Unix Shell Programming
Unit-3

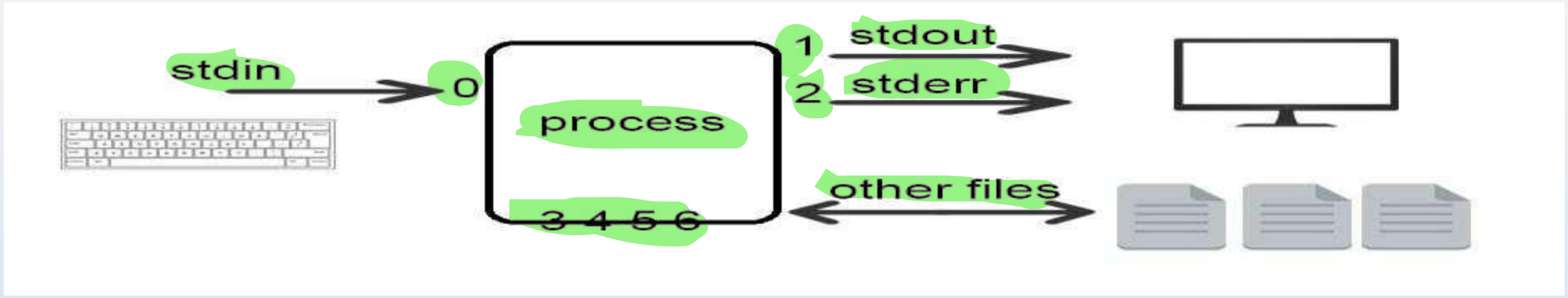
Ability Enhancement Course

AEC310

Standard Streams-Contents

- Redirection
- Pipes
- Tee Command
- Command Execution
- Command-Line Editing
- Quotes
- Command Substitution
- Job Control
- Aliases
- Variables
- Predefined Variables
- Shell/Environment Customization

Standard Streams



Unix defines three standard streams that are used by commands: **Standard input, Standard Output, and Standard Error.**

Each command takes its input from a stream known as a **standard input.**

The command that creates output sends it to a stream known as **standard output.**

If an executing command encounters an error message is sent to **standard error.**

Unix assigns descriptors for each stream as shown in the figure**.**

Note: Not all commands use the standard input/output file, there are exceptions. Ex:lpr

Redirection

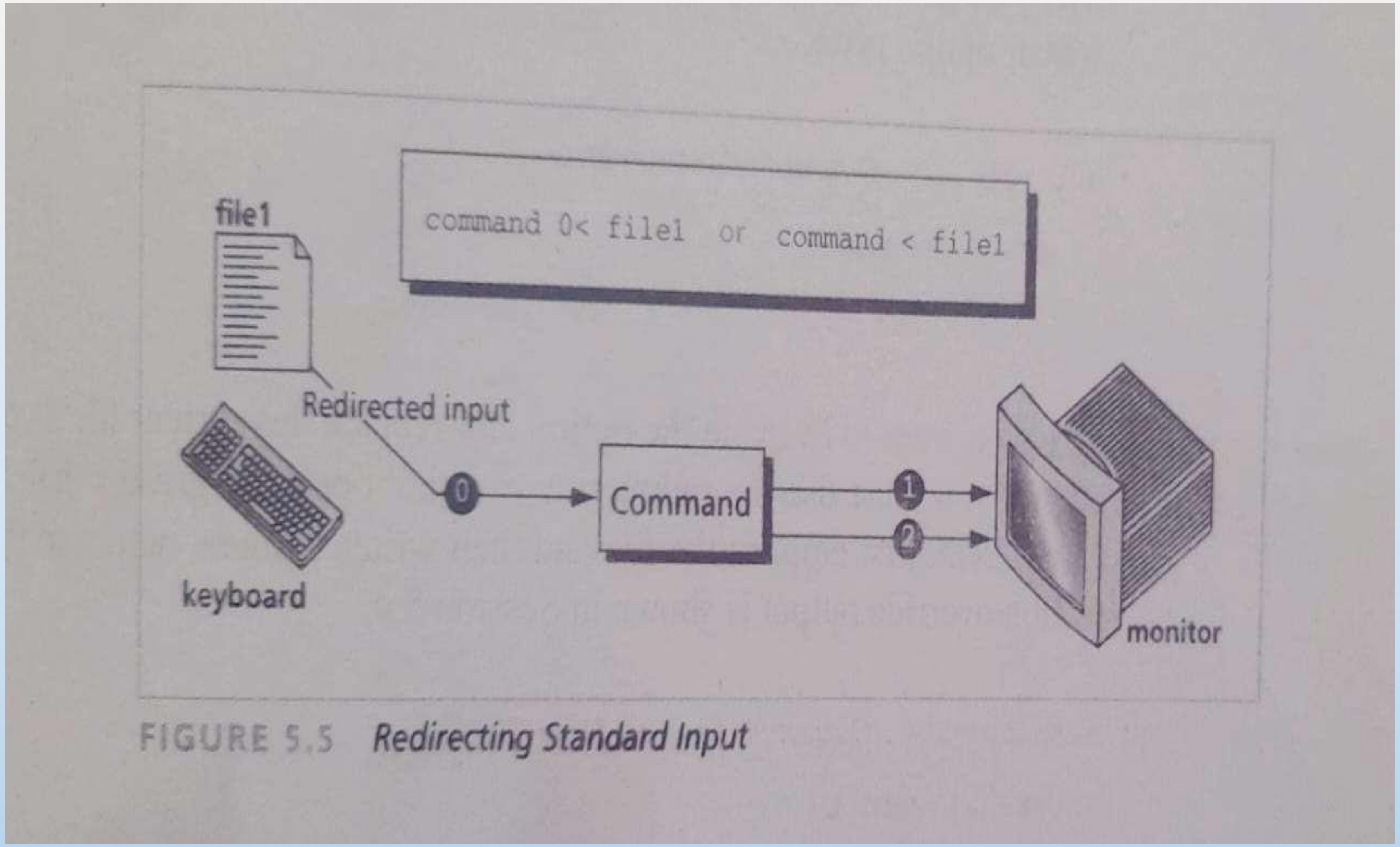
- Input/Output (I/O) redirection in unix refers to the ability of the unix operating system that allows user to change the standard input (stdin) and standard output (stdout) when executing a command on the terminal. By default, the standard input device is your **keyboard** and the standard output device is your **screen.**

- Redirection is the process by which we specify that a file is to be used in place of one of the standard stream files.

Redirecting input

- Command 0< file1

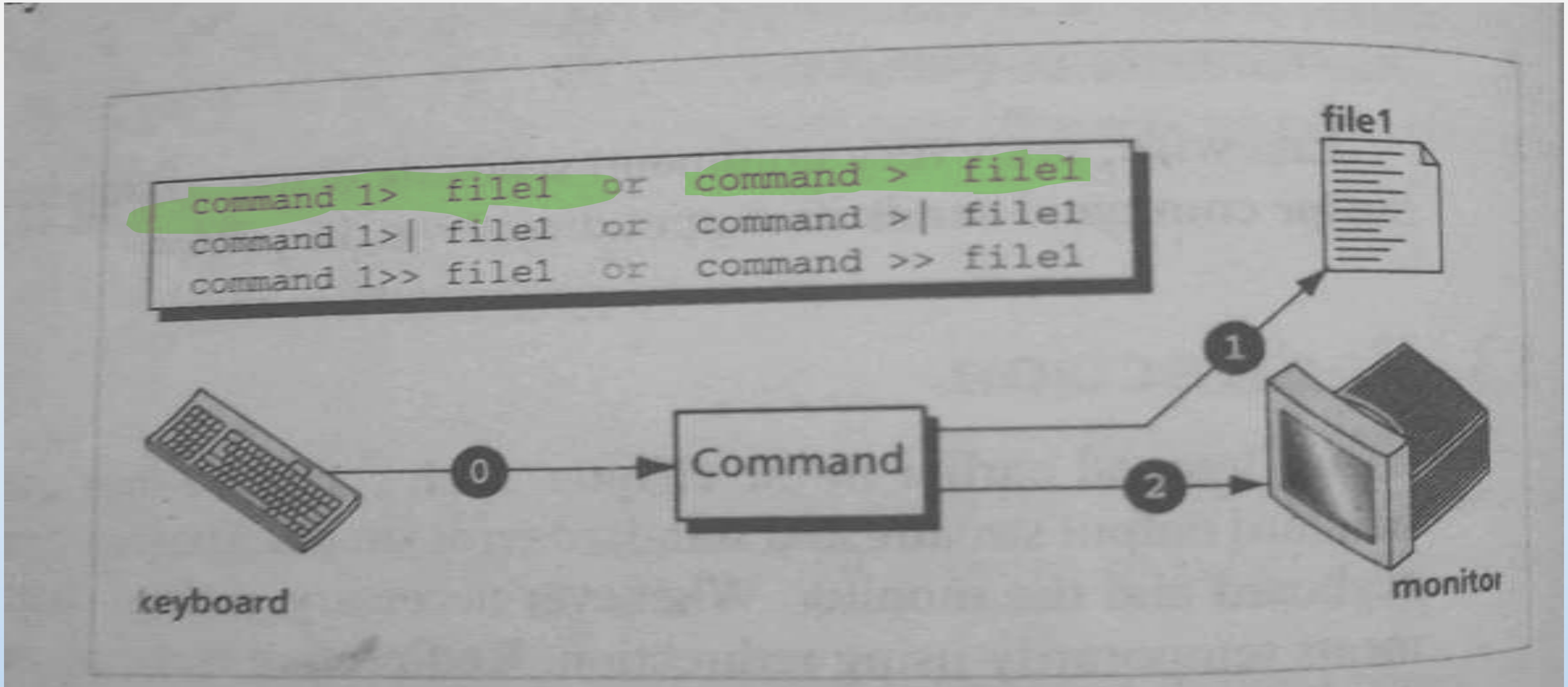
- Command < file1



```
command 0< file1   or   command < file1
```

FIGURE 5.5 *Redirecting Standard Input*

Redirecting Output



```
command 1>  file1  or  command >  file1
command 1>| file1  or  command >| file1
command 1>> file1  or  command >> file1
```

FIGURE 5.6 *Redirecting Standard Output*

Redirecting Output

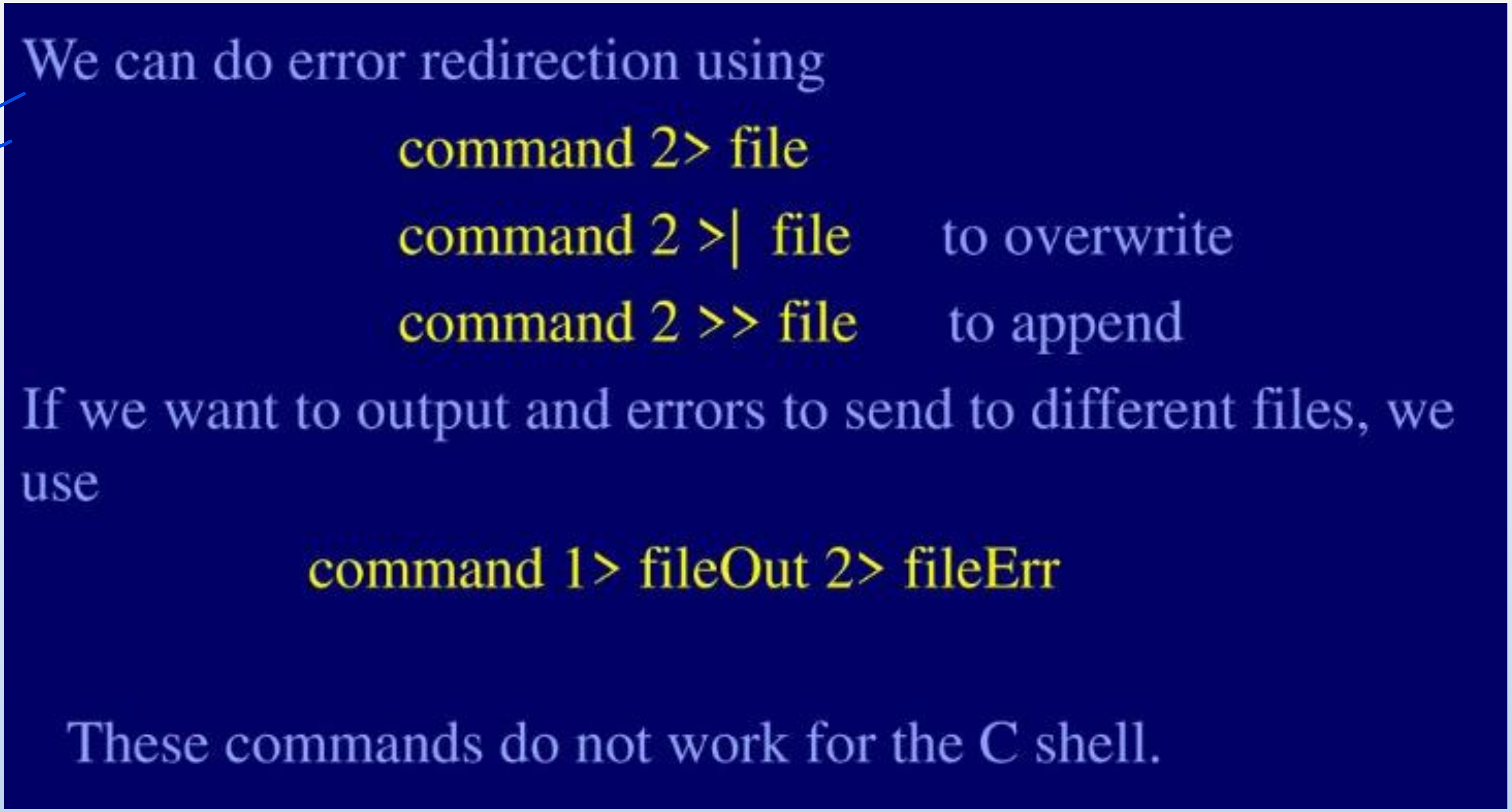
1. $who > whooct2 : only the output from this execution of the command
   - If the file does not exist, Unix creates it and writes the output
   - If the file already exists, it depends on the **noclobber** setting:
     - On: It prevents redirected output to destroy the existing file.
     - **$who > whooct2** throws an error whooct2: file already exists

2. >|: override existing file: empties the file then writes new o/p to the file.

