running in the foreground at a time.
- Foreground process is the one that provides interaction with the user, whether to receive user input or to output information to the window, or both.
- Jobs command responds with all of the active jobs in the terminal window. These are processes that have been started from the command line but have not yet terminated.

- A process may be in a *stopped* state. This means that the process has been running but is currently suspended.
- User can resume a suspended process. In doing so, we specify whether to resume the process in the foreground or in the background.
- In order to stop a running, foreground process, type control+z in the terminal window

MONITORING PROCESSES

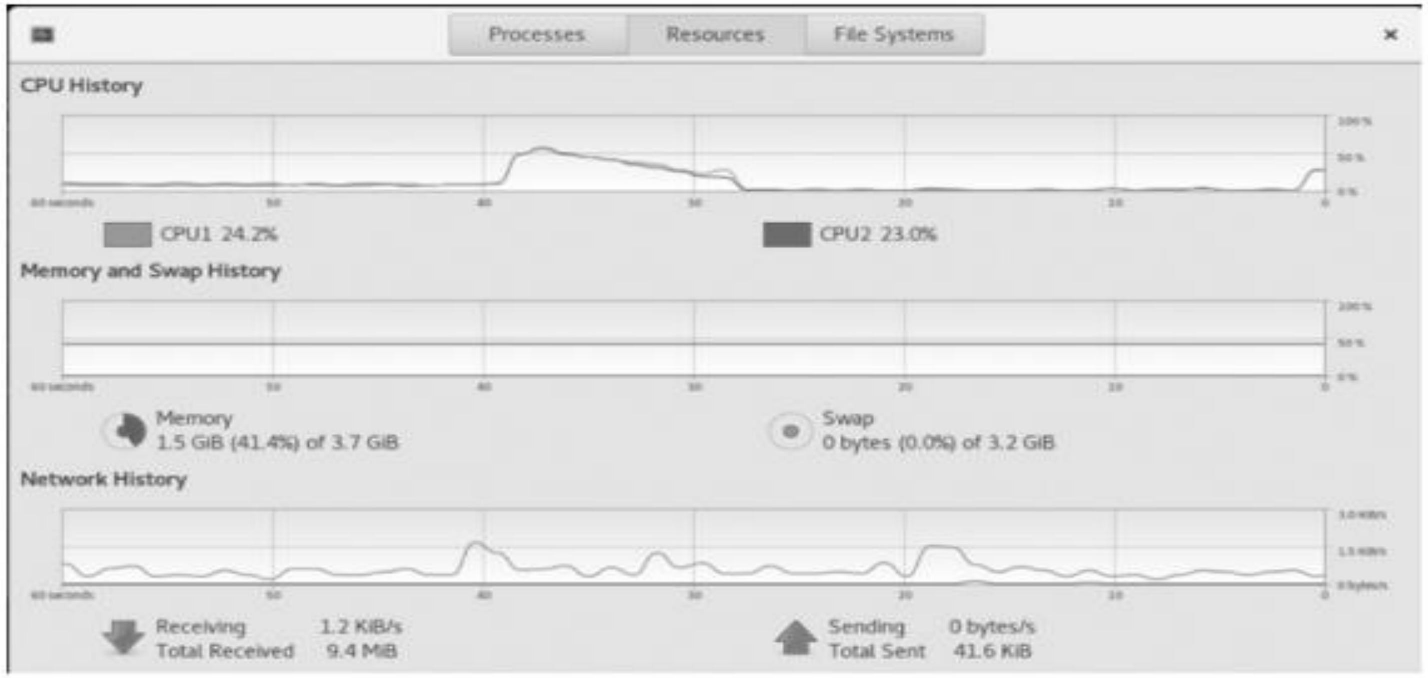
- Operating system can run multiple processes efficiently with little or no user intervention.
- Several different tools available to monitor processes and system resources.
- Primary GUI tool is called the System Monitor.
- Launching System Monitor from the GUI has changed.
- Two different versions of the System Monitor, one for Gnome and one for KDE (an alternative GUI desktop).

Gnome



Process Name	User	% CPU ▾	ID	Memory	Disk read tota	Disk write tot	Disk read	Disk write	Priority
gnome-shell	foxr	4	2285	197.4 MiB	8.2 MiB	852.0 KiB	N/A	N/A	Normal
gnome-system-monitor	foxr	1	6279	14.4 MiB	792.0 KiB	N/A	N/A	N/A	Normal
systemd	foxr	0	2193	1.7 MiB	161.0 KiB	16.0 KiB	N/A	N/A	Normal
(sd-pam)	foxr	0	2199	5.2 MiB	N/A	N/A	N/A	N/A	Normal
pulseaudio	foxr	0	2212	1.5 MiB	616.0 KiB	8.0 KiB	N/A	N/A	Very High
gnome-keyring-daemon	foxr	0	2218	1.1 MiB	N/A	N/A	N/A	N/A	Normal
dbus-daemon	foxr	0	2225	1.5 MiB	N/A	N/A	N/A	N/A	Normal
gdm-wayland-session	foxr	0	2228	1.6 MiB	4.0 KiB	N/A	N/A	N/A	Normal
gnome-session-binary	foxr	0	2231	2.9 MiB	36.0 MiB	8.0 KiB	N/A	N/A	Normal
gvfsd	foxr	0	2308	1.1 MiB	316.0 KiB	N/A	N/A	N/A	Normal
gvfsd-fuse	foxr	0	2317	3.1 MiB	356.0 KiB	N/A	N/A	N/A	Normal
Xwayland	foxr	0	2327	10.5 MiB	N/A	24.0 KiB	N/A	N/A	Normal
at-spi-bus-launcher	foxr	0	2333	1.0 MiB	N/A	N/A	N/A	N/A	Normal
dbus-daemon	foxr	0	2338	544.0 KiB	N/A	N/A	N/A	N/A	Normal
at-spi2-registryd	foxr	0	2343	1012.0 KiB	N/A	N/A	N/A	N/A	Normal
ibus-dconf	foxr	0	2350	812.0 KiB	24.0 KiB	N/A	N/A	N/A	Normal

- Resources tab provides a summary of CPU, memory/swap space and network utilization over time (the last minute).



CPU1 24.2%
CPU2 23.0%

Memory
1.5 GiB (41.4%) of 3.7 GiB

Swap
0 bytes (0.0%) of 3.2 GiB

Receiving 1.2 KiB/s
Total Received 9.4 MiB

Sending 0 bytes/s
Total Sent 41.6 KiB

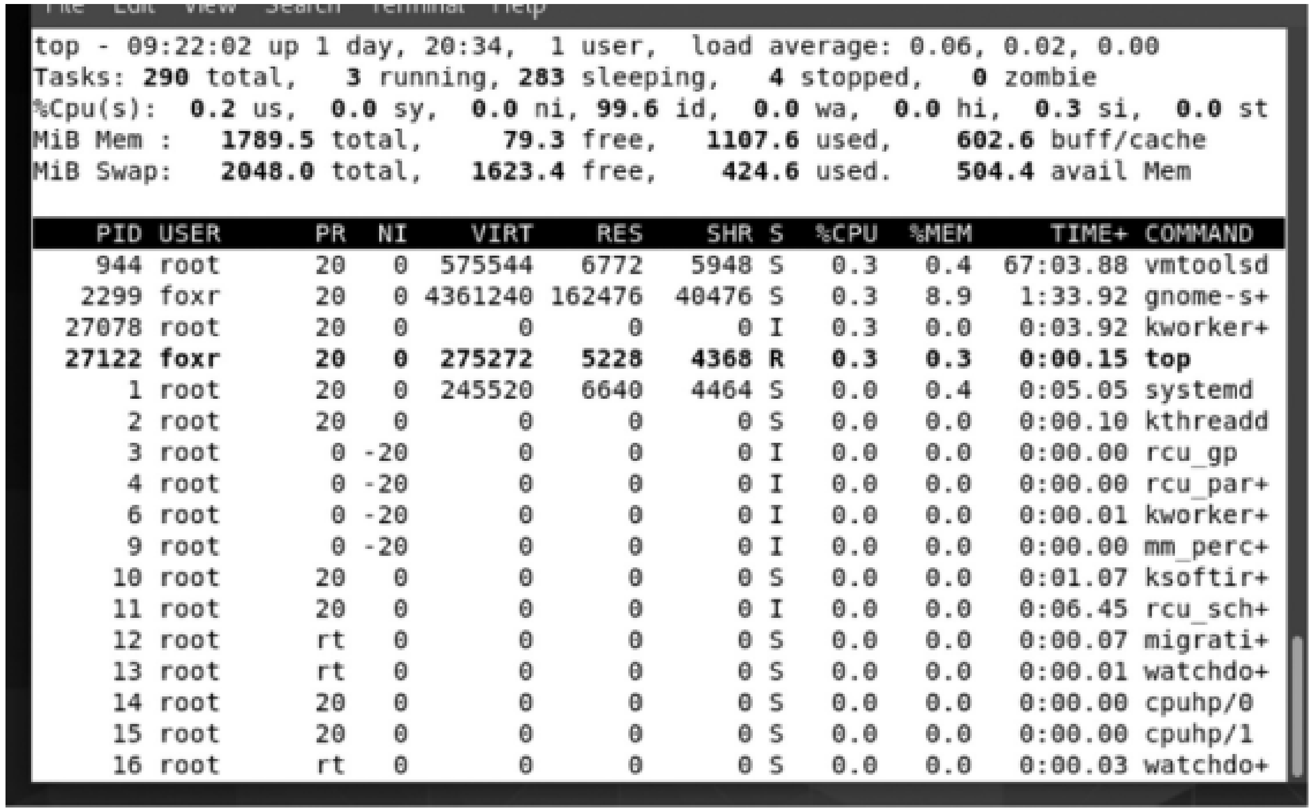
- File Systems tab displays statistics about the file system.

