



# More on Unix Shell

---



# Shell Operations

---

- When a shell is invoked automatically during a login (or manually from a keyboard or script), it follows a preset sequence:
  1. It reads a special start-up file, typically located in the user's home directory (e.g. `~/.cshrc`), that contains some initialization information. Each shell's start-up sequence is different.
  2. It displays a prompt and waits for a user command.
  3. If the user enters a Control-D character on a line of its own, this is interpreted by the shell as meaning "end of input" and causes the shell to terminate; otherwise, the shell executes the user's command and returns to step 2.



## Shell Operations (con't)

---

- Commands range from simple utility invocations, such as

```
sepc92: > ls
```

to complex-looking pipeline sequences,  
such as

```
sepc92: > ps -ef | sort | ul -tdumb | lp
```



## Shell Operations (con't)

---

- If you ever need to enter a command that is longer than a line on your terminal, you may terminate a portion of the command with a backslash (\) character, and the shell then allows you to continue the command on the next line:

*sepc92: > echo this is a very long shell command and  
needs to \*

*be extended with the line continuation character. Note \  
that a single command may be extended for several lines.*

this is a very long shell command and needs to be  
extended with the line continuation character. Note that a  
single command may be extended for several lines.

sepc92: > \_



# Executable Files Versus Built-in Commands

---

- Most UNIX commands invoke utility programs that are stored in the directory hierarchy. Utilities are stored in files that have execute permission. For example, when you type  
sepc92: > ls  
the shell locates the executable program called "ls," which is typically found in the *"/bin"* directory, and executes it.
- In addition to its ability to locate and execute utilities, the shell contains several built-in commands, such as *echo* and *cd*, which it recognizes and executes internally.



# Input, Output, And Error Channels

---

- In the example of the **date** command, the output was written to the terminal.
- In UNIX there are three default I/O channels that are always assumed to be active for every command or program:
  - Standard input, known as "stdin" whereby a program expects to find input
  - Standard output, known as "stdout" whereby a program writes its output by default
  - Standard error, known as "stderr" whereby a program writes error messages



# Input, Output, And Error Channels (con't)

---

- By default, the *standard input* of a process is the keyboard and the *standard output* and *standard error* are the screen.
- The default I/O channels can be easily changed on the command line by using “redirection.”



# Metacharacters

---

- Some characters receive special processing by the shell and are known as *metacharacters*.
- All four shells share a core set of common metacharacters, whose meanings are shown in the table.
- When you enter a command, the shell scans it for metacharacters and processes them specially. When all metacharacters have been processed, the command is finally executed. To turn off the special meaning of a metacharacter, precede it with a \ character.






# Metacharacters

---

- Metacharacters can be listed by using the **stty** utility with the -a (all) option. Here's an example:

```
sepc92: > stty -a          ...obtain a list of terminal metacharacters
speed 9600 baud;
rows = 39 columns = 142; ypixels = 0 xpixels = 0
intr = ^c; quit = ^\; erase = ^?; kill = ^u;
eof = ^d; eol = <undef>; eol2 = <undef>; swtch = <undef>;
start = ^q; stop = ^s; susp = ^z; dsusp = ^y;
rprnt = ^r; flush = ^o; werase = ^w; lnext = ^v;
-parenb -parodd cs8 -cstopb -hupcl cread -clocal -loblk -parext
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl -iuclic
ixon -ixany -ixoff imaxbel
isig icanon -xcase echo echoe echok -echonl -noflsh
-tostop echoctl -echoprt echoke -defecho -flusho -pendin iexten
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel -tabs
sepc92: > _
```

- The ^ in front of each letter means that the *Control* key must be pressed at the same time as the letter.



Symbol	Meaning
>	Output redirection; writes standard output to a file.
>>	Output redirection; appends standard output to a file.
<	Input redirection; reads standard input from a file.
*	File substitution wildcard; matches zero or more characters.
?	File substitution wildcard; matches any single character.
[...]	File substitution wildcard; matches any character between brackets.
`command`	Command substitution; replaced by the output from <i>command</i> .
	Pipe symbol; sends the output of one process to the input of another.
;	Used to sequence commands.
	Conditional execution; executes a command if the previous one failed.
&&	Conditional execution; executes a command if the previous one succeeded.
(...)	Groups commands.
&	Runs a command in the background.
#	All characters that follow, up to a newline, are ignored by the shell and programs (i.e., signifies a comment).
\$	Expands the value of a variable.
\	Prevents special interpretation of the next character.
<< tok	Input redirection; reads standard input from script, up to tok.



# Redirection

---

- The shell redirection facility allows you to do the following:
  - store the output of a process to a file (*output redirection*)
  - use the contents of a file as input to a process (*input redirection*)



# Output Redirection

---

- Output redirection is handy because it allows you to save a process' output into a file so it can be listed, printed, edited, or used as input to a future process. To redirect output, use either the `>` or `>>` metacharacter. The sequence

sepc92: `> command > fileName`

sends the standard output of *command* to the file with name *fileName*. The shell creates the file with name *fileName* if it doesn't already exist or overwrites its previous contents if it already exists.