3. >> : Append :

➢If the file does not exist, Unix creates it and writes the output.

➢If the file already exists, moves to the end of the file.



We can do error redirection using

```
command 2> file
command 2 >| file      to overwrite
command 2 >> file      to append
```

If we want to output and errors to send to different files, we use

```
command 1> fileOut 2> fileErr
```

These commands do not work for the C shell.

Redirecting Errors

- There are mainly two types of output streams in Linux-
- ***standard output*** **and** ***standard error*****.**
- The redirection operator (**command > file**) only redirects standard output and hence, the standard error is still displayed on the terminal.
- The default standard error is the screen.
- The standard error can also be redirected so that error messages do not clutter up the output of the program. '2' denotes the stderr of a program.

```
 $ ls –l file1 nofile
 Cannot access nofile : No such file or directory
-rw-r- -r- -  1 staff 1234 oct 2 18:16 file1
 $ ls –l file1 nofile 1>opfile
Cannot access nofile : No such file or directory  #on output terminal
$ more opfile
-rw-r- -r- -  1 staff 1234 oct 2 18:16 file1 #on file
```

Redirecting Errors

- **Redirecting to different files**
- ‘2>’ redirects the error of an output to a file.
- The Syntax of this operator is as follows:
- **command 2> file**

```
$ ls –l file1 nofile 1>opfile1 2>opfile2
```

```
$ more opfile1
  -rw-r- -r- -  1 staff 1234 oct 2 18:16 file1
```

```
$ more opfile2
Cannot access nofile : No such file or directory
```

- **Redirecting to one file**

- $ ls –l file1 nofile 1>opfile 2>opfile   #error, file already exists

  $ ls –l file1 nofile 1>| opfile 2>| opfile   # over ride oprator

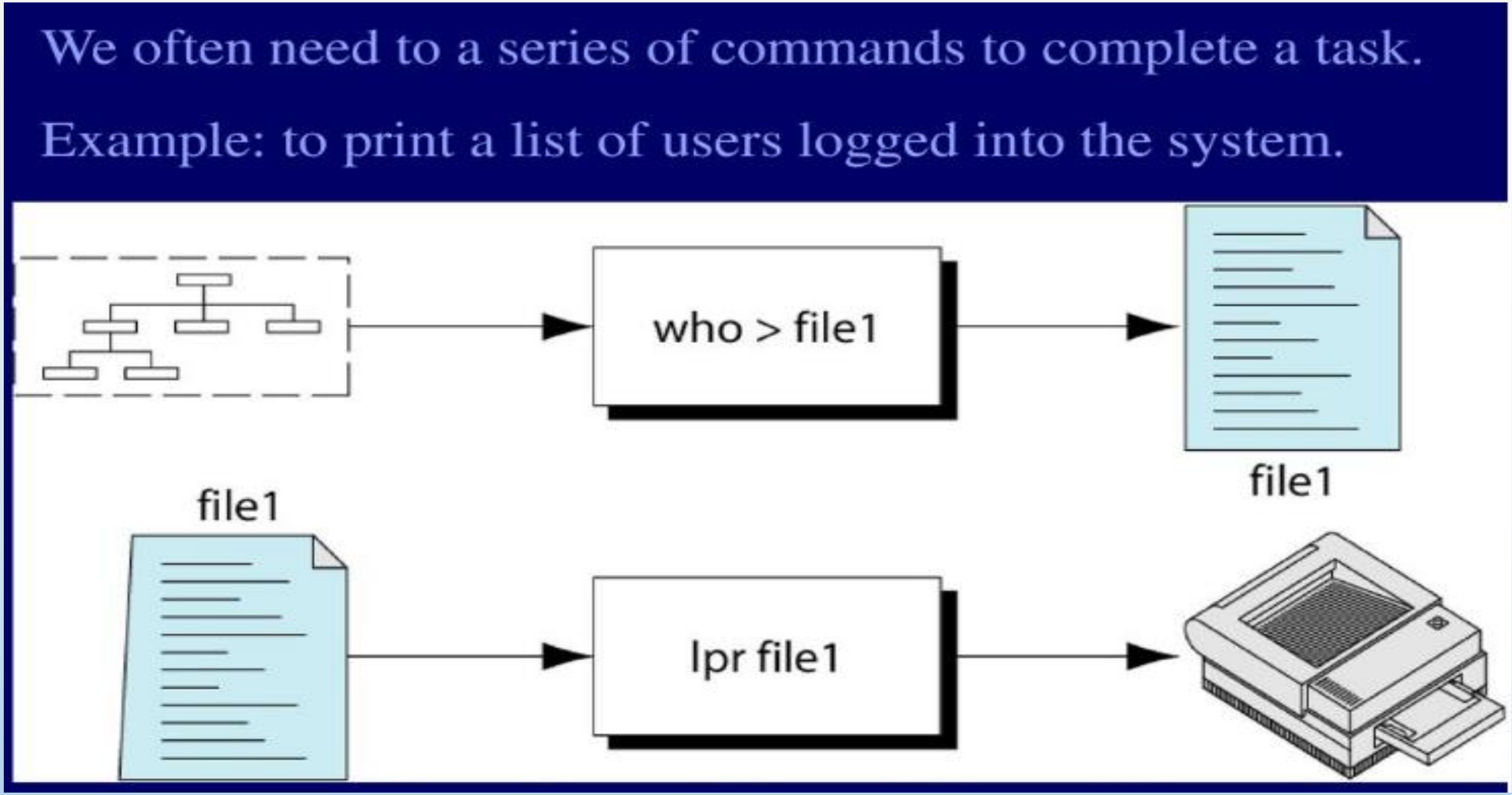
  $ ls opfile   # results of last o/p

  can not open the file

- When 2>& is used both standard error and standard output get redirected to the same file.

```
 $ ls –l file1 nofile 1> opfile  2>& 1
 $ more opfile
 Cannot access nofile : No such file or directory
-rw-r- -r- -  1 staff 1234 oct 2 18:16 opfile
```

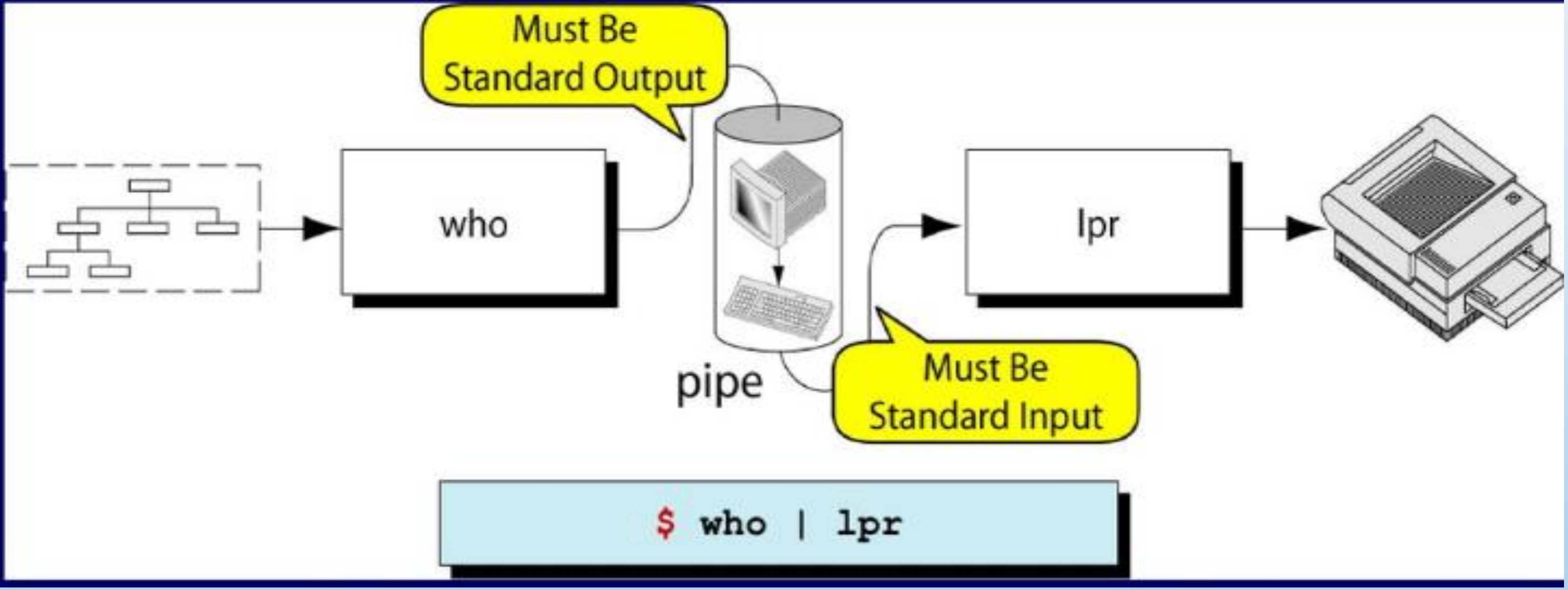
Pipes


We often need to a series of commands to complete a task.

Example: to print a list of users logged into the system.

Pipes

Pipe is an operator that temporarily saves the output of one command in a buffer that is used at the same timeas an input of the next command



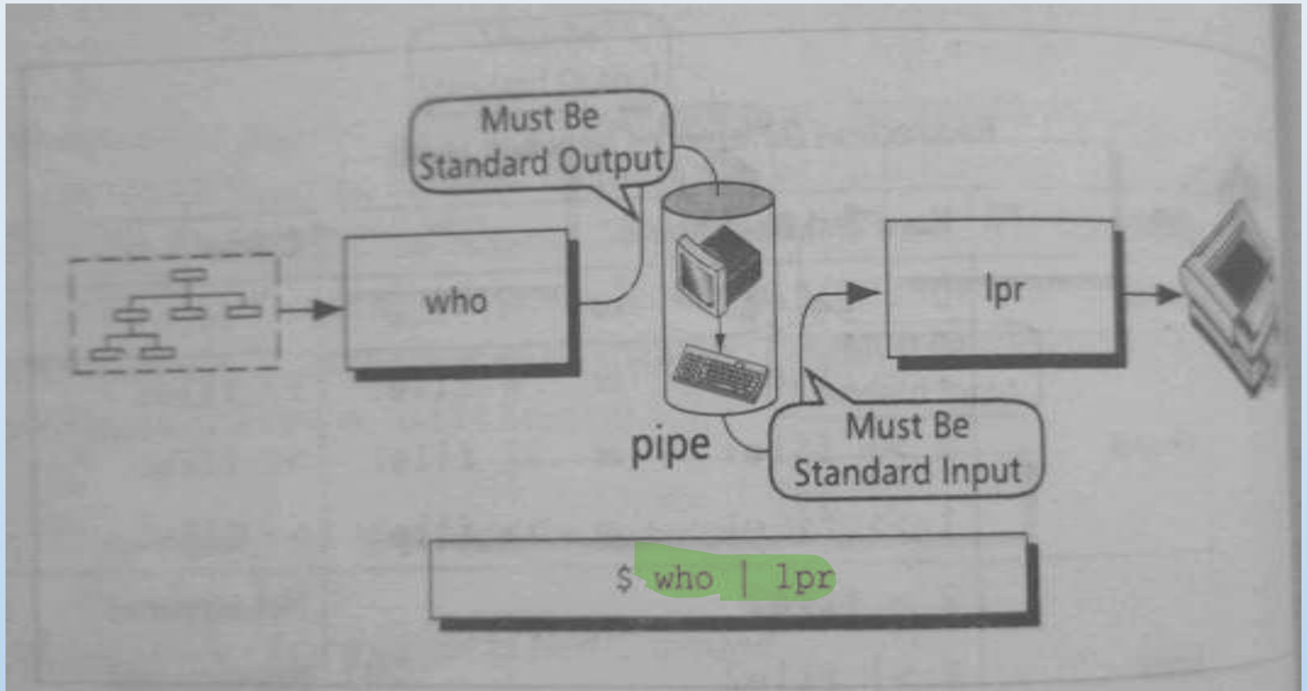
```
$ who | lpr
```

Pipes

- A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing.

AS AN INPUT

- The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character **'|'**.



```
$ who | lpr
```

FIGURE 5.8 *Piping Output to Next Command*

Pipes

- Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on.

- It can also be visualized as a temporary connection between two or more commands/ programs/ processes.
- The command line programs that do the further processing are referred to as **filters**.