Command-Line Monitoring Tools

- System Monitor provides a convenient way to view system usage and control running processes, we might be more interested in exploring system usage from the command line for two reasons.
- similar to the idea of opening multiple windows to run multiple processes that we explored in the last section, using the System Monitor takes more resources than command-line programs.
- information we can get from the System Monitor is not as comprehensive as what can be found from some of the command-line programs. So, we turn to two of the primarily monitoring tools: top and ps

top program

- top program is launched from the command line but unlike most of the commands we have viewed, this command fills the terminal window with its output and remains running.
- top is an interactive program that updates its output in a specified refresh rate and changes its appearance based on input keystrokes.
- Default refresh rate is 3 seconds.



```
top - 09:22:02 up 1 day, 20:34,  1 user,  load average: 0.06, 0.02, 0.00
Tasks: 290 total,   3 running, 283 sleeping,   4 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.0 sy,  0.0 ni, 99.6 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
MiB Mem :   1789.5 total,     79.3 free,   1107.6 used,    602.6 buff/cache
MiB Swap:   2048.0 total,   1623.4 free,    424.6 used.    504.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    944 root      20   0  575544   6772   5948 S   0.3   0.4  67:03.88 vmtoolsd
   2299 foxr      20   0 4361240 162476  40476 S   0.3   8.9   1:33.92 gnome-s+
  27078 root      20   0       0      0      0 I   0.3   0.0   0:03.92 kworker+
  27122 foxr      20   0  275272   5228   4368 R   0.3   0.3   0:00.15 top
      1 root      20   0  245520   6640   4464 S   0.0   0.4   0:05.05 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.10 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par+
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.01 kworker+
      9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_perc+
     10 root      20   0       0      0      0 S   0.0   0.0   0:01.07 ksoftir+
     11 root      20   0       0      0      0 I   0.0   0.0   0:06.45 rcu_sch+
     12 root      rt   0       0      0      0 S   0.0   0.0   0:00.07 migrati+
     13 root      rt   0       0      0      0 S   0.0   0.0   0:00.01 watchdo+
     14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     15 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
     16 root      rt   0       0      0      0 S   0.0   0.0   0:00.03 watchdo+
```

ps command

- ps command provides a detailed examination of the running processes as a snapshot.
- Unlike top, it is not interactive. It displays the information and exits.

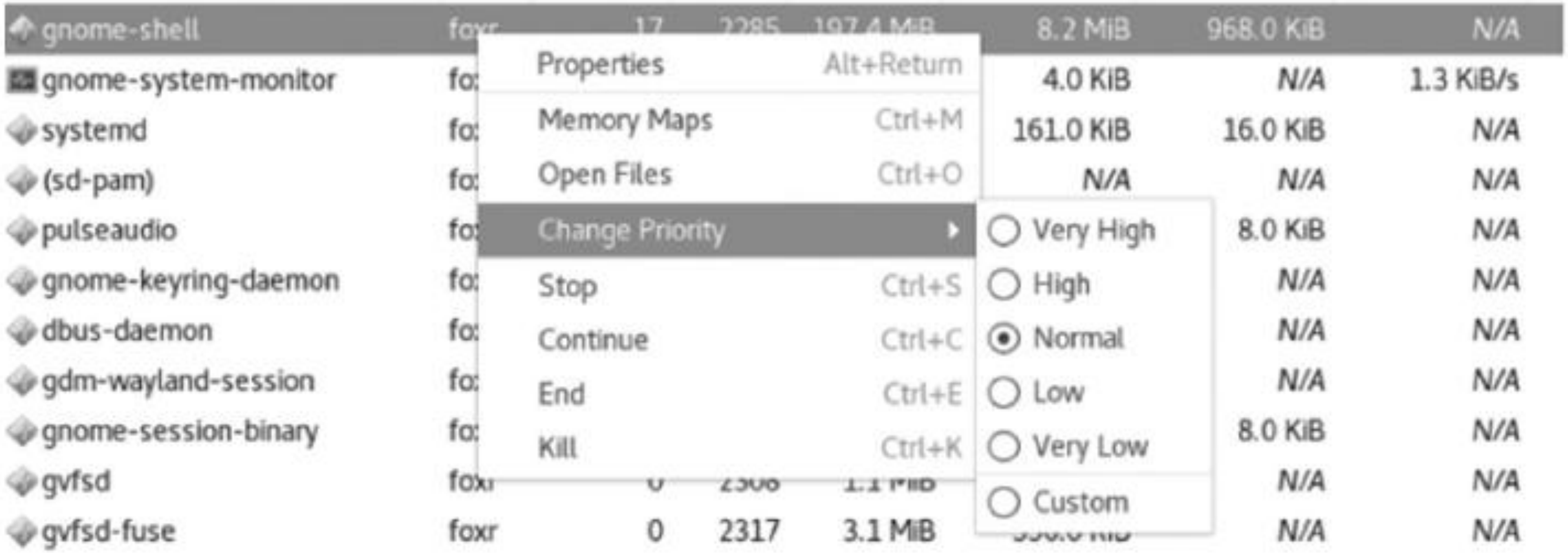
MANAGING PROCESS PRIORITY

- Linux runs processes and threads together by switching off between them.
- Because of the speed and power of modern processors, the user is typically unaware of the time elapsing as the processor moves from one process to another and back.
- processor moves through the processes in the ready queue and back to the first in under a millisecond (thousandths of a second).

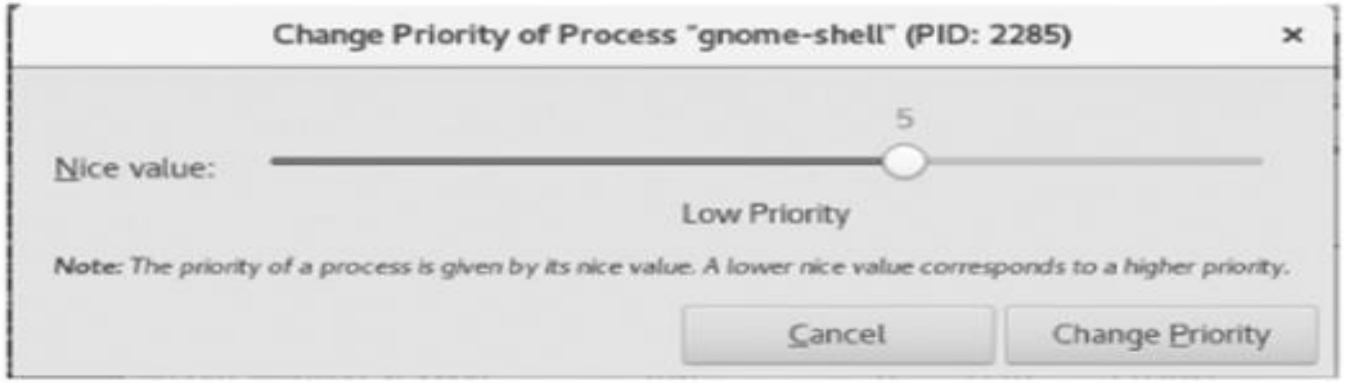
- A process is a standalone entity; it has its own code, data and status information.
- Thread, some-times called a *lightweight* process, shares its code and data (or at least some data) with other threads.
- Threads are, in essence, portions of a process.
- Linux executes processes and threads using multitasking and multithreading.
- Linux command batch, for instance, forces a process to run in batch mode.
- A batch process is one that does not interact with the user. Therefore, any batch process must be provided its input at the time the process is launched.

- In Linux, a process’ priority is established by setting its *niceness* value. Niceness refers to how nice a process is with respect to other processes.
- With a higher niceness value, a process will offer some of its CPU time to other processes.
- Higher the niceness value, the less CPU time it will take when it is that process’ turn, and thus higher niceness means lower priority.

https://www.geeksforgeeks.org/priority-of-process-in-linux-nice-value/



```
gnome-shell            foxr   17   2285   197.4 MiB   8.2 MiB     968.0 KiB   N/A
gnome-system-monitor   fo                             4.0 KiB     N/A         1.3 KiB/s
systemd                fo                             161.0 KiB   16.0 KiB    N/A
(sd-pam)               fo                             N/A         N/A         N/A
pulseaudio             fo                                         8.0 KiB     N/A
gnome-keyring-daemon   fo                                         N/A         N/A
dbus-daemon            fo                                         N/A         N/A
gdm-wayland-session    fo                                         N/A         N/A
gnome-session-binary   fo                                         8.0 KiB     N/A
gvfsd                  foxr   0    2308   1.1 MiB                 N/A         N/A
gvfsd-fuse             foxr   0    2317   3.1 MiB                 N/A         N/A
```



Change Priority of Process "gnome-shell" (PID: 2285)

Nice value: 5

Low Priority

Note: *The priority of a process is given by its nice value. A lower nice value corresponds to a higher priority.*

Regular Expressions

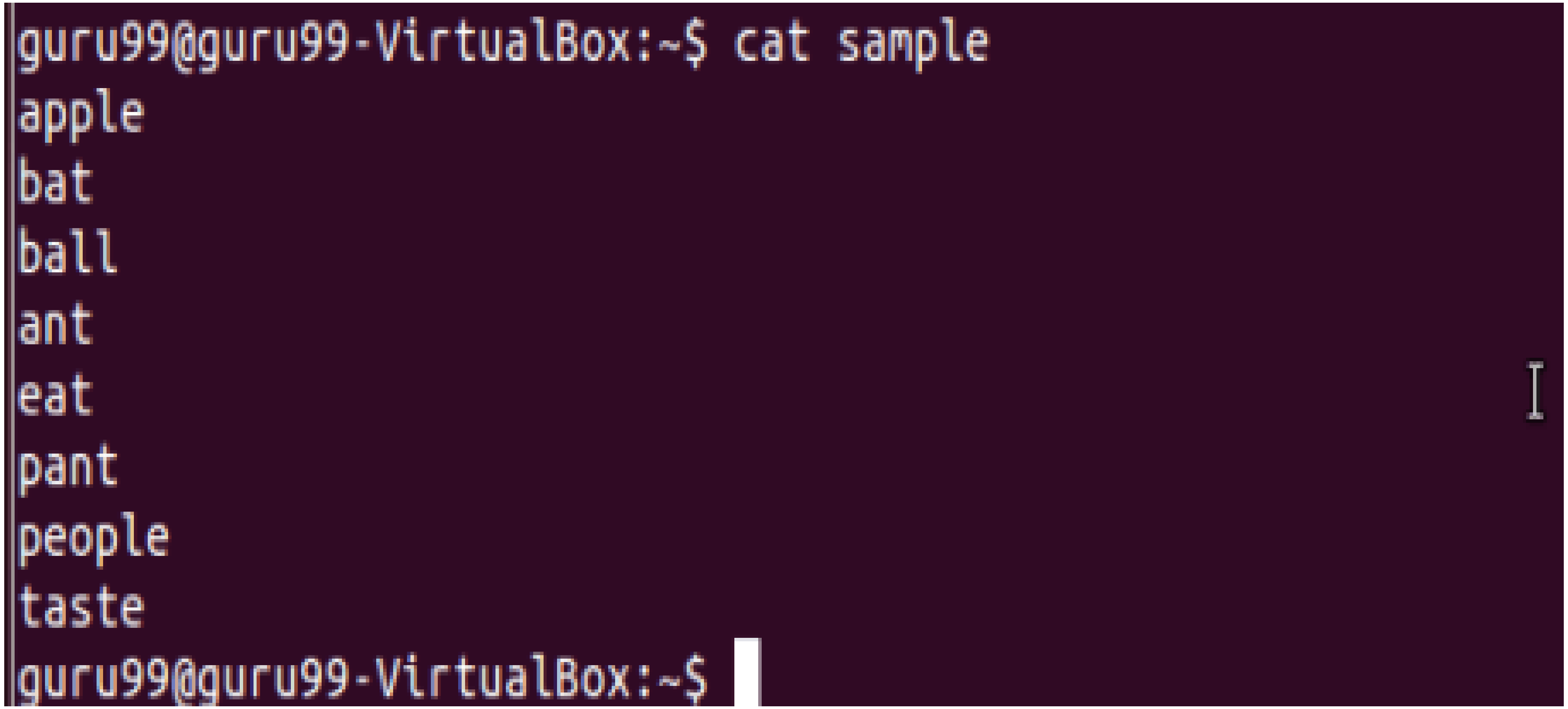
- **Linux Regular Expressions** are special characters which help search data and matching complex patterns.
- Regular expressions are shortened as ‘regexp’ or ‘regex’.
- They are used in many Linux programs like grep, bash, rename, sed, etc.