# Output Redirection

---

- If the file already exists and doesn't have write permission an error occurs.

- The following is an example:

```
sepc92: > ls > files.txt
```

...create a text file  
files.txt storing the  
result of ls

```
sepc92: > cat files.txt  
program1.c    program1.h
```

...look at its contents.  
*sortfunc.c*



## Output Redirection (con't)

---

- In the next example, I created a file called “alice.txt” by redirecting the output of the **cat** utility. Without parameters, **cat** simply copies its standard input which in this case is the keyboard—to its standard output:

```
sepc92: > cat > alice.txt          ...create a text file.
```

```
In my dreams that fill the night,  
I see your eyes,
```

```
^D                                ...end-of -input.
```

```
sepc92: > cat alice.txt            ...look at its contents.
```

```
In my dreams that fill the night,  
I see your eyes,
```

```
sepc92: > _
```



# Output Redirection (con't)

---

- The sequence

sepc92: > command >> fileName

appends the standard output of *command* to the file with name *fileName*. The shell creates the file with name *fileName* if it doesn't already exist. In the following example, I appended some text to the existing "alice.txt" file:

```
sepc92: > cat >> alice.txt
And I fall into them,
Like Alice fell into Wonderland.
```

...append to the file.

```
^D
```

...end-of-input.

```
sepc92: > cat alice.txt
In my dreams that fill the night,
I see your eyes,
And I fall into them,
Like Alice fell into Wonderland.
sepc92: > _
```

...look at the new contents.



# Metacharacters (con't)

---

- Here's an example:

sepc92: > *echo hi > file*

...store output of echo in file.

sepc92: > *cat file*

...look at the contents of "file."

hi

sepc92: > *echo hi \> file*

...inhibit > metacharacter

hi > file

...> is treated like other characters

sepc92: > \_

...and output comes to terminal  
instead





# Input Redirection

---

- Input redirection is useful because it allows you to prepare a process' input and store in a file for later use. To redirect input, use either the `<` or `<<` metacharacter. The sequence

`sepc92: > command < fileName`

executes *command*, using the contents of the file *fileName* as its standard input. If the file doesn't exist or doesn't have read permission, an error occurs.



# Input Redirection (con't)

---

- sepc92: > *wc* ... counting statistics  
This is the first line.  
This is the second line.  
<CTRL-D>  
2 10 49
- Suppose that the text file “test.txt” contains the two lines above.
- sepc92: > *wc < test.txt*  
sepc92: > 2 10 49



# Pipes

---

- The shell allows you to use the standard output of one process as the standard input of another process by connecting the processes together via the pipe (`|`) Metacharacter. The sequence

sepc92: > *command1* | *command2*

causes the standard output of *command1* to “flow through” to the standard in *command2*. Any number of commands may be connected by pipes. A sequence of commands chained together in this way is called a *pipeline*.



## Pipes (con't)

---

- Pipelines support one basic UNIX philosophy, which is that large problems can often be solved by ad smaller processes, each performed by a relatively small, reusable utility.
- The standard error channel is not piped through a standard pipeline, although some shells support this capability. In the following example, I piped the output of the **ls** utility to the input of the **wc** utility to count the number of files in the current directory:

```
sepc92: > ls ...list the current directory.
```

```
a.c      b.c      cc.c  dir1  dir2
```

```
sepc92: > ls | wc -w ...count the entries.
```

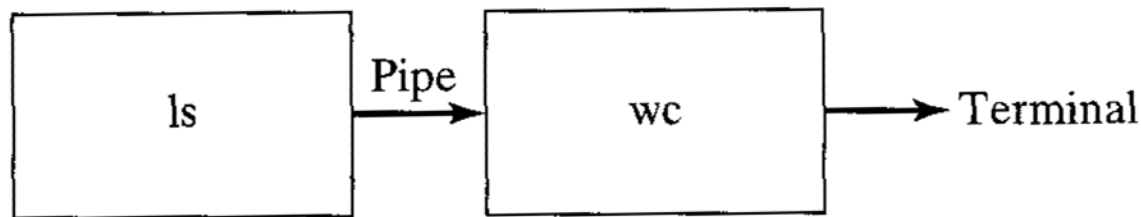
```
5
```

```
sepc92: > _
```

# Pipes (con't)

---

- An illustration of a pipeline



- In the next example, I piped the contents of the “who” utility into the **awk** utility to extract the first field of each line. The output of **awk** was then piped to the sort utility, which sorted the lines alphabetically. The result was a sorted list of every user on the system.