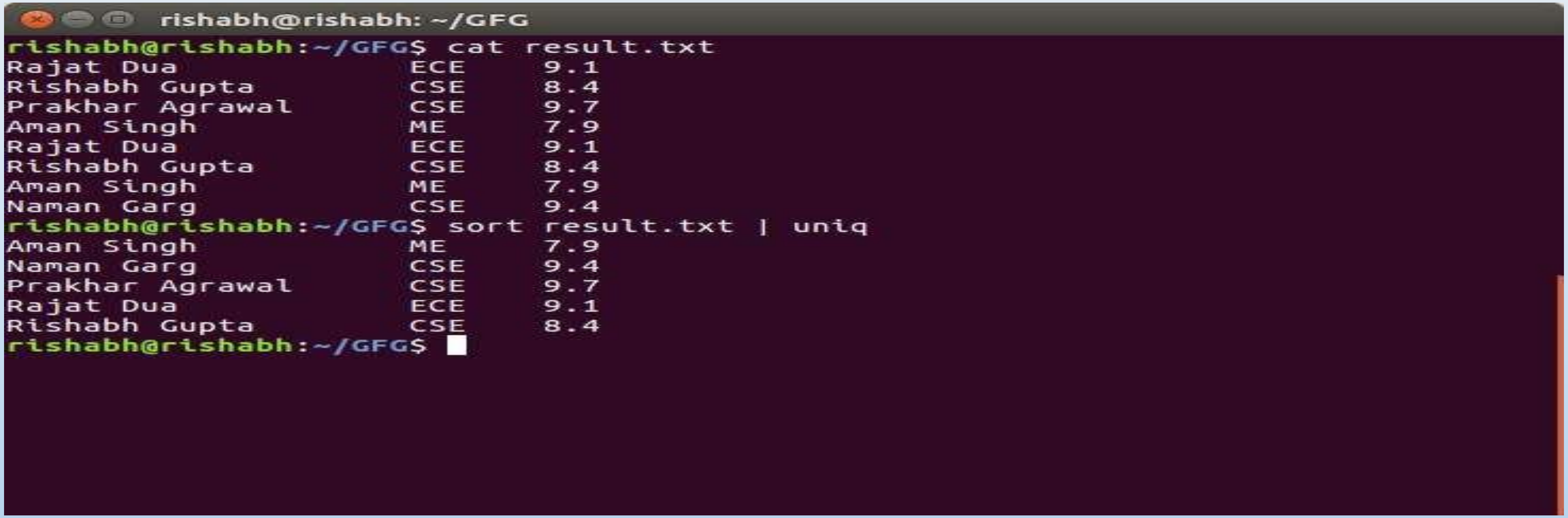
- This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen.

- Pipes are unidirectional **i.e data flows from left to right through the pipeline.**

- Syntax: **command_1 | command_2 | command_3 | .... | command_N**

- **Use sort and uniq command to sort a file and print unique values.**

  $ sort record.txt | uniq



```
rishabh@rishabh: ~/GFG
rishabh@rishabh:~/GFG$ cat result.txt
Rajat Dua               ECE     9.1
Rishabh Gupta           CSE     8.4
Prakhar Agrawal         CSE     9.7
Aman Singh              ME      7.9
Rajat Dua               ECE     9.1
Rishabh Gupta           CSE     8.4
Aman Singh              ME      7.9
Naman Garg              CSE     9.4
rishabh@rishabh:~/GFG$ sort result.txt | uniq
Aman Singh              ME      7.9
Naman Garg              CSE     9.4
Prakhar Agrawal         CSE     9.7
Rajat Dua               ECE     9.1
Rishabh Gupta           CSE     8.4
rishabh@rishabh:~/GFG$ 
```

tee command

- **tee command** reads the standard input and writes it to both the standard output and one or more files.
- It does both the tasks simultaneously, copies the result into the specified files and also display the result.
- If files do not exists it creates the output file and overwrites them if exists.
- To prevent over written we can use **‘-a’** which append the output to existing files.

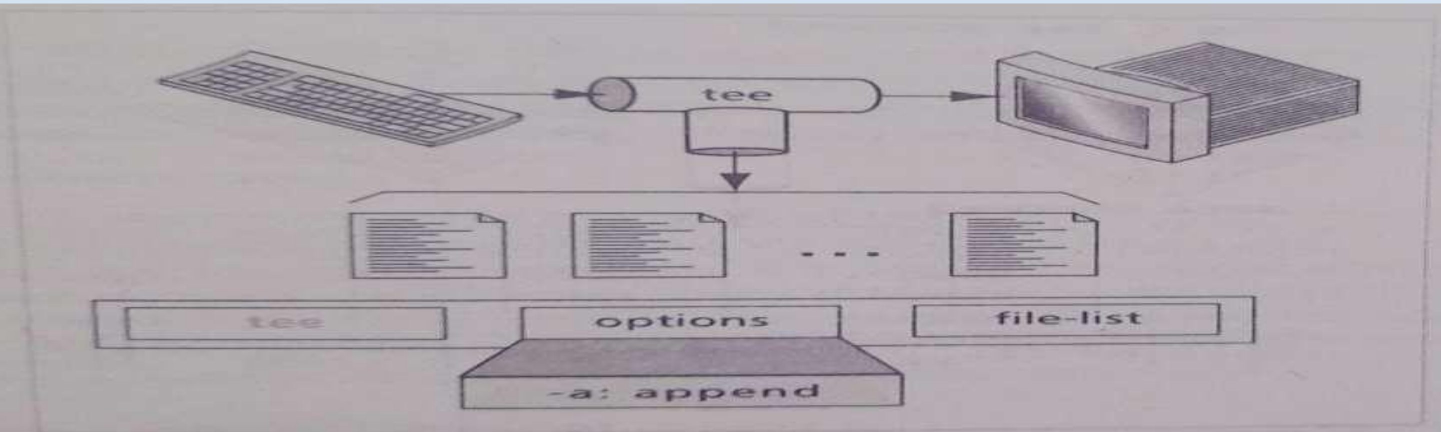


FIGURE 5.9 *The* **tee** *Command*

tee command

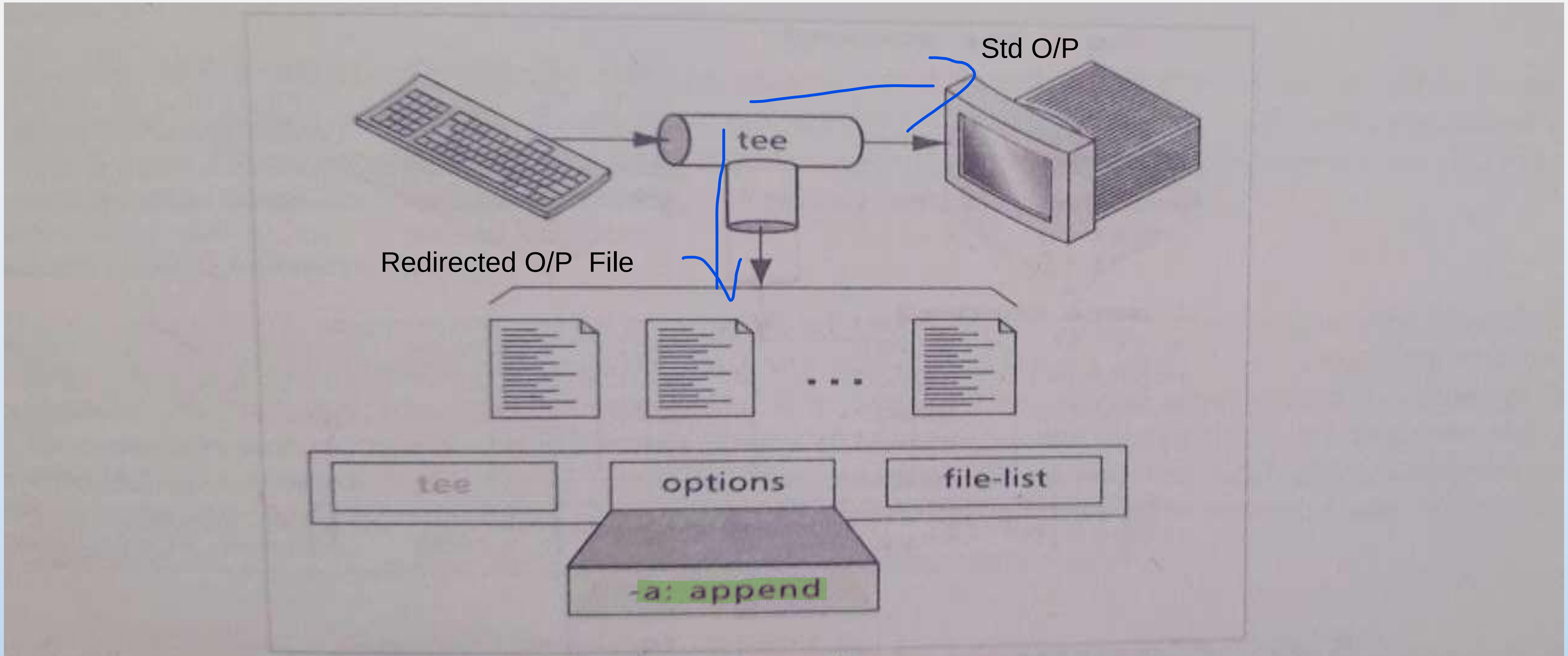
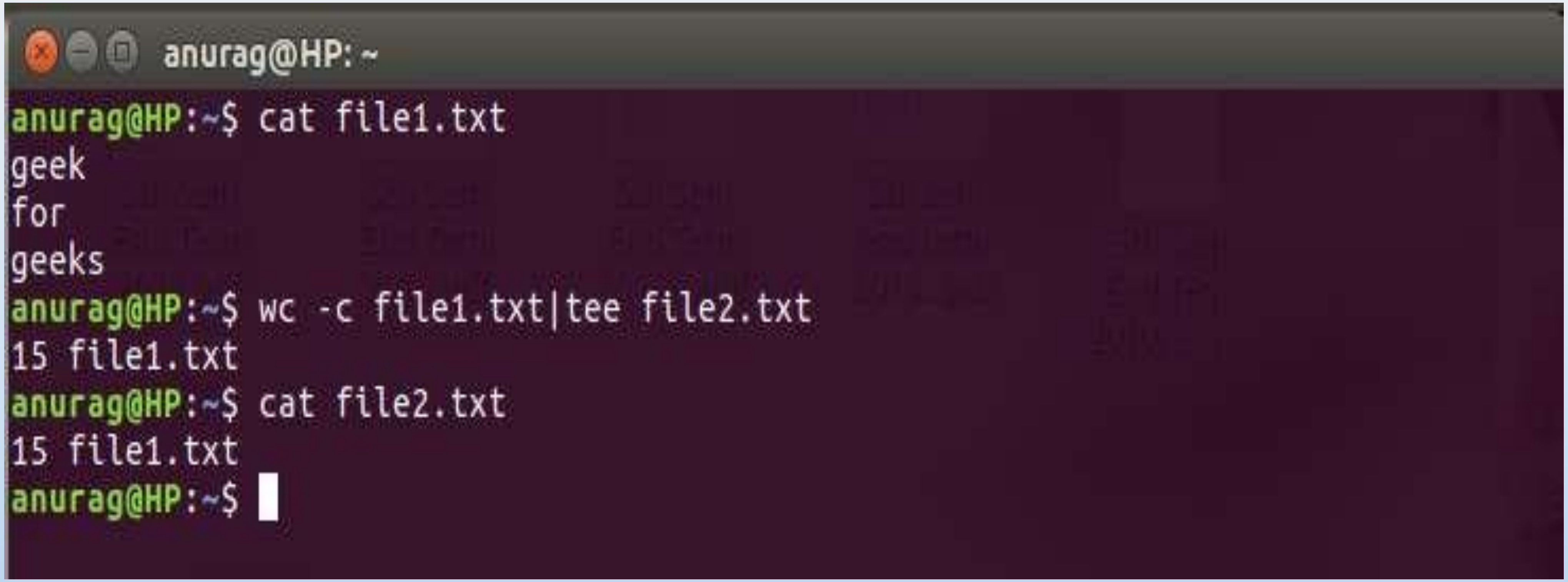


FIGURE 5.9 *The* **tee** *Command*

SYNTAX:

tee [OPTION]... [FILE]...



```
anurag@HP:~$ cat file1.txt
geek
for
geeks
anurag@HP:~$ wc -c file1.txt|tee file2.txt
15 file1.txt
anurag@HP:~$ cat file2.txt
15 file1.txt
anurag@HP:~$ 
```

Command Execution

Sometimes we need to combine several commands.

There are four formats for combining commands into one line: sequenced, grouped, chained, and conditional.

Sequenced Commands

A sequence of commands can be entered on one line. Each command must be separated from its predecessor by semicolon.

There is no direct relationship between the commands.

```
command1; command2; command3
```

Command Execution

- **Sequenced** Commands
  - Sequence of commands can be entered on one line. Each command must be separated from its predecessor by semicolon. One command does not communicate with other.
  - Ex: $ echo “\n Hello \nMonth” >June2023; cal 10 2022 >>June2023

- **Grouped** Commands: Commands are grouped by placing them in parentheses
  - Ex: $ (echo “\n Hello \nMonth” ; cal 10 2023) >June2023

    $ more June2023

- **Chained** Commands: Combine commands using pipes.

  $ who | lpr

- **Conditional** Commands : Combine two or more commands using conditional relationships and (&&), or (||)

  Ex: cp f1 f2 && echo “copy successful” #2nd command executes only when first is successful.