Basic Regular expressions

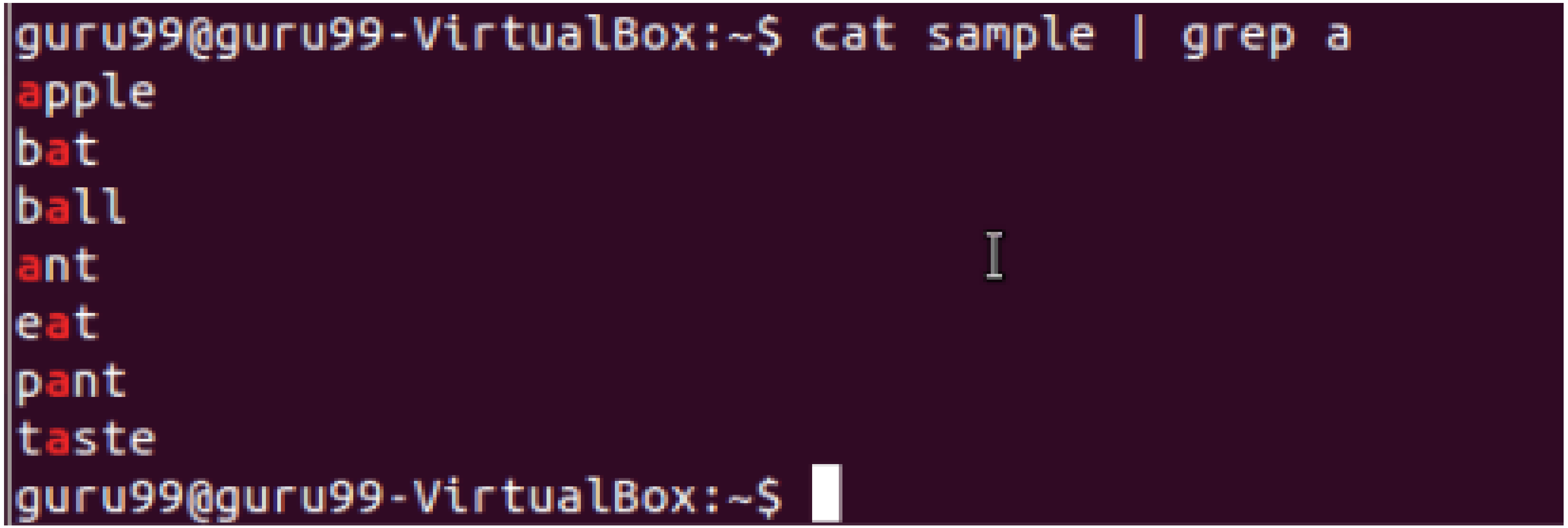
Symbol	Descriptions
.	replaces any character
^	matches start of string
$	matches end of string
*	matches up zero or more times the preceding character
\	Represent special characters
()	Groups regular expressions
?	Matches up exactly one character

Syntax:

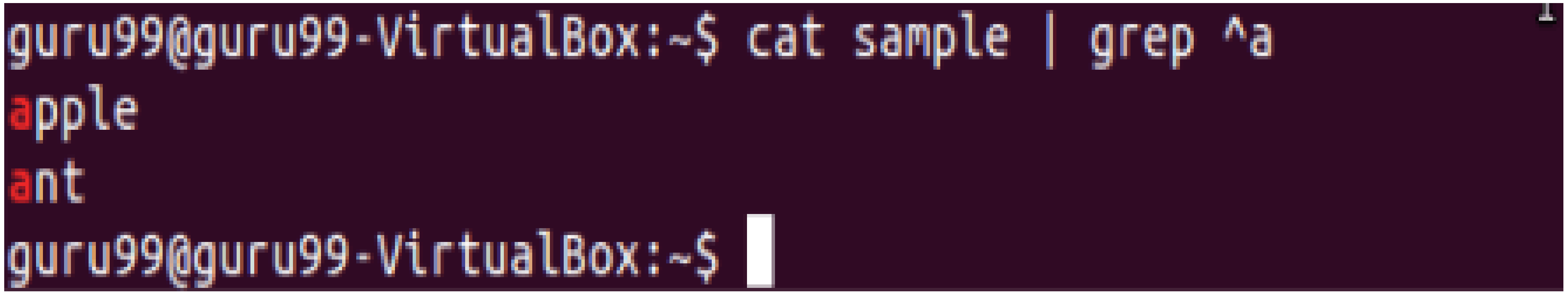
- cat sample | grep string



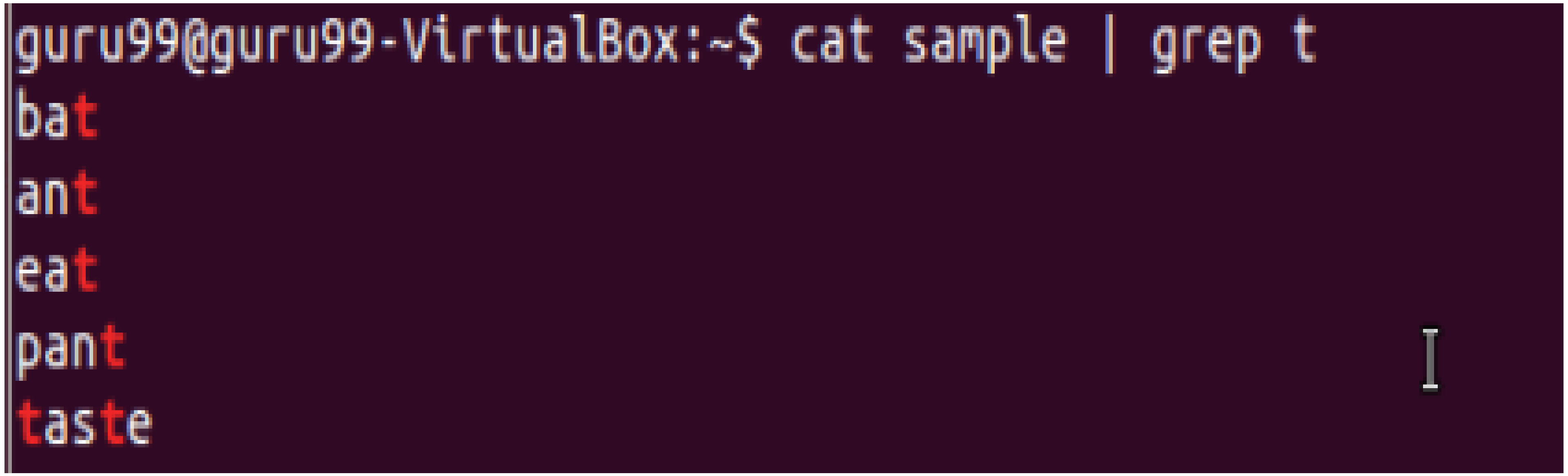
```
guru99@guru99-VirtualBox:~$ cat sample
apple
bat
ball
ant
eat
pant
people
taste
guru99@guru99-VirtualBox:~$
```



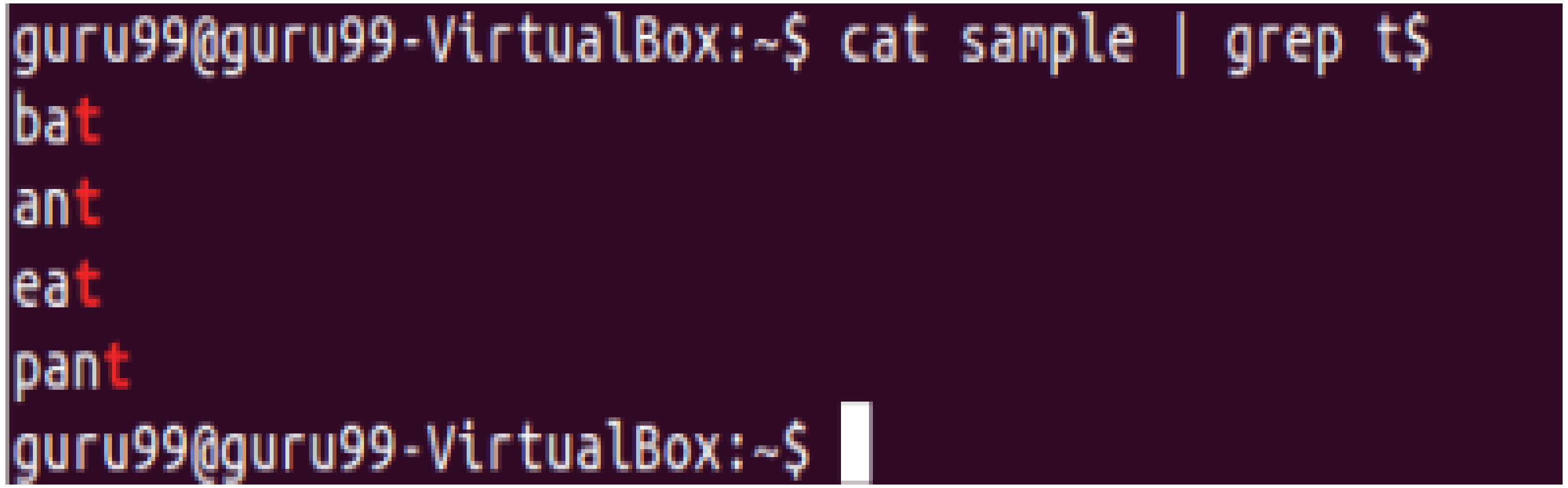
```
guru99@guru99-VirtualBox:~$ cat sample | grep a
apple
bat
ball
ant
eat
pant
taste
guru99@guru99-VirtualBox:~$
```



```
guru99@guru99-VirtualBox:~$ cat sample | grep ^a
apple
ant
guru99@guru99-VirtualBox:~$
```



```
guru99@guru99-VirtualBox:~$ cat sample | grep t
bat
ant
eat
pant
taste
```