## Pipes (con't)

---

- The commands are as follows:

```
sepc92: > who
```

```
swchan pts/0 2020-10-21 14:34 (vpn.se.cuhk.edu.hk)
```

```
tkwong pts/1 2020-10-21 15:10 (pn-204.itsc.cuhk.edu.hk)
```

```
ajlam pts/2 2020-10-21 15:49 (sepc204.se.cuhk.edu.hk)
```

- sepc92: > *who | awk -F " " '{ print \$1 }'*

```
swchan
```

```
tkwong
```

```
ajlam
```

- sepc92: > *who | awk -F " " '{ print \$1 }' | sort*

```
ajlam
```

```
swchan
```

```
tkwong
```



## Special Local Variables of a Shell

---

- Every shell has a set of predefined *environment variables* that have special meanings to the shell

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 username
\$SHELL	the full pathname of your login shell
\$TERM	the type of your terminal



# Determining Your Terminal's Type: `tset`

---

- Several UNIX utilities, including the two standard editors **vi (or vim)** and **emacs**, need “know” what kind of terminal you’re using so that they can control the screen correctly.
- The type of your terminal is stored by your shell in the TERM environment variable.
- You may think of environment variables as being global variables for the shell and they can hold strings.





# Determining Your Terminal's Type:

## **tset (con't)**

---

- Before **vi**, **vim** or **emacs** can work correctly, your shell's TERM environment variable must be set to your terminal type. Common settings for this variable include "vt100" and "vt52." There are several ways to set TERM:
- Your shell start-up file can set TERM directly by containing a line of the form 'setenv TERM vt100' (C shell) or 'TERM=vt100; export TERM' (for Bourne, Korn, and Bash shells). This method of setting TERM is practical only if you know the type of your terminal in advance and you always log into the same terminal.
- Your shell start-up file can invoke the **tset** utility, which looks at the communications port that you're connected to and then set TERM accordingly. Consult the online manual for **tset**.



# Setting \$TERM environment variable

---

- You can manually set TERM by:  
sepc92: > *setenv TERM vt100*
- To display the content of an environment variable:  
sepc92: > *echo \$TERM*  
vt100

# Creating environment variables

---

- You can create your own variable  
sepc92: > *setenv TEST content1*
- To display the content of an environment variable:  
sepc92: > *echo \$TEST*  
content1
- To display the content of all environment variables:  
sepc92: > *setenv*  
USER=wlam  
LOGNAME=wlam  
HOME=/sac/lec/wlam  
DISPLAY=localhost:10.0  
:  
:  
TEST=content1



# More on environment variables

---

- You can remove an environment variable:  
sepc92: > *unsetenv TEST*
- Generally the user-defined environment variables will vanish when you log out.
- In fact, the system administrator has defined some environment variables in the startup file `.cshrc`
  - Then when you login, those environment variables will exist in your shell.



# How a Shell Finds A Command

---

- When a shell processes a command, it first checks to see whether the command is a built-in; if it is, the shell executes it directly. *echo* is an example of a built-in shell command:

```
sepc92: > echo some information
some information
sepc92: > _
```



# How a Shell Finds A Command

- If the command in question isn't a built-in command, then
  - the shell checks whether it begins with a / character.
    - If it does, the shell assumes that the first token is the absolute pathname of a command and tries to execute the file with the stated name.
    - If the file doesn't exist or isn't an executable file, an error occurs:

sepc92: > */usr/ucb/l*s                   ... full pathname of the *ls* utility  
abc.txt           program.cpp

sepc92: > */usr/local/bin/n*sx           ... a non-existent filename  
*/usr/local/bin/n*sx: not found

sepc92: > */etc/passwd*                   ... the name of the password file  
*/etc/passwd*: Permission denied       ...it's not executable

- If it is not an absolute pathname, the shell checks whether it begins with a path-related character such as ~ or . or .. and tries to execute the file with the stated full pathname similar to the above.

sepc92: > *~david/demo/g*ame   ... execute the *game* program under  
                                  ... the home directory of the user *david*



# How a Shell Finds A Command : \$PATH

---

- If the command in question isn't a built-in command or a full pathname, the shell searches the directories whose names are stored in the PATH environment variable.
- To display the content of an environment variable:  

```
sepc92: > echo $PATH
```

```
/sbin:/usr/ucb:/usr/local/bin
```
- Each directory in the PATH environment variable is searched (from left to right) for an executable file matching the command name.
  - If a match is found, the file is executed.
  - If a match isn't found in any of the directories, then it looks for the current directory. If it still fails to find the file, or if the file that matches is not executable, an error occurs.

# How a Shell Finds A Command : \$PATH

- Here is an example:

```
sepc92: > echo $PATH  
/sbin:/usr/ucb:/usr/local/bin    ...directories searched
```

```
sepc92: > ls  
abc.txt      program.cpp    ... /ls is located in "/usr/ucb/"  
... the result of executing ls
```

```
sepc92: > nsx  
nsx: not found _    ... not located anywhere
```

- If the PATH environment variable is not set or is equal to the empty string, then only the current directory is searched.
- As a result, to make sure that only the current directory is searched, one can start with metacharacters ./
- sepc92: > ./who ... only the current directory is searched  
... instead of the built-in Unix command





# Setting \$PATH

---

- The contents of the PATH variable may be changed thereby allowing you to tailor the search path to your needs.
- The original search path is usually initialized by the shell's start-up file (e.g. ~/.cshrc) and typically includes all of the standard UNIX directories that contain executable utilities.



## The *which* Utility

---

- The ***which*** utility can show the current path of a certain utility found by the shell using the steps discussed above

```
sepc92: > which ls  
/usr/ucb/ls
```