  Ex: cp f1 f2 || echo “copy failed” #second command executes only when first fails

Command Line Editing : Vi Edit Mode

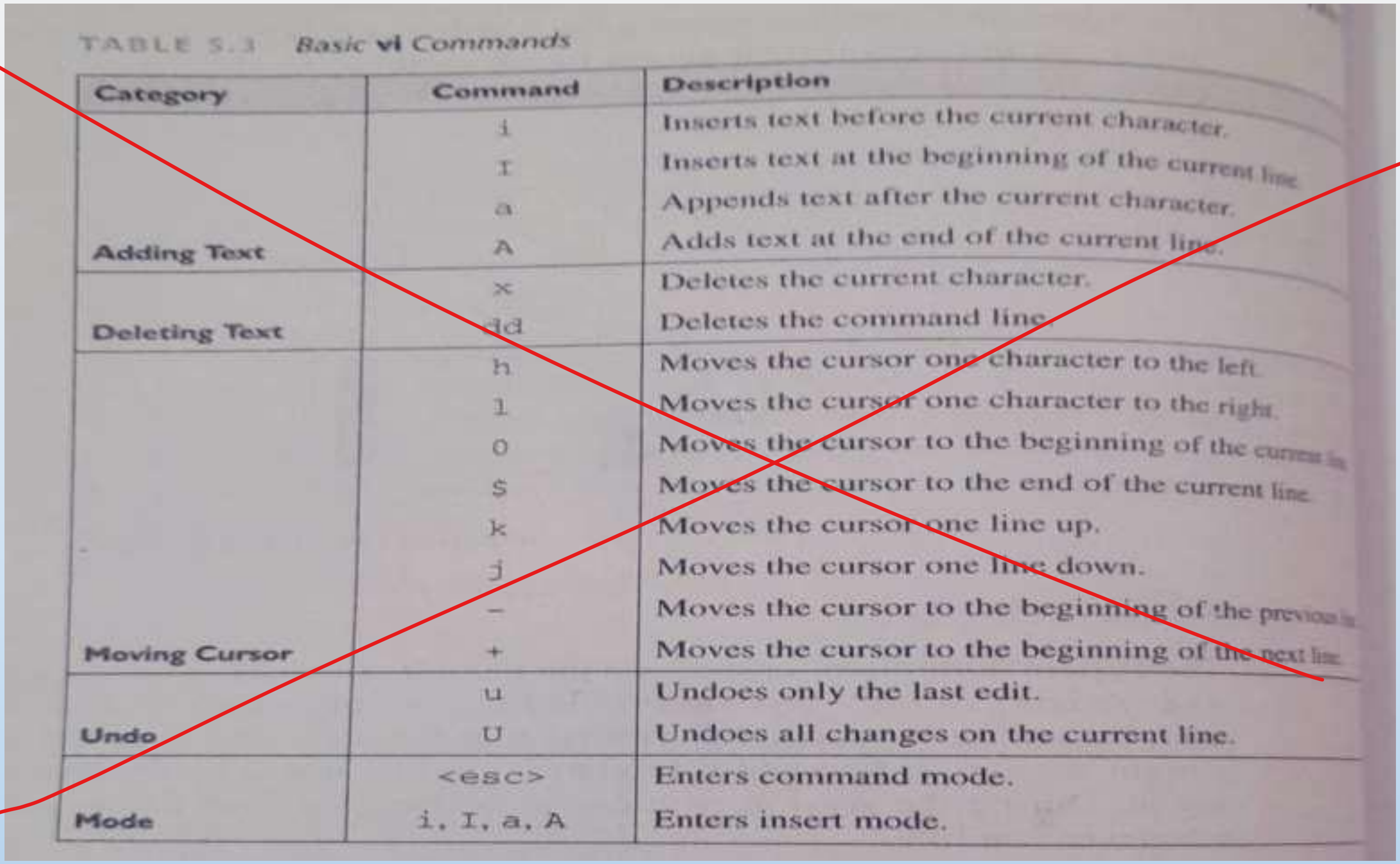


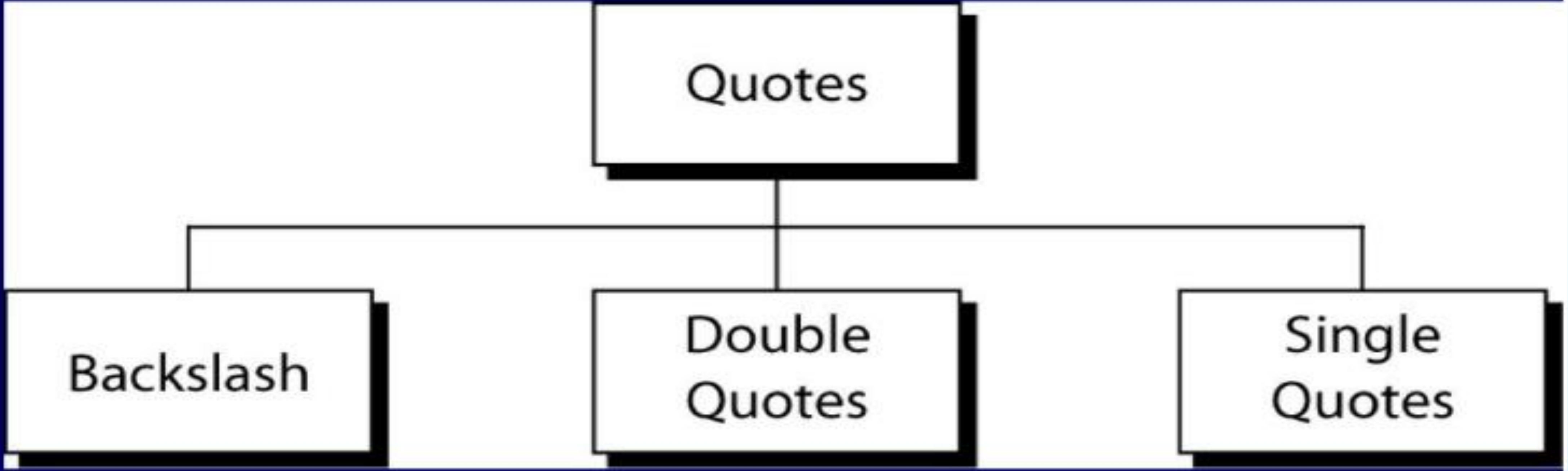
TABLE 5.3 *Basic **vi** Commands*

Category	Command	Description
	i	Inserts text before the current character.
	I	Inserts text at the beginning of the current line.
	a	Appends text after the current character.
Adding Text	A	Adds text at the end of the current line.
	x	Deletes the current character.
Deleting Text	dd	Deletes the command line.
	h	Moves the cursor one character to the left.
	l	Moves the cursor one character to the right.
	0	Moves the cursor to the beginning of the current line.
	$	Moves the cursor to the end of the current line.
	k	Moves the cursor one line up.
	j	Moves the cursor one line down.
	–	Moves the cursor to the beginning of the previous line.
Moving Cursor	+	Moves the cursor to the beginning of the next line.
	u	Undoes only the last edit.
Undo	U	Undoes all changes on the current line.
	<esc>	Enters command mode.
Mode	i, I, a, A	Enters insert mode.

Quotes

Metacharacters are characters that have a special interpretation, for example the pipe |.

We need a way to tell the shell interpreter when we want to use them as text characters.

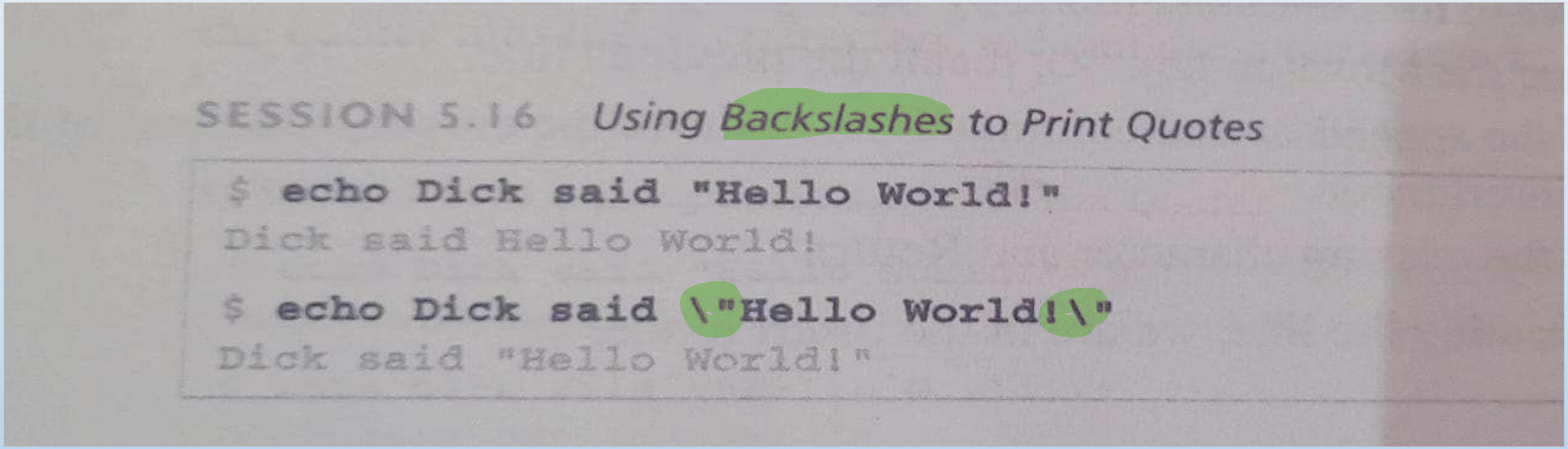


Quotes

- Shells use metacharacters (that have special interpretation) in commands. Example Pipe (|).

\, ' ', " "

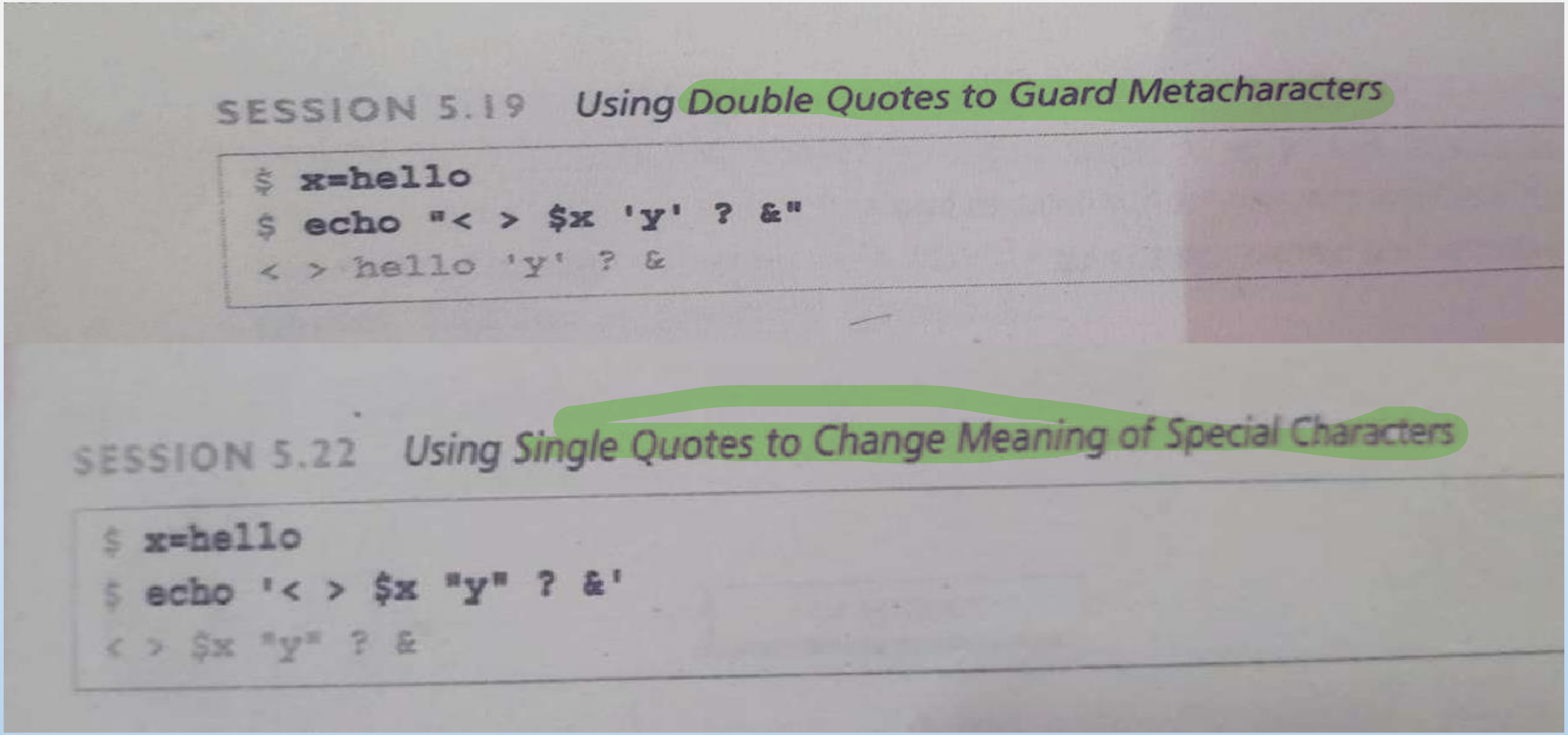
- Three of metacharacters (**Back slash**, **Single quotes**, **Double quotes) known collectively as quotes.**



SESSION 5.16 *Using Backslashes to Print Quotes*

```
$ echo Dick said "Hello World!"
Dick said Hello World!
$ echo Dick said \"Hello World!\"
Dick said "Hello World!"
```