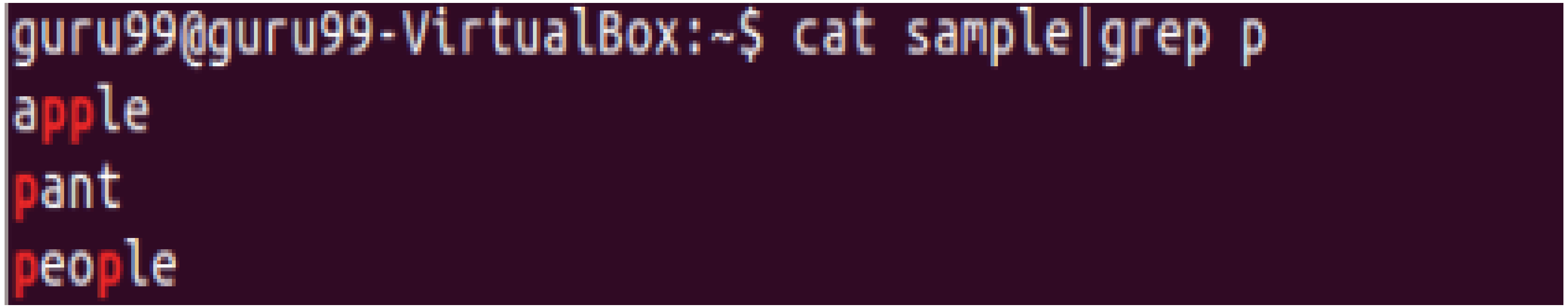
```
guru99@guru99-VirtualBox:~$ cat sample | grep t$
bat
ant
eat
pant
guru99@guru99-VirtualBox:~$
```

Interval Regular expressions

- These expressions tell us about the number of occurrences of a character in a string.

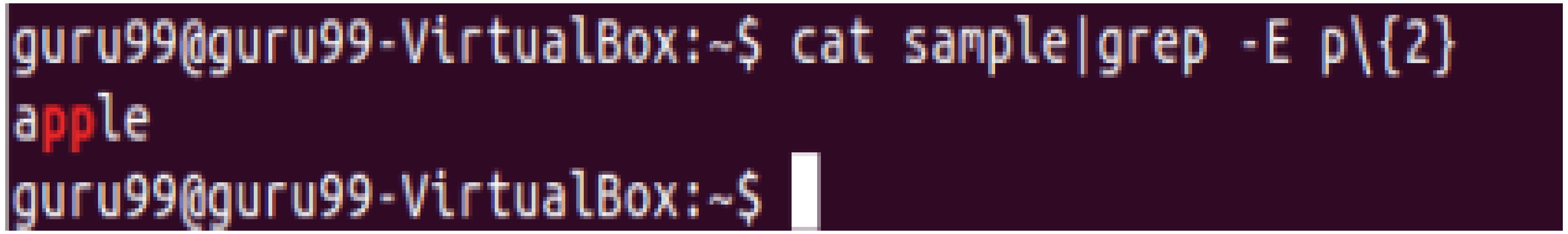
Expression	Description
{n}	Matches the preceding character appearing ‘n’ times exactly
{n,m}	Matches the preceding character appearing ‘n’ times but not more than m
{n, }	Matches the preceding character only when it appears ‘n’ times or more

Filter out all lines that contain character ‘p’



```
guru99@guru99-VirtualBox:~$ cat sample|grep p
apple
pant
people
```

Check that the character ‘p’ appears exactly 2 times in a string one after the other.



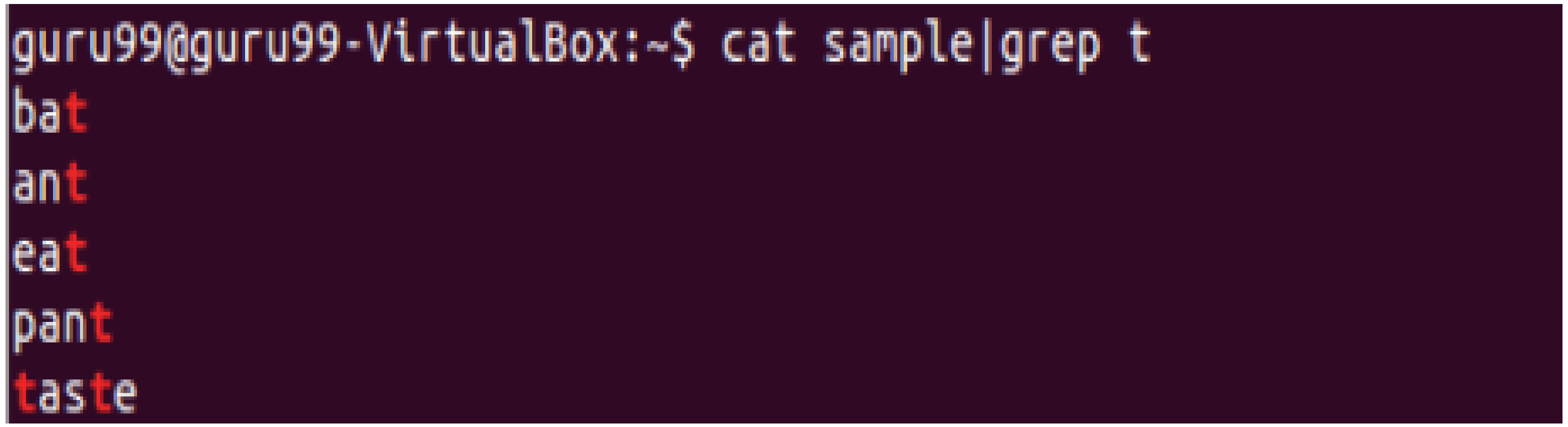
```
guru99@guru99-VirtualBox:~$ cat sample|grep -E p\{2}
apple
guru99@guru99-VirtualBox:~$ 
```

need to add -E with these regular expressions.

Extended regular expressions

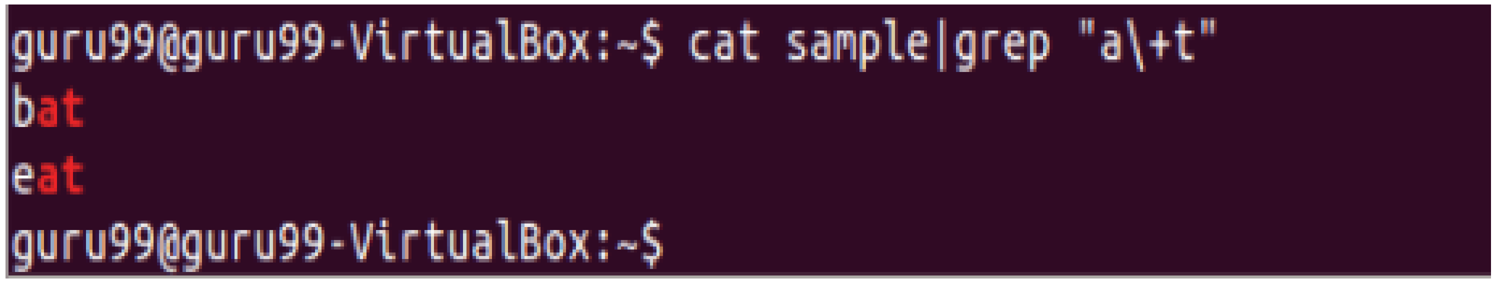
- regular expressions contain combinations of more than one expression.

Expression	Description
\+	Matches one or more occurrence of the previous character
\?	Matches zero or one occurrence of the previous character



```
guru99@guru99-VirtualBox:~$ cat sample|grep t
bat
ant
eat
pant
taste
```

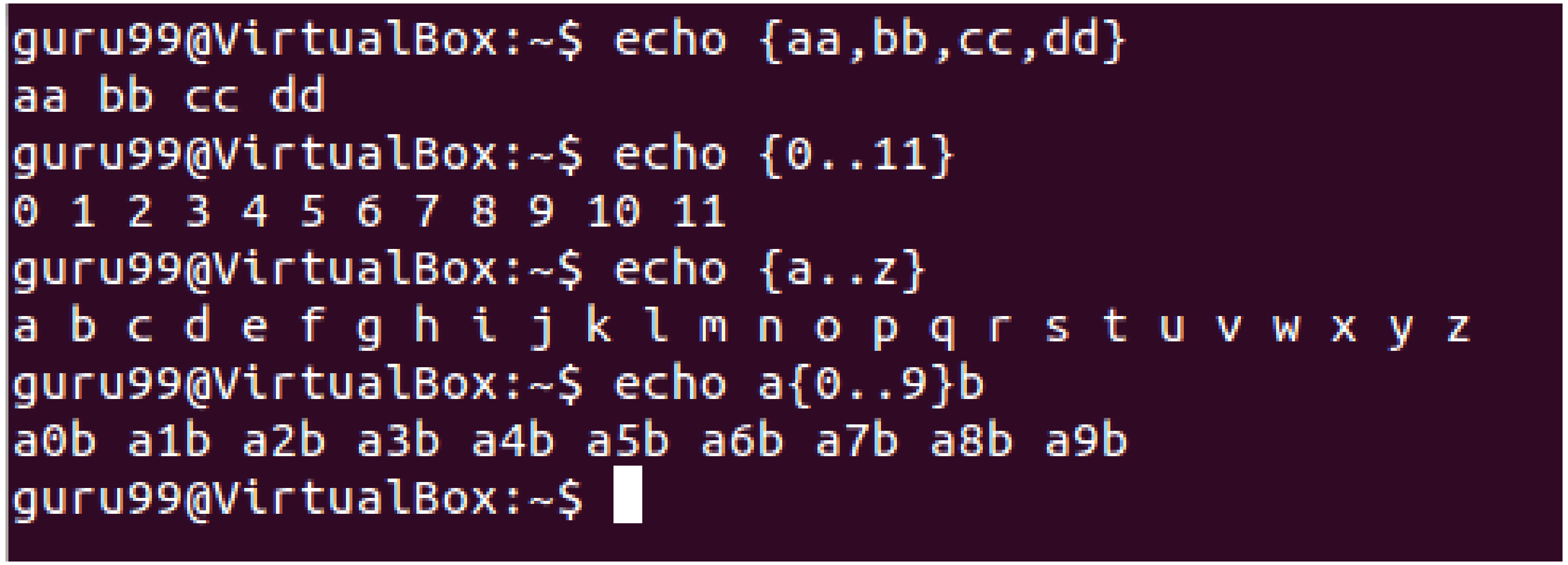
Filter out lines where character ‘a’ precedes character ‘t’



```
guru99@guru99-VirtualBox:~$ cat sample|grep "a\+t"
bat
eat
guru99@guru99-VirtualBox:~$
```

Brace expansion

- Brace expansion is either a sequence or a comma separated list of items inside curly braces “{}”.
- Starting and ending items in a sequence are separated by two periods “..”.



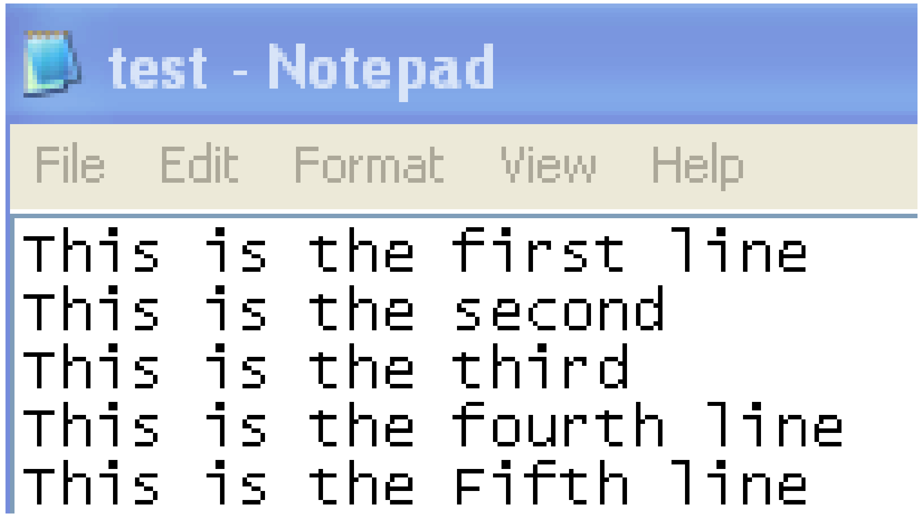
```
guru99@VirtualBox:~$ echo {aa,bb,cc,dd}
aa bb cc dd
guru99@VirtualBox:~$ echo {0..11}
0 1 2 3 4 5 6 7 8 9 10 11
guru99@VirtualBox:~$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z
guru99@VirtualBox:~$ echo a{0..9}b
a0b a1b a2b a3b a4b a5b a6b a7b a8b a9b
guru99@VirtualBox:~$
```

Matching Control

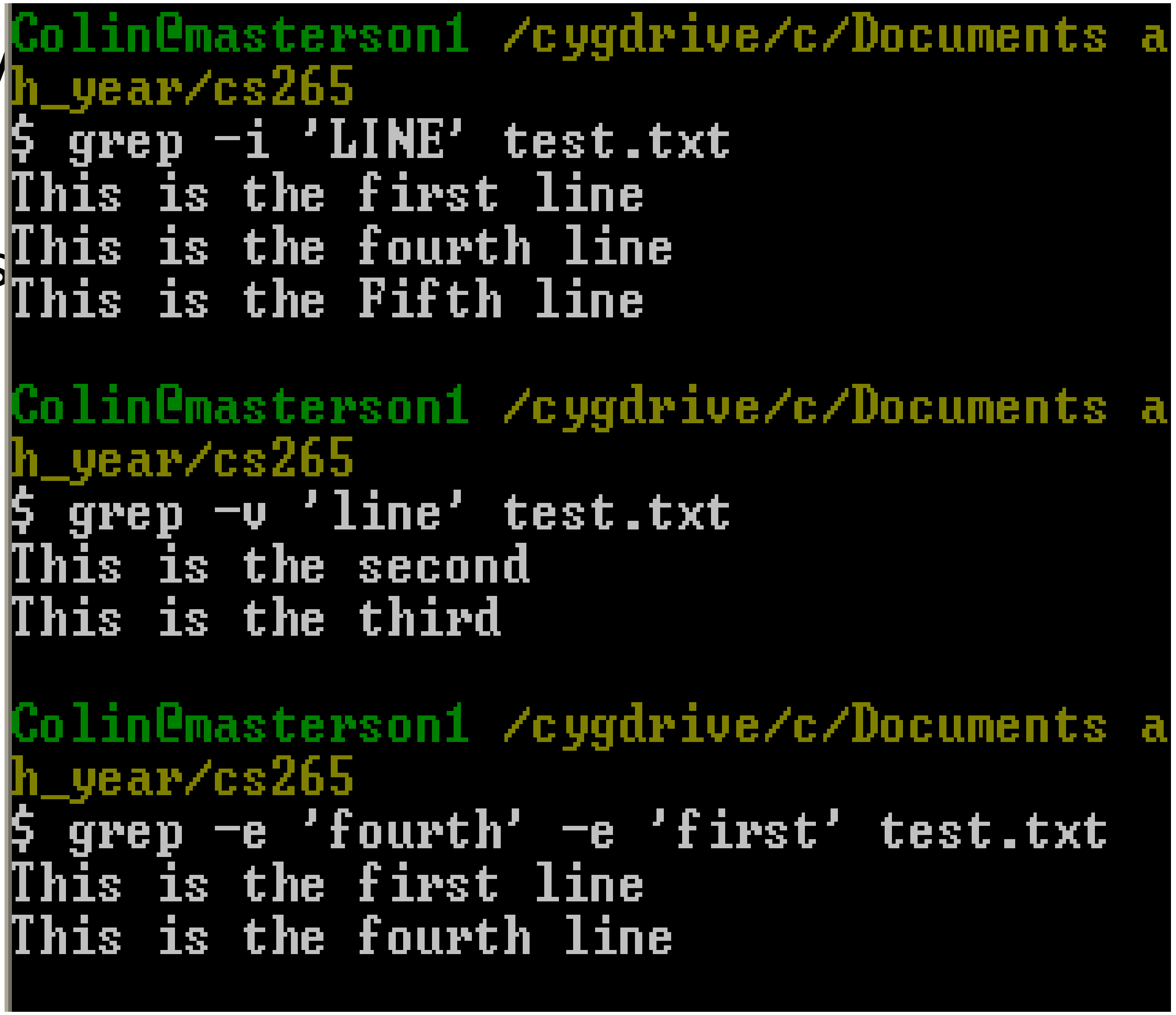
- ‘-i’
  - Ignores case.
- ‘-v’
  - Inverts the matching. When used, grep will print out lines that do not match the pattern
- ‘-e pattern’
  - Pattern is the pattern. This can be used to specify multiple patterns, or if the pattern starts with a ‘-’. A line only has to contain one of the patterns to be matched.

Examples using ‘-i’, ’-v

- The file below was used for thes



```
This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line
```



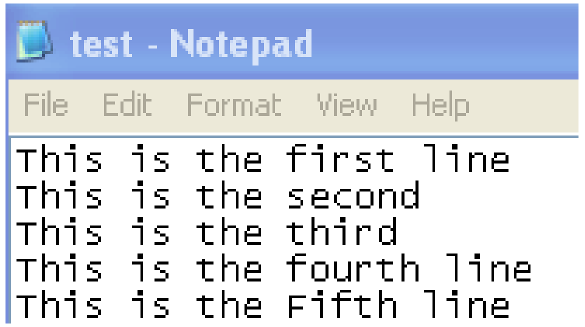
```
Colin@masterson1 /cygdrive/c/Documents a
h_year/cs265
$ grep -i 'LINE' test.txt
This is the first line
This is the fourth line
This is the Fifth line

Colin@masterson1 /cygdrive/c/Documents a
h_year/cs265
$ grep -v 'line' test.txt
This is the second
This is the third

Colin@masterson1 /cygdrive/c/Documents a
h_year/cs265
$ grep -e 'fourth' -e 'first' test.txt
This is the first line
This is the fourth line
```

General Output Control

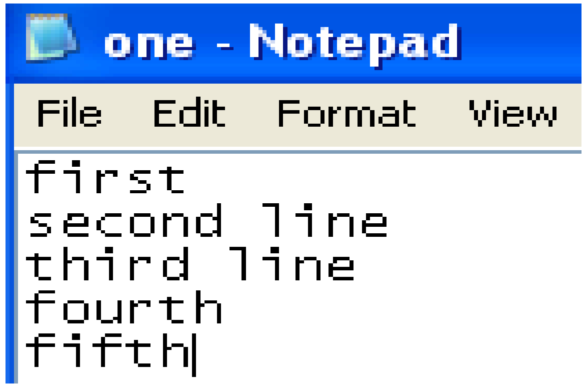
- ‘-c’
  - Suppress normal output and instead print out a count of matching lines for each input file
- ‘-l’
  - Suppress normal output and print the name of each file that contains at least one match
- ‘-L’
  - Suppress normal output. Print the name of each file that does not contain any matches
- Note: both the ‘-l’ and ‘-L’ options will stop searching a file once a match is found

Examples using ‘-c’, ’-l’, and ‘-L’

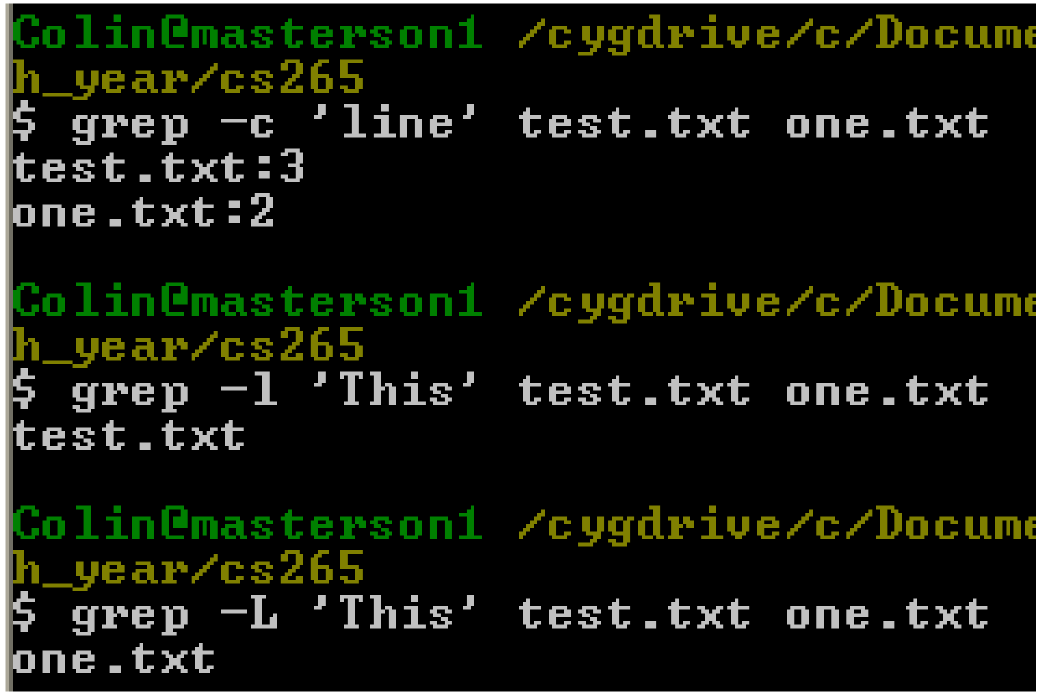
- Two files were used and specified below



```
This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line
```



```
first
second line
third line
fourth
fifth
```