Quotes



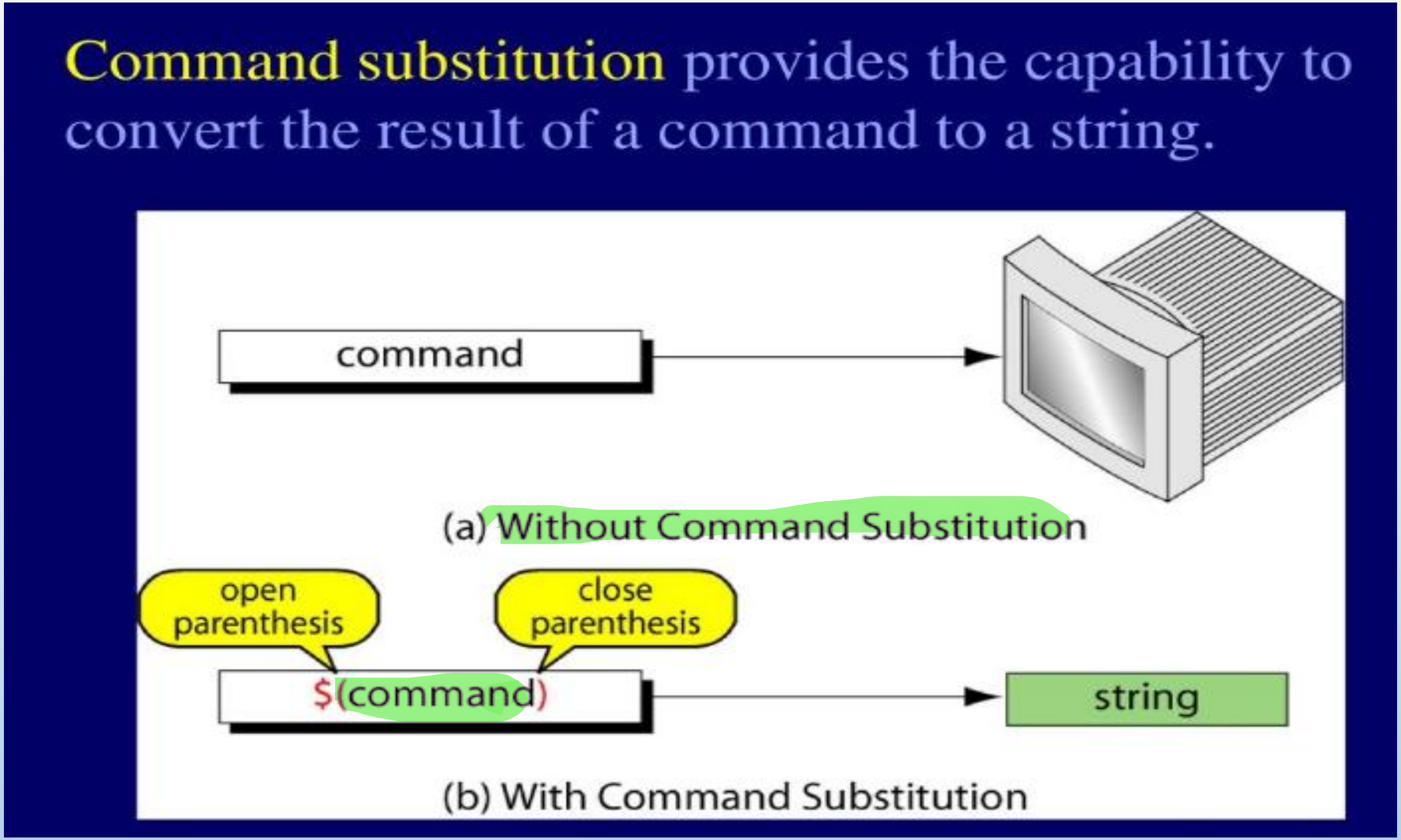
SESSION 5.19 *Using Double Quotes to Guard Metacharacters*

```
$ x=hello
$ echo "< > $x 'y' ? &"
< > hello 'y' ? &
```

SESSION 5.22 *Using Single Quotes to Change Meaning of Special Characters*

```
$ x=hello
$ echo '< > $x "y" ? &'
< > $x "y" ? &
```

Command substitution



Command substitution provides the capability to convert the result of a command to a string.

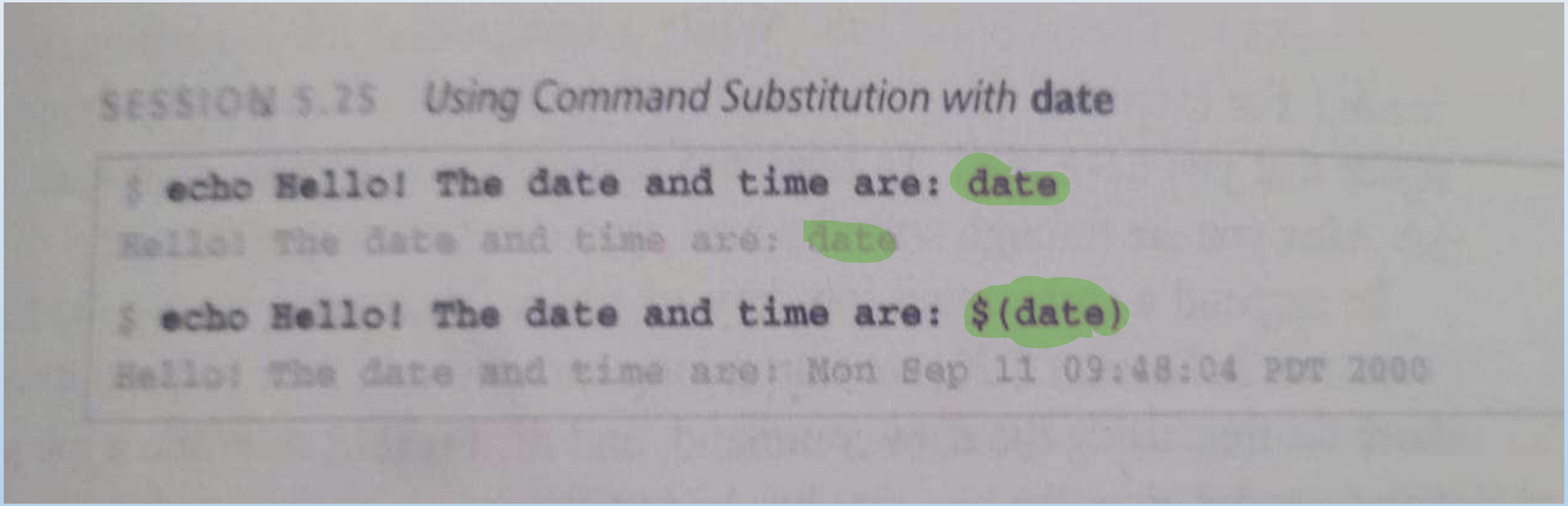
(a) Without Command Substitution

(b) With Command Substitution

Command substitution

- Command substitution **provides the capability to convert the result of a command to a string**.

  The command substitution operator is **$()**



SESSION 5.25 *Using Command Substitution with* **date**

```
$ echo Hello! The date and time are: date
Hello! The date and time are: date

$ echo Hello! The date and time are: $(date)
Hello! The date and time are: Mon Sep 11 09:48:04 PDT 2000
```

Job Control

- One of the important features of a **shell** is job control.
- In general a job is a user task run on the computer.
- Editing, sorting and reading mail are **all examples** of jobs.

  However UNIX has a specific **definition** of a job.
- **A job is a command or set of commands entered on one command line.**

  For example:
- $ ls
- $ ls | lpr
- Both are jobs.

UNIX defines two types of jobs: foreground and background.

- Because UNIX is a multitasking operating system, we can run more than one job at a time.
- **To allow multiple jobs, UNIX defines two types of jobs: foreground and background.**

Foreground

A foreground job is any job run under the active supervision of the user.

- It is started by the user and may interact with the user through standard input and output.
- While it is running, no other jobs may be started.
- To start a foreground job, we simply enter a command and key Return.
- Keying Return at the end of the command starts it in the foreground.
- Suspending a foreground job While a foreground job is running it can be suspended.
- To suspend the foreground job, key ctrl+z.
- To resume it, use the foreground command fg.
- Terminating a foreground job If for any reason we want to terminate (kill) a running foreground job, we use the cancel meta-character, ctrl+c.
- If the job has been suspended, it must first be resumed using the foreground command. and then killed

BACKGROUND JOBS:

- When we know a job will take a long time, we may want to run it in the background - using &.
- Jobs run in the background free the keyboard and monitor so that we may use them for other tasks like editing files and sending mail.

Note: Foreground and Background jobs share the keyboard and monitor.

- Suspending, Restarting and Terminating Background jobs:

➢ To suspend the background job, we use the stop command.

➢ To restart it, we use the bg command.

➢ To terminate the background job, we use the kill command.

➢ All three commands require the job number prefaced with a percent sign (%).

kill %JobID   Kills that very job

Example:

```
$ longjob.scr&
[1] 1795841
$ stop %1
[1] + 1795841 stopped (SIGSTOP) longjob.scr&
$ bg %1
1    longjob.scr&
$ kill %1
[1] + Terminated longjob.scr&
```

- Moving between Background and Foreground:

➢ To move a job between the foreground and background, the job must be suspended.

➢ Once the job is suspended, we can move it from the suspended state to the background with the bg command. Because job is in the foreground no job number is required.

➢ To move a background job to a foreground job, we use the fg command.

Moving between Foreground and background

- $ longsrc.scr
- Ctrl+z

      [1] + stopped (SIGTSTP)   longsrc.scr

- $ bg

1   jobsrc.src

- $ fg %1

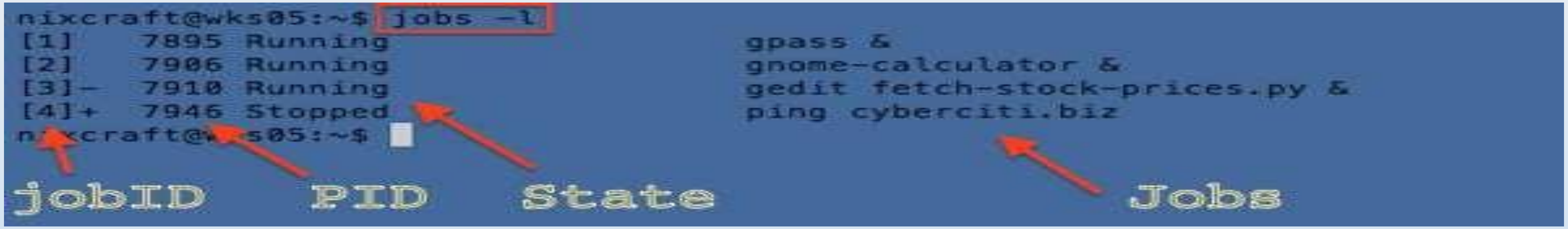
longsrc.rsc

fg to bg doesnt need %JobID
bcuz only one job is present

But for fg u need jobID

MULTIPLE BACKGROUND JOBS:

- When multiple background jobs are running in the background, the job number is required on commands to identify which job we want to affect.
- Jobs command to list the current jobs and their status, we use the jobs command. This command lists all jobs. Whether or not they are running or stopped.
- For each job, it shows the job number, currency, and state - running or stopped.



```
nixcraft@wks05:~$ jobs -l
[1]    7895 Running                        gpass &
[2]    7906 Running                        gnome-calculator &
[3]-   7910 Running                        gedit fetch-stock-prices.py &
[4]+   7946 Stopped                        ping cyberciti.biz
nixcraft@wks05:~$
```

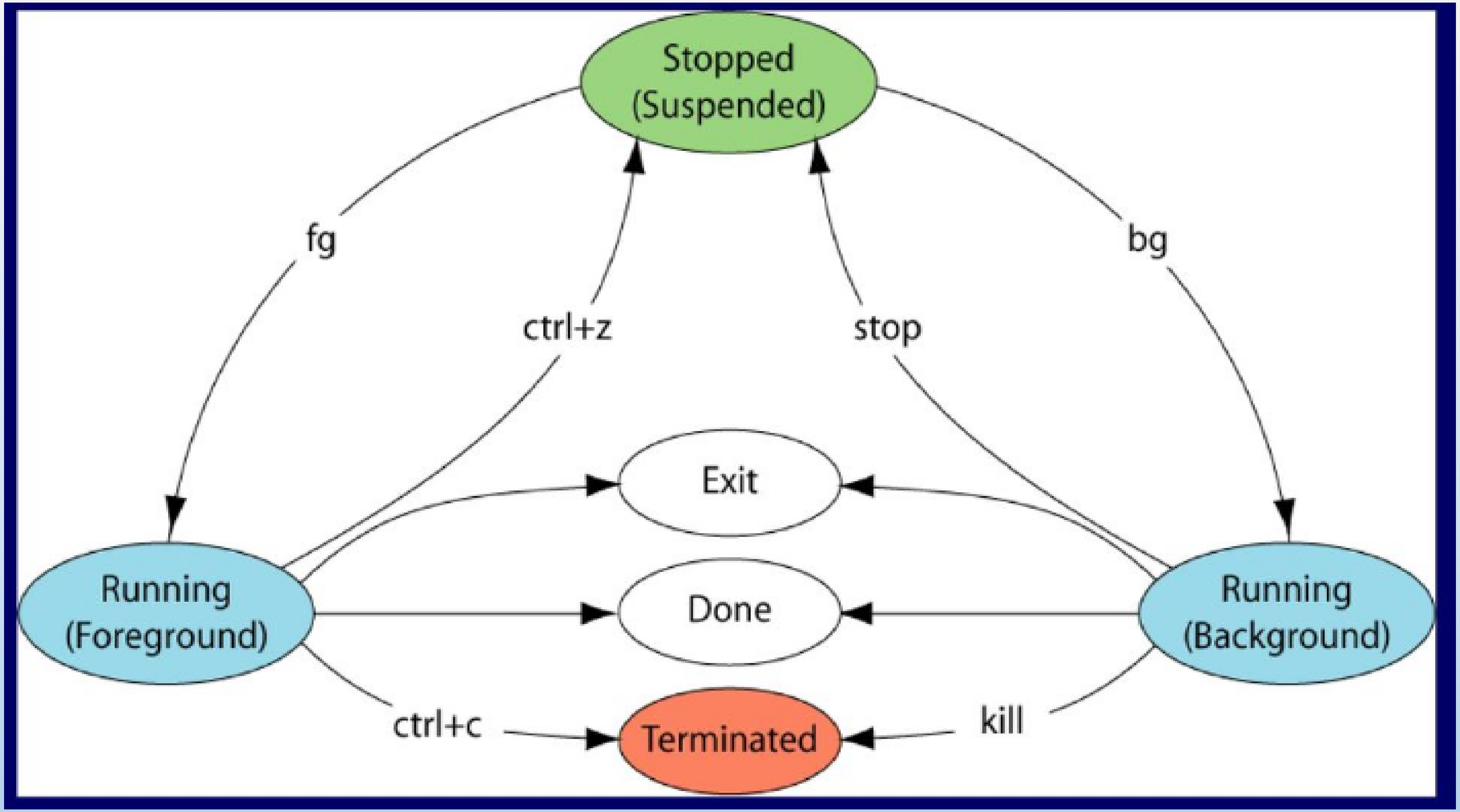
Currency Flag:

+ which job is the default if a command entered without job id.

- which job will be default if first job were to complete.