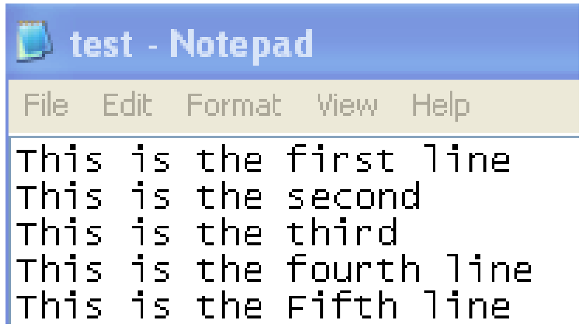
```
Colin@masterson1 /cygdrive/c/Docume
h_year/cs265
$ grep -c 'line' test.txt one.txt
test.txt:3
one.txt:2

Colin@masterson1 /cygdrive/c/Docume
h_year/cs265
$ grep -l 'This' test.txt one.txt
test.txt

Colin@masterson1 /cygdrive/c/Docume
h_year/cs265
$ grep -L 'This' test.txt one.txt
one.txt
```

Output Line Prefix Control

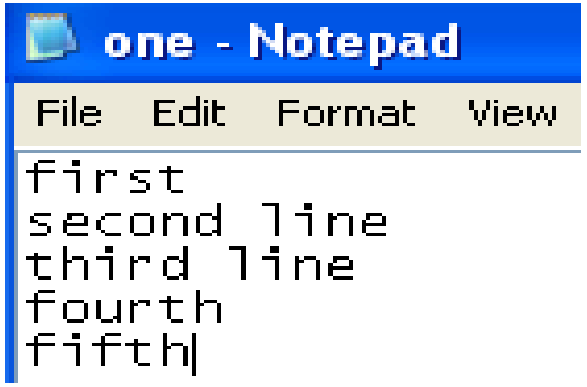
- ‘-n’
  - Prefixes each line of output with the line number from the input file the match was found on
- ‘-H’
  - Prefix each line of output with the input file name that the match was found in
- ‘-T’
  - Makes sure that the actual line content (or whatever content comes after the ‘-T’) lands on a tab stop

Examples using ‘-H’, ’-n’, and ‘-T’

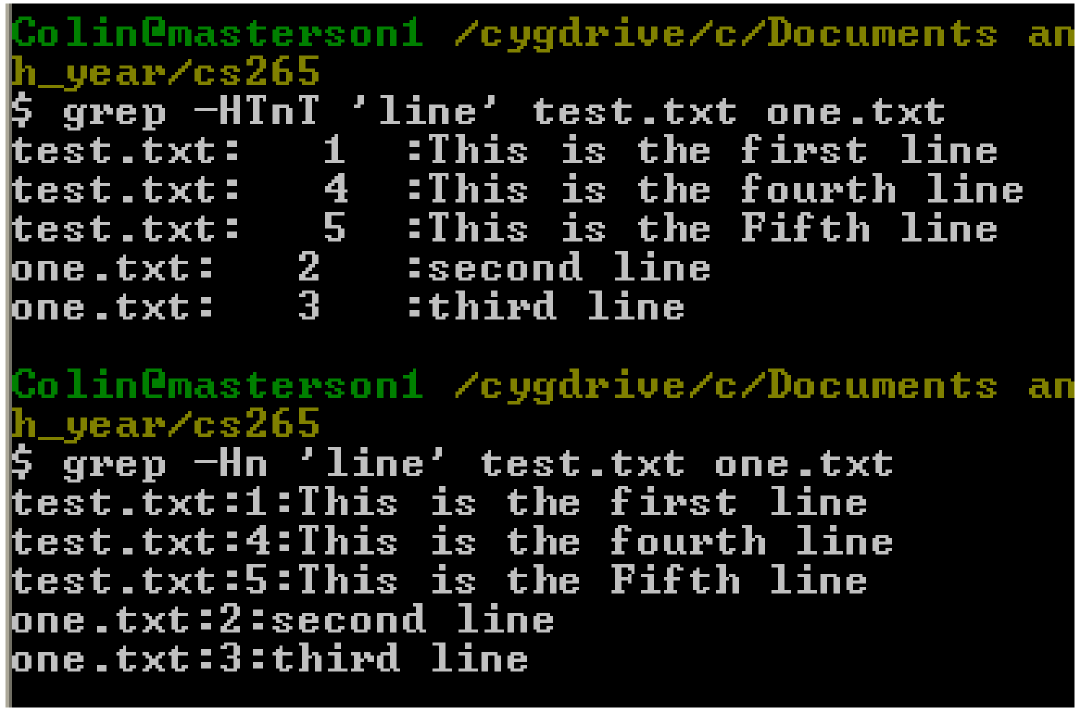
- Two files were used and specified below



```
This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line
```



```
first
second line
third line
fourth
fifth
```



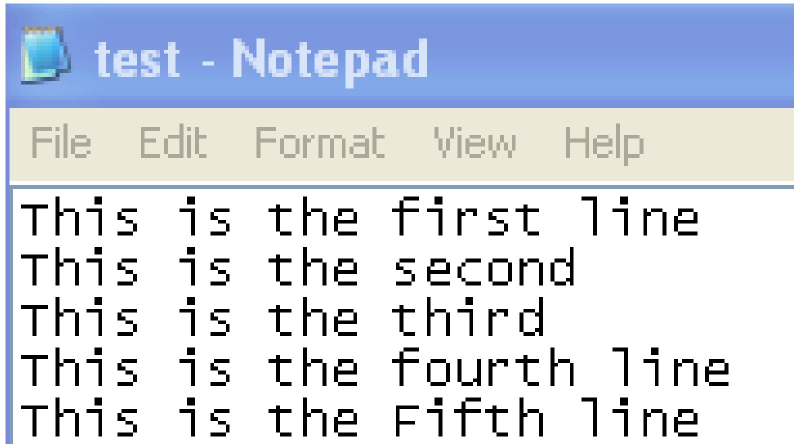
```
Colin@masterson1 /cygdrive/c/Documents an
h_year/cs265
$ grep -HTnT 'line' test.txt one.txt
test.txt:    1   :This is the first line
test.txt:    4   :This is the fourth line
test.txt:    5   :This is the Fifth line
one.txt:    2    :second line
one.txt:    3    :third line

Colin@masterson1 /cygdrive/c/Documents an
h_year/cs265
$ grep -Hn 'line' test.txt one.txt
test.txt:1:This is the first line
test.txt:4:This is the fourth line
test.txt:5:This is the Fifth line
one.txt:2:second line
one.txt:3:third line
```

Context Line Control

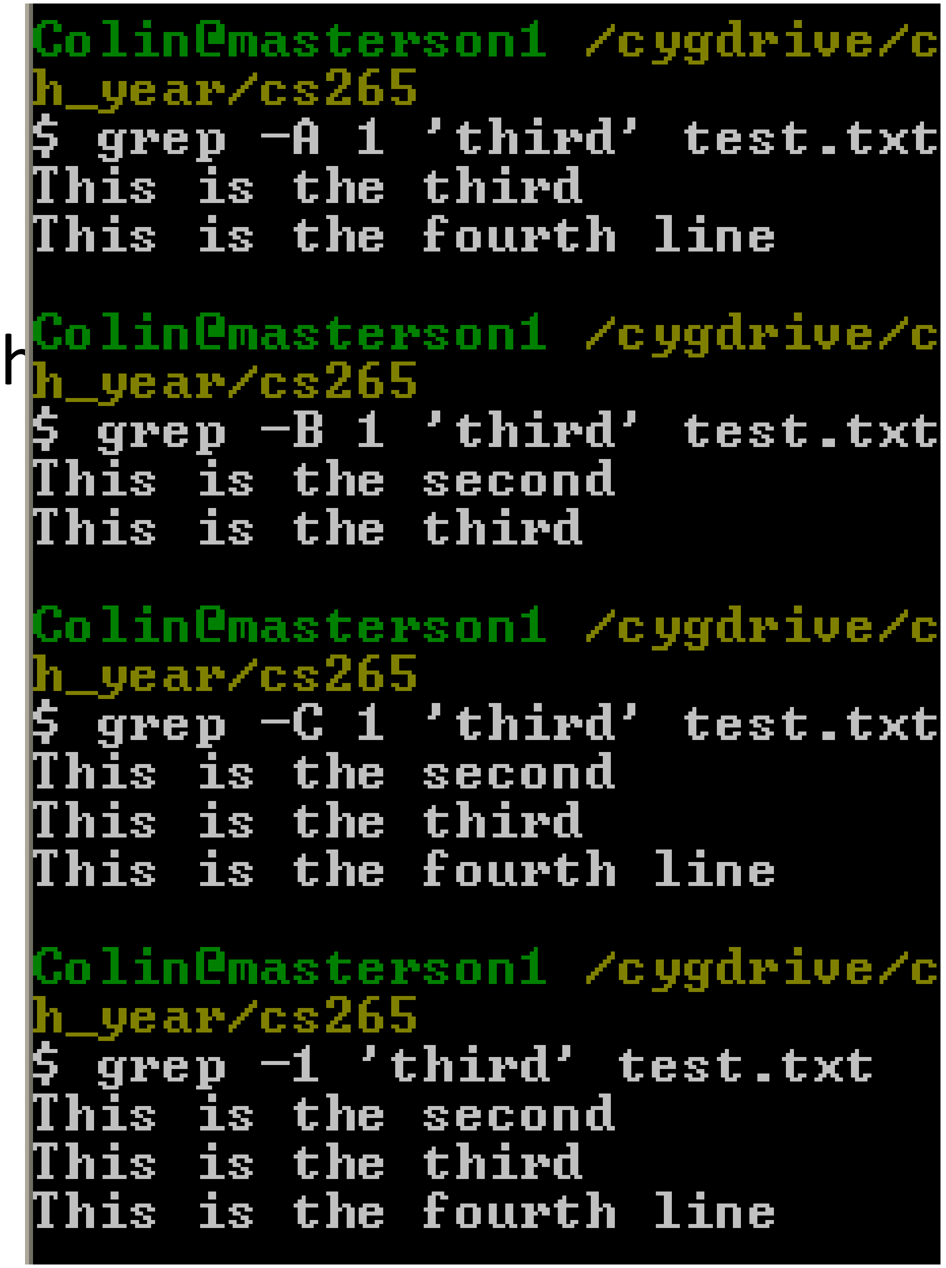
- ‘-A *num*’
  - Print *num* lines of trailing context after matching lines
- ‘-B *num*’
  - Print *num* lines of leading context before matching lines
- ‘-C *num*’ or ‘-*num*’
  - Print *num* lines of leading and trailing output context