Imp

Job States



Aliases

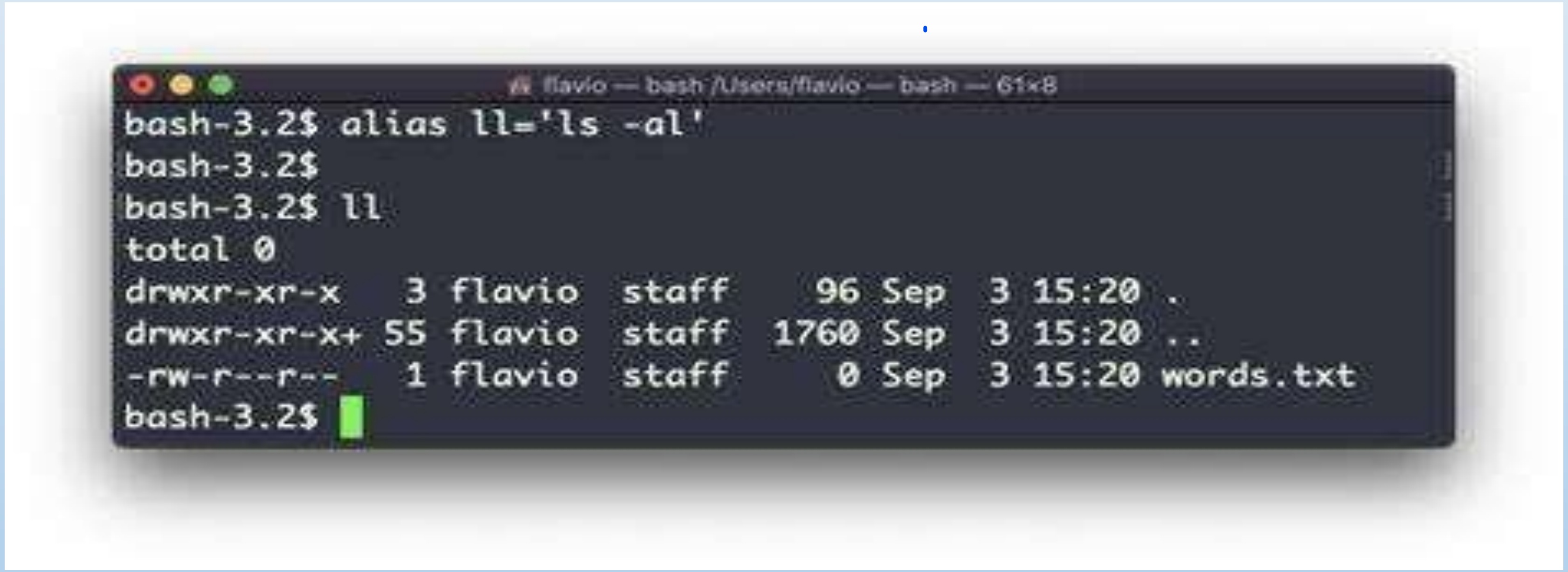
- An Alias provides a means of creating customized commands by assigning name to a command

alias name=command-definition

No space before and after command-definition.

```
$ alias dir=ls
$ dir


$ alias dir=ls -l
$ dir
```



```
bash-3.2$ alias ll='ls -al'
bash-3.2$
bash-3.2$ ll
total 0
drwxr-xr-x    3 flavio  staff    96 Sep  3 15:20 .
drwxr-xr-x+ 55 flavio  staff  1760 Sep  3 15:20 ..
-rw-r--r--    1 flavio  staff     0 Sep  3 15:20 words.txt
bash-3.2$
```

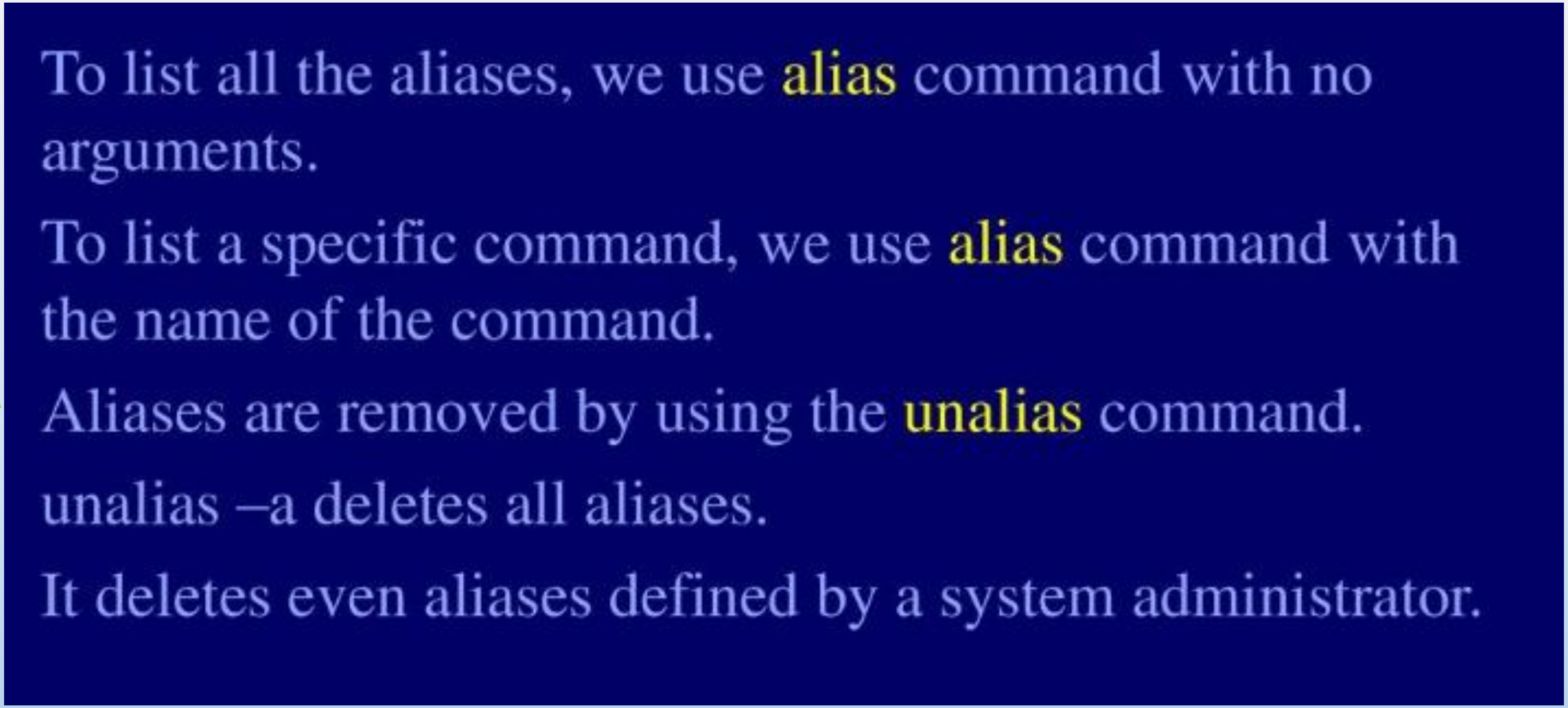
Arguments to Alias Command

- $ alias f1 = “ls –l”   alias name="command"
- $ f1 f*

Ambiguous Alias Command

- $alias lndir=‘ls –l | more’   alias name='command1 | command2'
- $ lndir

Alias in Korn and Bash Shells



To list all the aliases, we use alias command with no arguments.

To list a specific command, we use alias command with the name of the command.

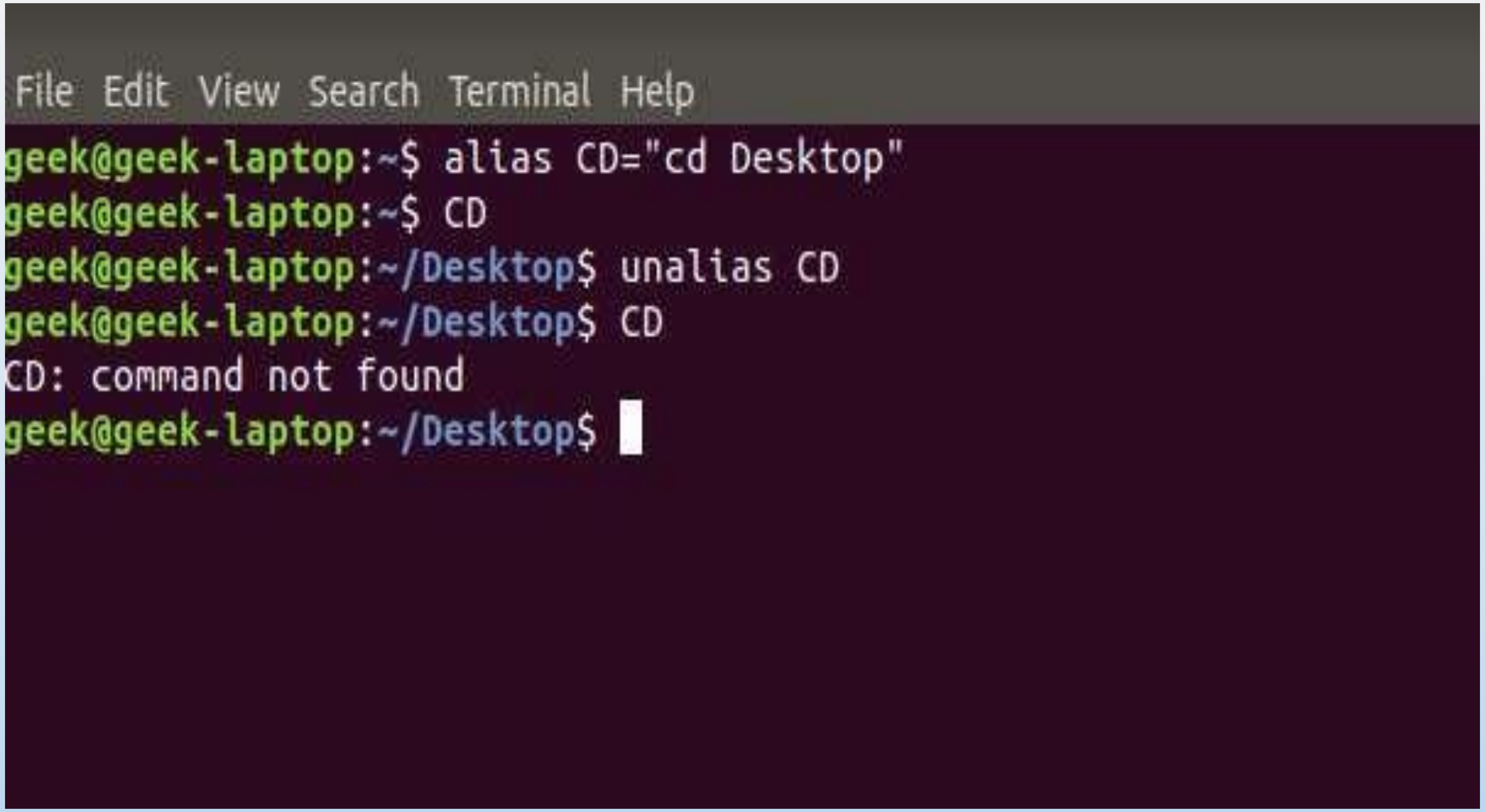
Aliases are removed by using the unalias command.

unalias –a deletes all aliases.

It deletes even aliases defined by a system administrator.

Listing Alias & removing

- $ alias
- $ alias dir
- unalias dir



```
geek@geek-laptop:~$ alias CD="cd Desktop"
geek@geek-laptop:~$ CD
geek@geek-laptop:~/Desktop$ unalias CD
geek@geek-laptop:~/Desktop$ CD
CD: command not found
geek@geek-laptop:~/Desktop$
```

Alias definition in C shell

- $ alias name definition
- $ alias dir ls
- $alias f1 ‘ls -l \!* | more’
- $ f1 TheRaven

Designator	Meaning
\!*	Position of only argument
\!^	Position of first argument
\!$	Position of last argument
\!:n	Position of the nth argument

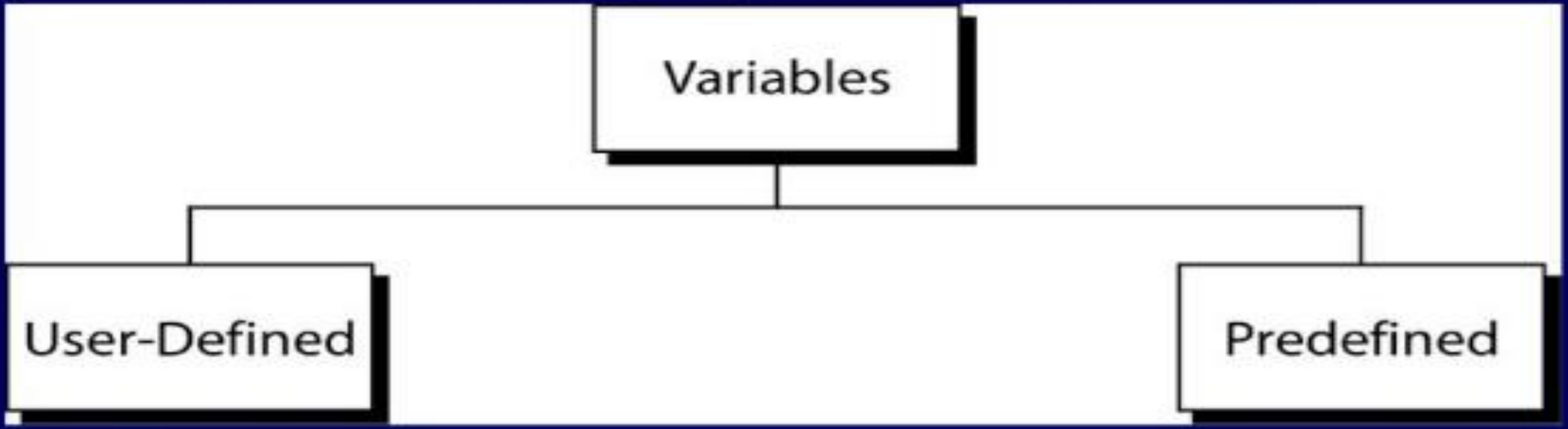
Alias Summary

Feature	Korn & Bash	C
Define	$ alias x=command	% alias x command
Argument	Only at end	Anywhere
List	$ alias	% alias
Remove	$ unalias x y z	% unalias x y z
Remove all	$ unalias -a	% unalias *

Variables

A variable is a location in memory where values can be stored.