Examples using ‘-A’,

- The file below was used for th



```
This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line
```

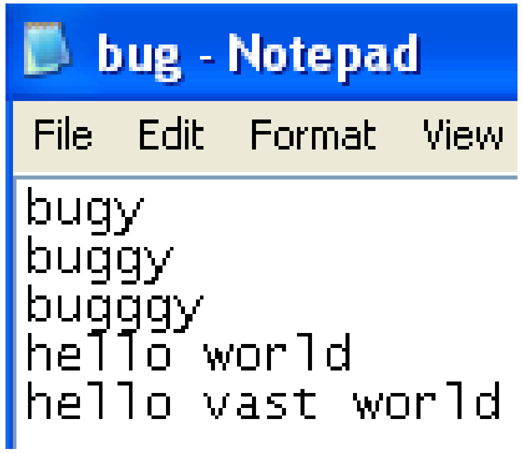


```
Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -A 1 'third' test.txt
This is the third
This is the fourth line

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -B 1 'third' test.txt
This is the second
This is the third

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -C 1 'third' test.txt
This is the second
This is the third
This is the fourth line

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -1 'third' test.txt
This is the second
This is the third
This is the fourth line
```

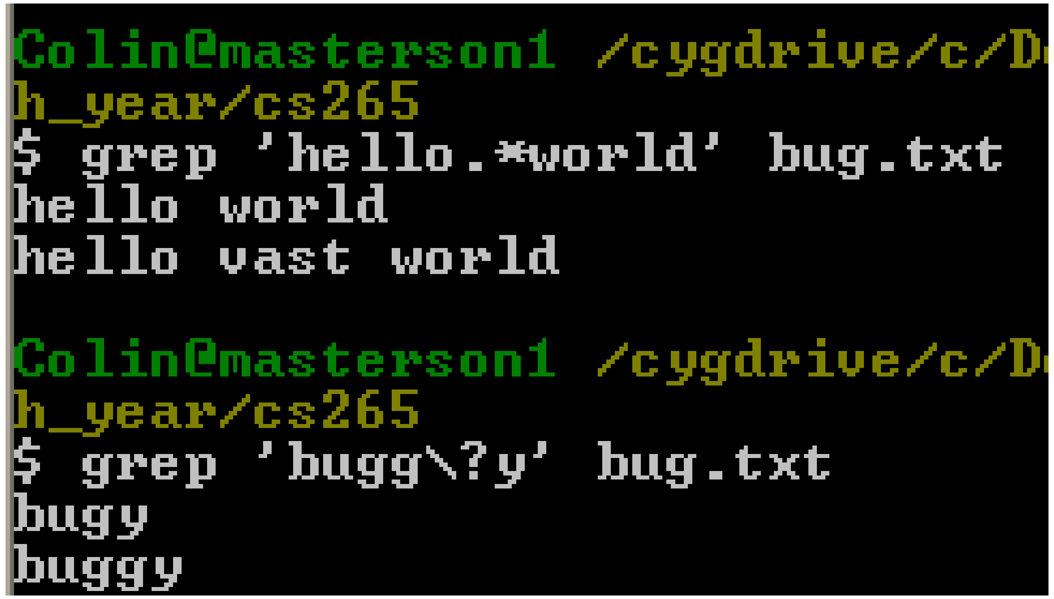
Special Characters

- ‘.’ The period ‘.’ matches any single character.
- ‘?’ The preceding item is optional and will be matched at most once.
- ‘*’ The preceding item will be matched zero or more times.
- ‘+’ The preceding item will be matched one or more times.
- ‘{n}’ The preceding item is matched exactly n times.
- ‘{n,}’ The preceding item is matched n or more times.
- ‘{,m}’ The preceding item is matched at most m times.
- ‘{n,m}’ The preceding item is matched at least n times, but not more than m times.

Examples using ‘.’, ‘*’, and ‘?’



```
bugy
buggy
bugggy
hello world
hello vast world
```

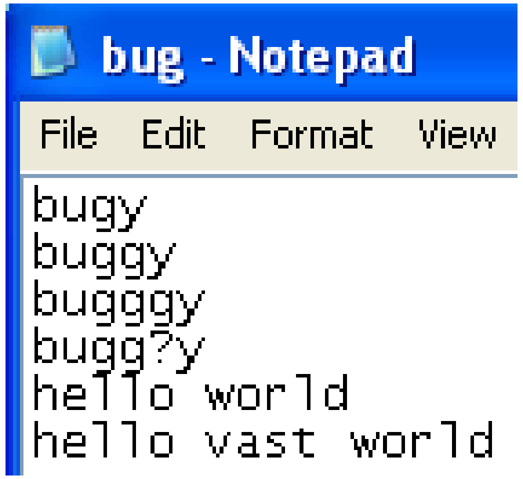


```
Colin@masterson1 /cygdrive/c/D
h_year/cs265
$ grep 'hello.*world' bug.txt
hello world
hello vast world

Colin@masterson1 /cygdrive/c/D
h_year/cs265
$ grep 'bugg\?y' bug.txt
bugy
buggy
```

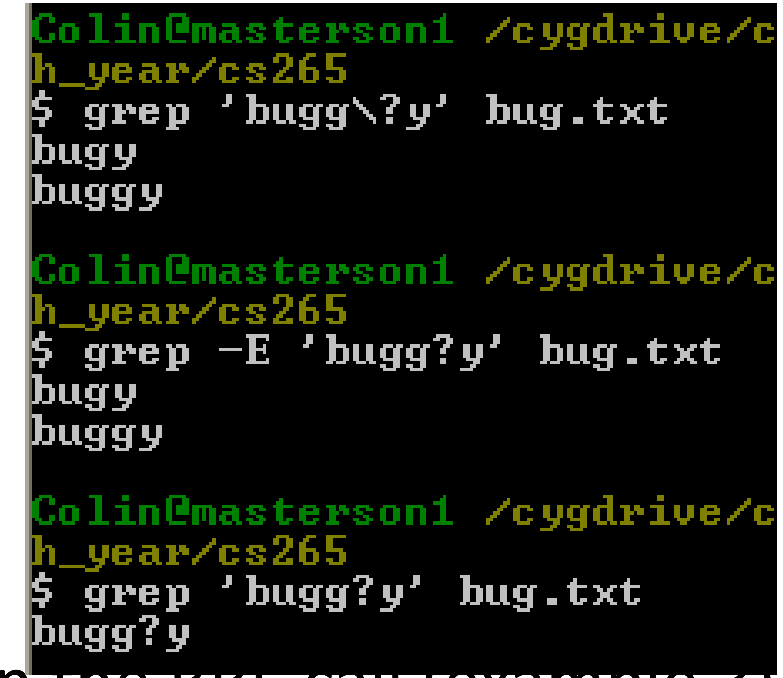
Basic vs Extended Regular Expressions

- ‘-G’
  - Interpret pattern as basic regular expression (BRE). This is the default.
- ‘-E’
  - Interpret pattern as extended regular expression (ERE)
- When using basic regular expression some special characters (like ‘?’ in the previous example) loose their special meaning and must have a ‘\’, the escape character, before them
- When using ERE, the escape character is unnecessary

BRE and ERE Difference



```
bugy
buggy
bugggy
bugg?y
hello world
hello vast world
```



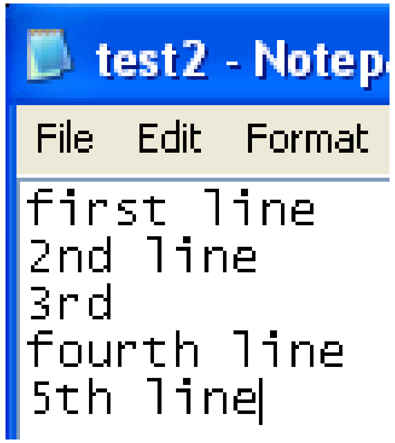
```
Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep 'bugg\?y' bug.txt
bugy
buggy

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -E 'bugg?y' bug.txt
bugy
buggy

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep 'bugg?y' bug.txt
bugg?y
```

- Note that without the '\' in the BRE call (example 3), the '?' is seen as a normal character

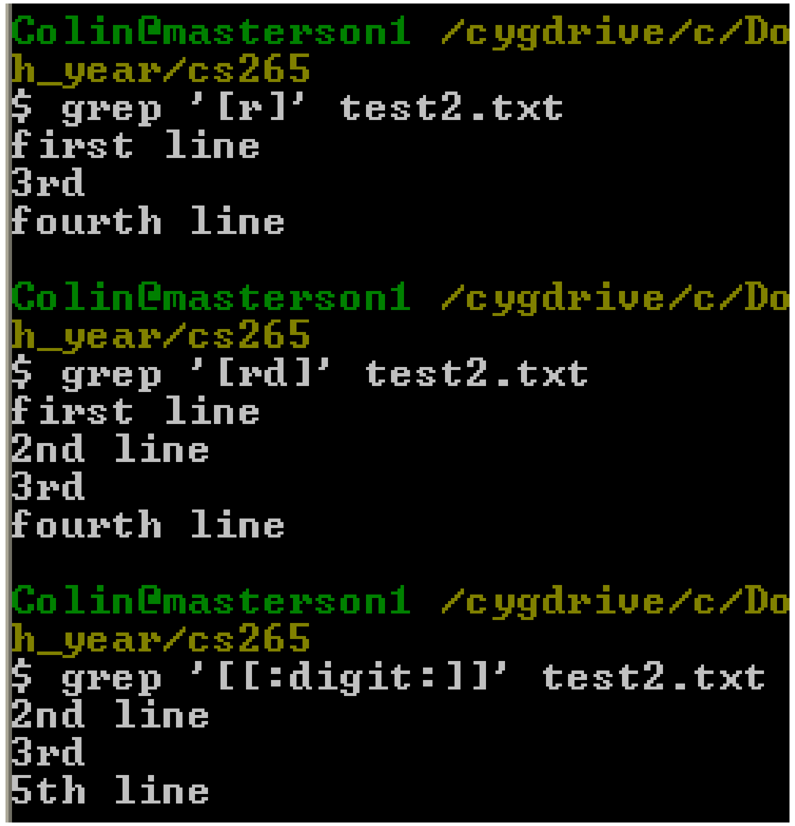
Bracket Expressions

- A bracket expression is a list of characters enclosed by ‘[‘ and ‘]’. It matches any single character in the list
- However, if the first character in the list is ‘^’, it matches any character not in the list
- A range can be done by using ‘-’ in a bracket expression
  - [0-5] is the same as [012345]
- Some ranges are pre-defined in character classes
  - [:digit:] is the same as 0123456789
  - When using grep, the class name (including brackets) must be contained within another set of brackets

Bracket Expression Example



```
first line
2nd line
3rd
fourth line
5th line
```



```
Colin@masterson1 /cygdrive/c/Do
h_year/cs265
$ grep '[r]' test2.txt
first line
3rd
fourth line

Colin@masterson1 /cygdrive/c/Do
h_year/cs265
$ grep '[rd]' test2.txt
first line
2nd line
3rd
fourth line

Colin@masterson1 /cygdrive/c/Do
h_year/cs265
$ grep '[[:digit:]]' test2.txt
2nd line
3rd
5th line
```

àwk

- awk is a programmable, pattern-matching, and processing tool available in UNIX.
- It works equally well with text and numbers.
- It derives its name from the first letter of the last name of its three authors namely Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan.

Simple awk Filtering

- awk is not just a command, but a programming language too.
- awk utility is a pattern scanning and processing language.
- It searches one or more files to see if they contain lines that match specified patterns and then perform associated actions, such as writing the line to the standard output or incrementing a counter each time it finds a match.

AWK Operations:

(a) Scans a file line by line
(b) Splits each input line into fields
(c) Compares input line/fields to pattern
(d) Performs action(s) on matched lines

```
awk 'pattern { action }' filename
```

Common Patterns and Special Variables:

- `BEGIN` - Executes before processing any lines
- `END` - Executes after processing all lines
- `NR` - Current line number
- `NF` - Number of fields in current line
- `$0` - Entire line
- `$1, $2, ...` - First field, second field, etc.
- `FS` - Field separator (default is whitespace)
- `OFS` - Output field separator

- **NR:** NR command keeps a current count of the number of input records.
- **NF:** NF command keeps a count of the number of fields within the current input record.
- **FS:** FS command contains the field separator character which is used to divide fields on the input line→ default is "white space", meaning space and tab characters.
- **RS:** RS command stores the current record separator character.
- **OFS:** OFS command stores the output field separator, which separates the fields when Awk prints them.
- **ORS:** ORS command stores the output record separator, which separates the output lines when Awk prints them.

$cat > employee.txt

```
ajay manager account 45000
sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000
```

Print the lines which match the given pattern.

```
awk '{print}' employee.txt
```

```
ajay manager account 45000
sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000
```

awk '/manager/ {print}' employee.txt

```
ajay manager account 45000
varun manager sales 50000
amit manager account 47000
```

```
# Print entire file
awk '{print}' file.txt
# or
awk '{print $0}' file.txt

# Print first column
awk '{print $1}' file.txt

# Print first and second columns
awk '{print $1, $2}' file.txt

# Print last column
awk '{print $NF}' file.txt

# Print second-to-last column
awk '{print $(NF-1)}' file.txt
```

```
# Print line numbers
awk '{print NR, $0}' file.txt

# Print line numbers for non-empty lines
awk 'NF {print NR, $0}' file.txt

# Print line numbers and first column
awk '{print NR ":", $1}' file.txt
```

```
# Print lines containing "pattern"
awk '/pattern/' file.txt

# Print lines NOT containing "pattern"
awk '!/pattern/' file.txt

# Print lines longer than 80 characters
awk 'length($0) > 80' file.txt

# Print lines with more than 5 fields
awk 'NF > 5' file.txt
```

```
# Print number of fields in each line
awk '{print NF}' file.txt

# Print lines with field count
awk '{print NF "fields:", $0}' file.txt

# Print first and last fields
awk '{print $1, $NF}' file.txt

# Swap first and second fields
awk '{print $2, $1}' file.txt
```

```
# Sum first column
awk '{sum += $1} END {print sum}' file.txt

# Average of first column
awk '{sum += $1} END {print sum/NR}' file.txt

# Count lines
awk 'END {print NR}' file.txt

# Count non-empty lines
awk 'NF {count++} END {print count}' file.txt
```

```
# Use comma as separator
awk -F',' '{print $1}' file.csv

# Use colon as separator
awk -F':' '{print $1}' /etc/passwd

# Multiple character separator
awk -F'[ ,]' '{print $1}' file.txt  # Space or comma

# Change output separator
awk 'BEGIN {OFS=","} {print $1, $2}' file.txt
```

```
# Print first 10 lines
awk 'NR <= 10' file.txt

# Print lines 5 through 10
awk 'NR >= 5 && NR <= 10' file.txt

# Print odd numbered lines
awk 'NR % 2 == 1' file.txt

# Print even numbered lines
awk 'NR % 2 == 0' file.txt
```

```
# Format numbers with spacing
awk '{printf "%5d %s\n", NR, $0}' file.txt

# Add zeros before numbers
awk '{printf "%03d %s\n", NR, $0}' file.txt

# Format as table
awk '{printf "%-20s %s\n", $1, $2}' file.txt

# Add dashes between fields
awk '{print $1 "-" $2}' file.txt
```

Print lines where first field > 100

- awk '$1 > 100' file.txt

# Sum values in third column

- awk '{sum += $3} END {print sum}' file.txt

# Use comma as separator

- awk -F',' '{print $1}' file.csv

Add line numbers

```
awk '{print NR ") " $0}' file.txt
```

Filter with multiple conditions

# Print lines where $1 > 100 and $2 contains "test"

```
awk '$1 > 100 && $2 ~ /test/' file.txt
```

# Average of second column

```
awk '{sum += $2} END {print "Average = " sum/NR}' file.txt
```

Formatted Output

# Format as CSV

- awk 'BEGIN {OFS=","} {print $1, $2, $3}' file.txt

Splitting a Line Into Fields

- awk '{print $1,$4}' employee.txt

```
ajay 45000
sunil 25000
varun 50000
amit 47000
tarun 15000
deepak 23000
sunil 13000
satvik 80000
```

Built-In Variables In Awk

- Awk’s built-in variables include the field variables—$1, $2, $3, and so on ($0 is the entire line) — that break a line of text into individual words or pieces called fields.

$ awk '{print NR,$0}' employee.txt

```
1 ajay manager account 45000
2 sunil clerk account 25000
3 varun manager sales 50000
4 amit manager account 47000
5 tarun peon sales 15000
6 deepak clerk sales 23000
7 sunil peon sales 13000
8 satvik director purchase 80000
```

Use of NF built-in variables (Display Last Field)

```
$ awk '{print $1,$NF}' employee.txt

ajay 45000
sunil 25000
varun 50000
amit 47000
tarun 15000
deepak 23000
sunil 13000
satvik 80000
```

awk 'NR==3, NR==6 {print NR,$0}' employee.txt

```
3 varun manager sales 50000
4 amit manager account 47000
5 tarun peon sales 15000
6 deepak clerk sales 23000
```

SED command

- SED command in UNIX stands for stream editor and it can perform lots of functions on file like searching, find and replace, insertion or deletion.
- SED is a powerful text stream editor.
- Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

$cat > geekfile.txt

```
unix is great os. unix is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
unix is easy to learn.unix is a multiuser os.Learn unix .
unix is a powerful.
```

Replacing or substituting string

```
$sed 's/unix/linux/' geekfile.txt
```

```
linux is great os. unix is opensource.unix is free os.
learn operating system.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

“s” specifies the substitution operation.

“/” are delimiters.

Replacing all the occurrence of the pattern in a line

```
$sed 's/unix/linux/g' geekfile.txt
```

```
linux is great os. linux is opensource. linux is free os.
learn operating system.
linux linux which one you choose.
linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.
```

Replacing from nth occurrence to all occurrences in a line

$sed 's/unix/linux/3g' geekfile.txt

unix is great os. unix is opensource. **linux** is free os.

learn operating system.

unix linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn linux .linux is a powerful.

Parenthesize first character of each word

```
$ echo "Welcome To The Geek Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g‘
```

```
(W)elcome (T)o (T)he (G)eek (S)tuff
```

Replacing string on a specific line number

```
$sed '3 s/unix/linux/' geekfile.txt
```

```
unix is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a
powerful.
```

Duplicating the replaced line with /p flag

- /p print flag prints the replaced line twice on the terminal. If a line does not have the search pattern and is not replaced, then the /p prints that line only once.

```
linux is great os. unix is opensource. unix is free os.
linux is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

Printing only the replaced lines :

```
$sed -n 's/unix/linux/p' geekfile.txt
```

```
linux is great os. unix is opensource. unix is free os.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

Replacing string on a range of lines :

- $sed '1,3 s/unix/linux/' geekfile.txt

```
linux is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

$sed '2,$ s/unix/linux/' geekfile.txt

unix is great os. unix is opensource. unix is free os.

learn operating system.

linux linux which one you choose.

linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful

Deleting lines from a particular file

To Delete a particular line say n in this example

Syntax:

```
$ sed 'nd' filename.txt
```

Example:

```
$ sed '5d' filename.txt
```

- To Delete a last line

```
$ sed '$d' filename.txt
```

- To Delete from nth to last line

```
$ sed '3,6d' filename.txt
```

- To Delete pattern matching line

```
$ sed '/abc/d' filename.txt
```

cat a.txt

life isn't meant to be easy, life is meant to be lived.

Try to learn & understand something new everyday in life.

Respect everyone & most important love everyone.

Don’t hesitate to ask for love & don’t hesitate to show love too.

Life is too short to be shy.

In life, experience will help you differentiating right from wrong.

- Insert one blank line after each line : sed G a.txt
- To insert two blank lines :sed 'G;G' a.txt
- Insert 5 spaces to the left of every lines : sed 's/^/   /' a.txt