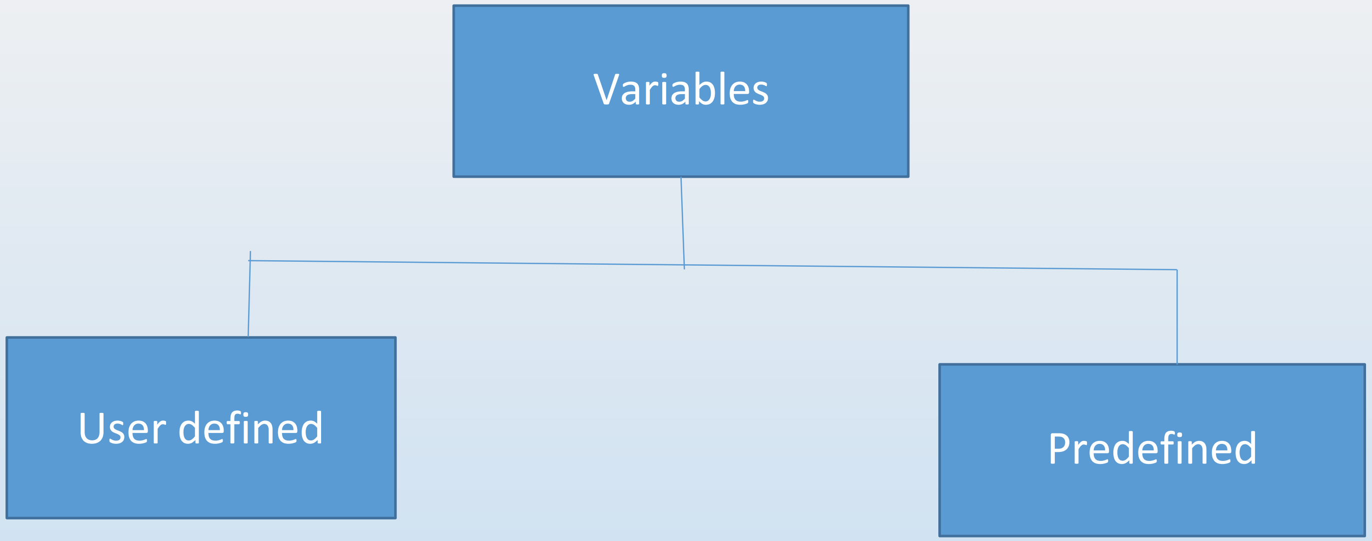
Each variable has a name, which starts with an alphabetic or _ character followed by alphanumeric or _ characters.



Not defined in UNIX, defined by a user

Defined in UNIX, used to configure a shell environment

Variables

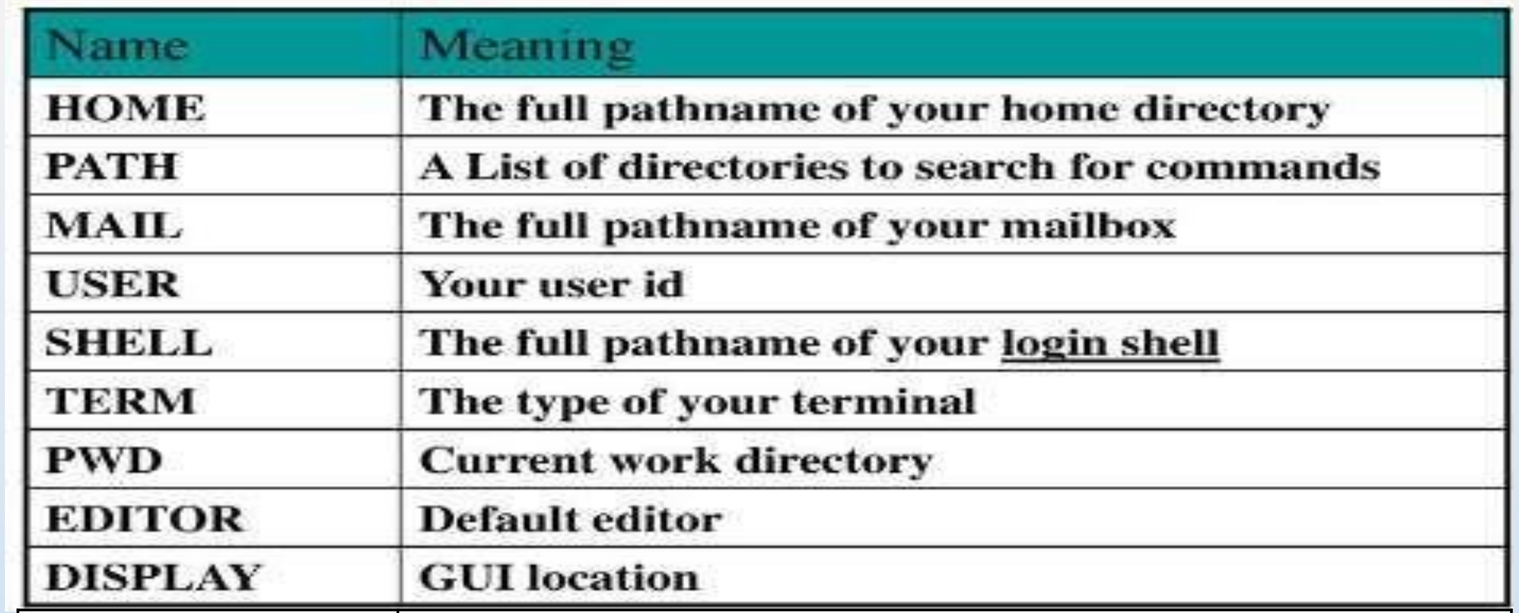


```
x= 23
echo $x
23
x=Hello
echo $x
Hello

 set x= 23
echo $x
23
set x=Hello
echo $x
Hello
```

Action	Korn & Shell	C Shell
Assignment	variable=value	set variable=value
Reference	$variable	$variable

Predefined Variables



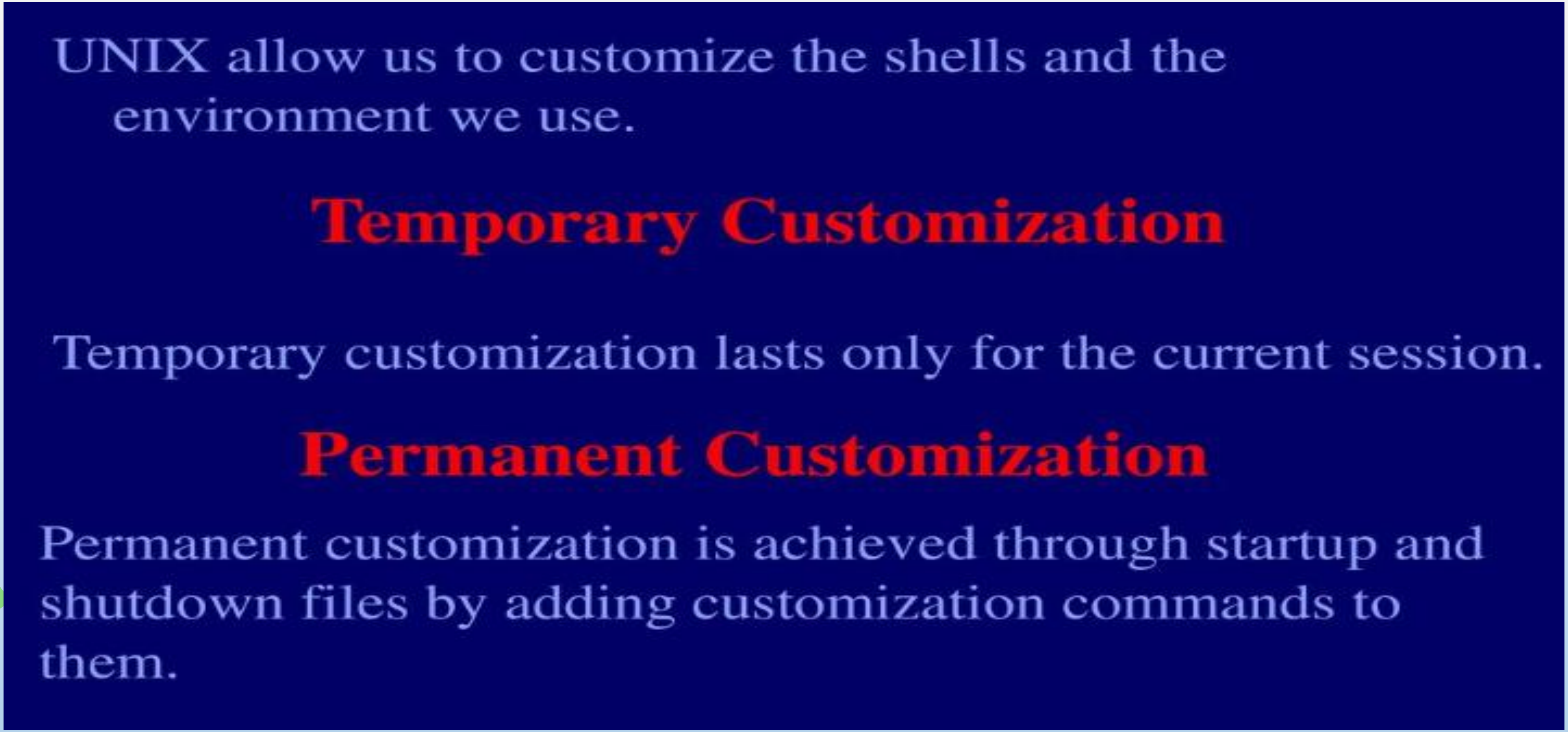
Name	Meaning
HOME	The full pathname of your home directory
PATH	A List of directories to search for commands
MAIL	The full pathname of your mailbox
USER	Your user id
SHELL	The full pathname of your login shell
TERM	The type of your terminal
PWD	Current work directory
EDITOR	Default editor
DISPLAY	GUI location
CDPATH	**Contains the search path for cd command when the directory argument is a relative pathname**
ENV	Path name for environment file
PS1	Primary prompt
TMOUT	Defines ideal time in seconds, before shell automatically log you off
VISUAL	Path name for editor

Options

- Global(noglob) : off : enables wildcard character matching
  on: disables wildcard character matching
- Print Commands(verbose and xtrace)
- Command Line editor(vi and emacs)\
- Ignore end of file(ignoreeof)
- No Clobber Redirection(noclobber)-to avoid replace of files
- Handling options

set	set –o option	set option
unset	set +o option	unset option
Display all	set –o	set

Shell/environment Customization

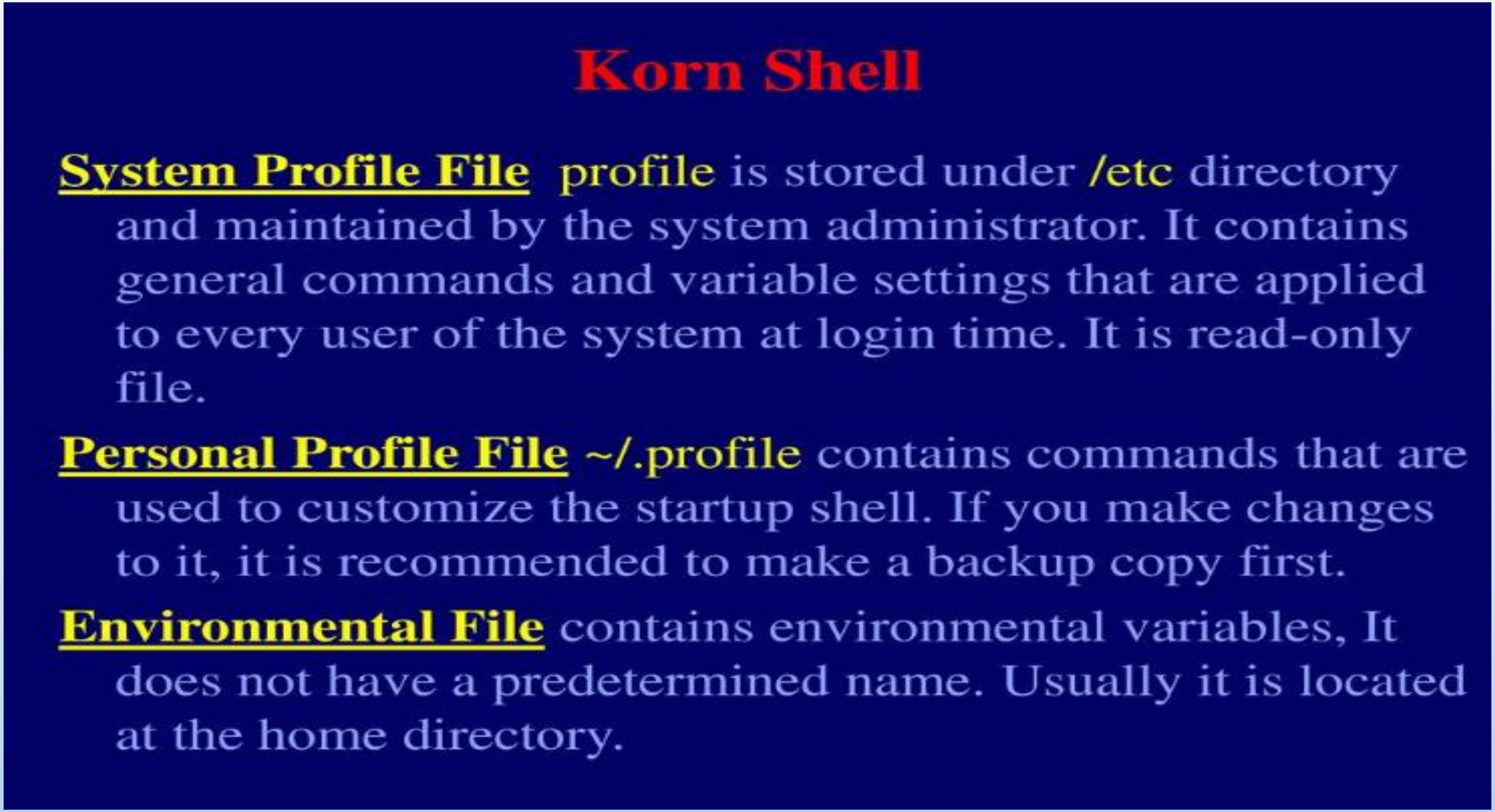

UNIX allow us to customize the shells and the environment we use.

Temporary Customization

Temporary customization lasts only for the current session.

Permanent Customization

Permanent customization is achieved through startup and shutdown files by adding customization commands to them.



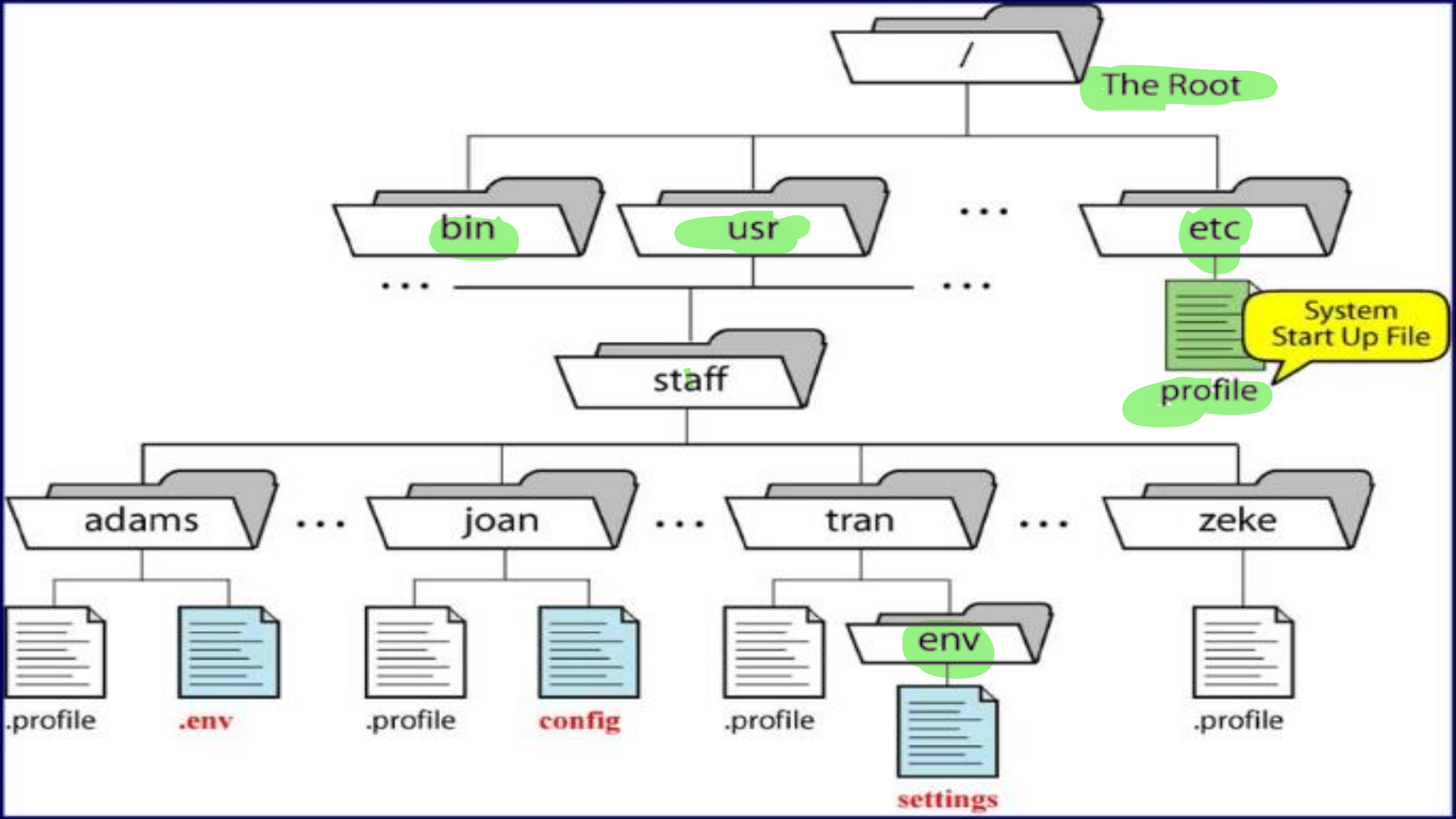
Korn Shell

<u>System Profile File</u> profile is stored under /etc directory and maintained by the system administrator. It contains general commands and variable settings that are applied to every user of the system at login time. It is read-only file.

<u>Personal Profile File</u> ~/.profile contains commands that are used to customize the startup shell. If you make changes to it, it is recommended to make a backup copy first.

<u>Environmental File</u> contains environmental variables, It does not have a predetermined name. Usually it is located at the home directory.

Korn's Environmental File



Bash Shell

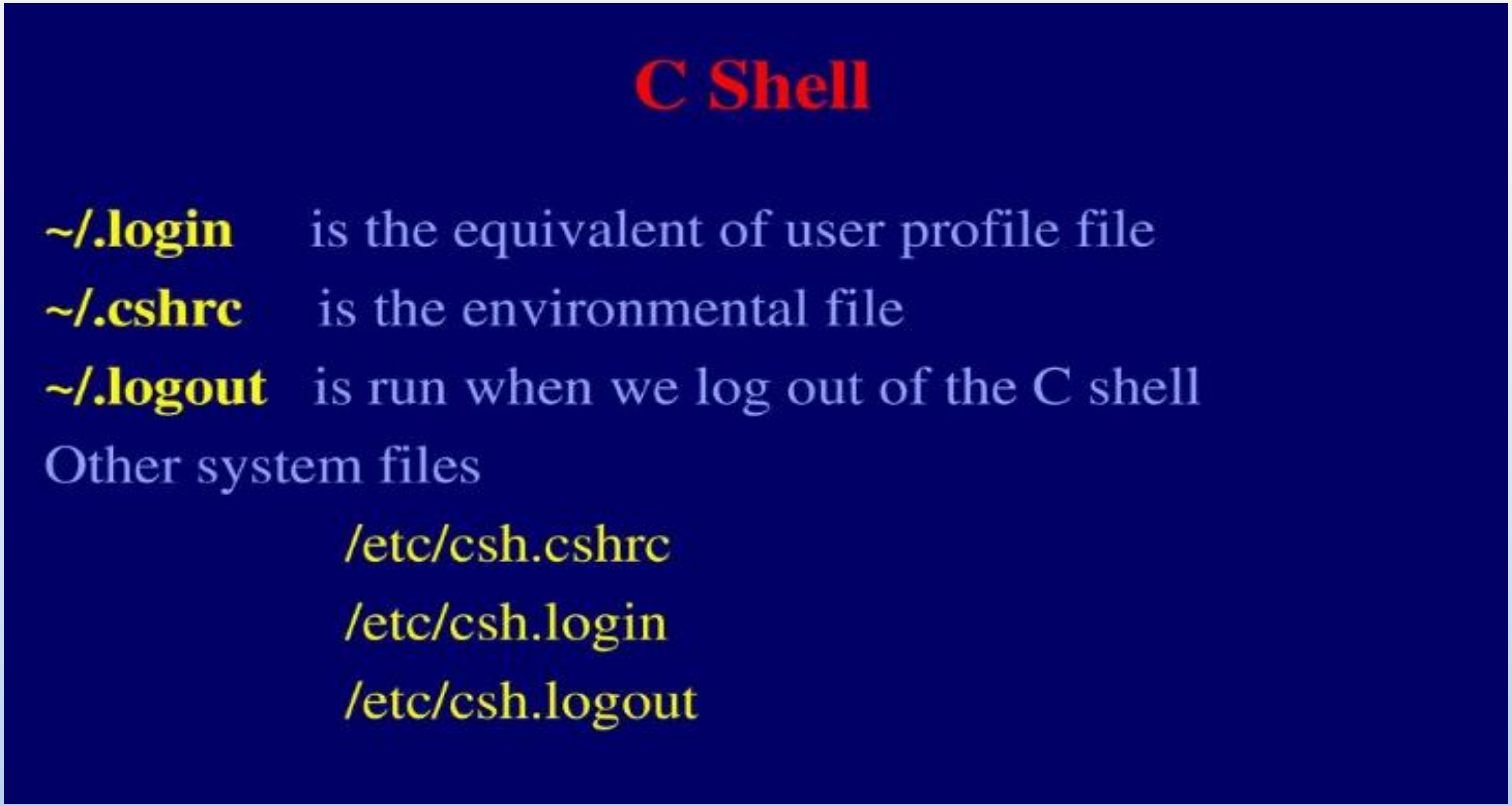
/etc/profile is used for the system profile file.

For personal profile file, one of the three files is used:

~/.bash_profile **~/.bash_login** **~/.profile**

The environmental filename is stored under the BASH_ENV variable.

There is a logout file **~/.bash_logout**



C Shell

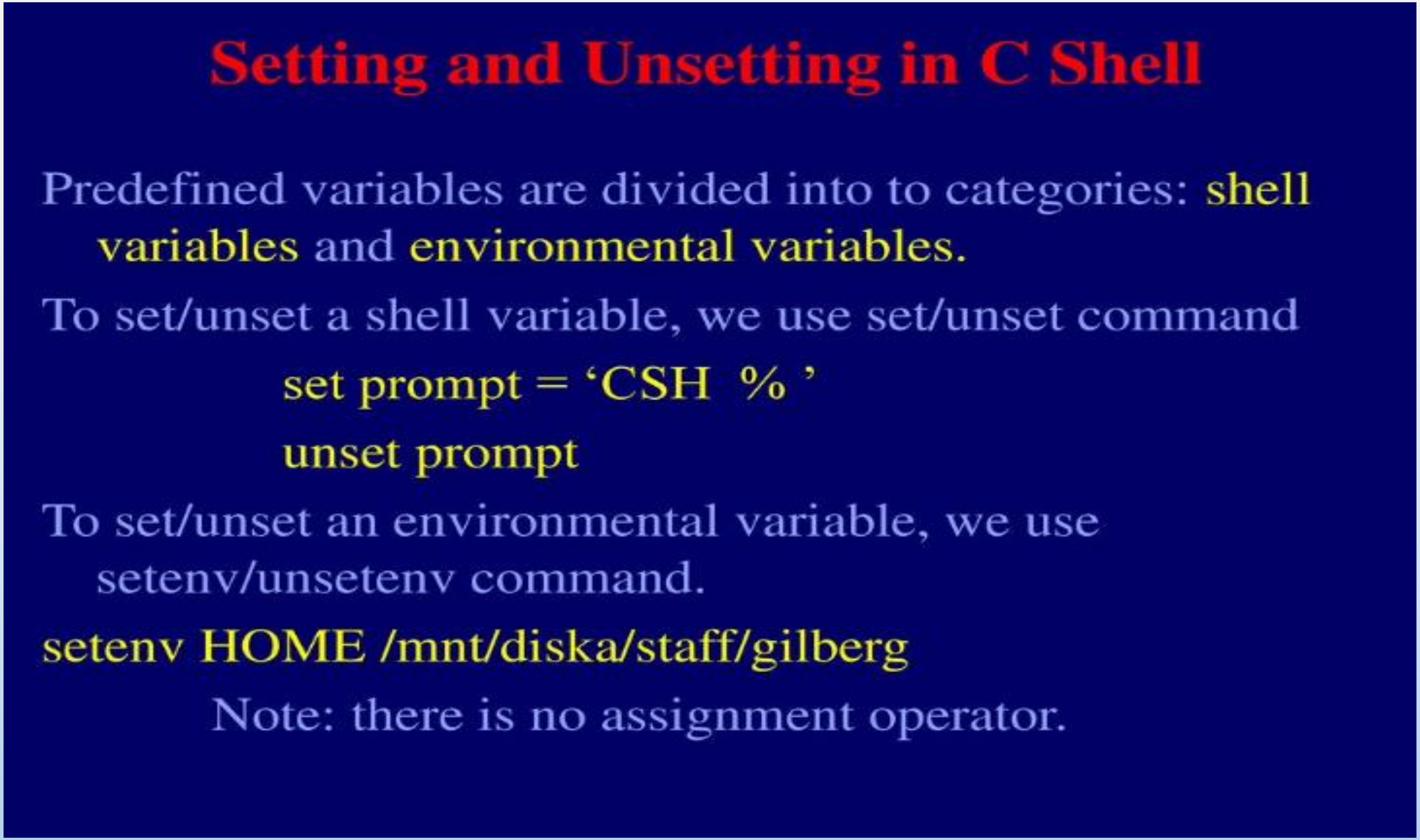
~/.login is the equivalent of user profile file

~/.cshrc is the environmental file

~/.logout is run when we log out of the C shell

Other system files

```
/etc/csh.cshrc
/etc/csh.login
/etc/csh.logout
```



Setting and Unsetting in C Shell

Predefined variables are divided into to categories: shell variables and environmental variables.

To set/unset a shell variable, we use set/unset command

```
set prompt = ‘CSH  % ’
unset prompt
```

To set/unset an environmental variable, we use setenv/unsetenv command.

```
setenv HOME /mnt/diska/staff/gilberg
```

Note: there is no assignment operator.

Thank You