UNIT 4: Shell and Process

BCAC601: UNIX and Shell Programming

Dr. Tumpa Banerjee
Assistant Professor, Department of MCA

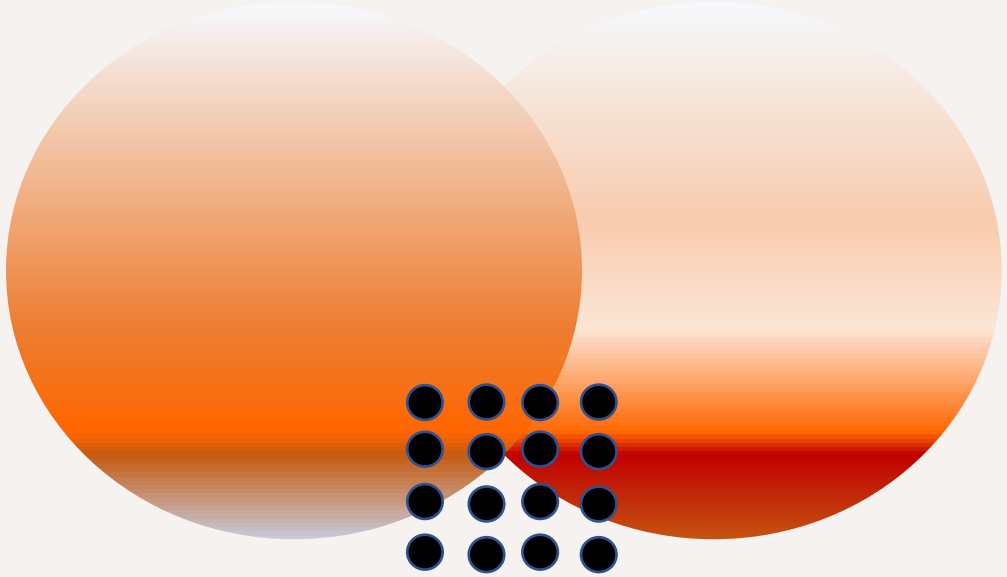

Table of Content

Shell

- The interface through which we control resources of Unix operating system is called shell.
- The shell provides many of the features that make the Unix system a uniquely powerful and flexible environment
- The shell allowed many features that allow you to customize your working environment.
- we can use shell variables and allies to simplify many tasks.
- We can use shell command language as a high-level programming language called script.

The Common Shell

- The original UNIX Shell, sh was written by Steve Bourne and as a result known as Bourne shell.
- The C shell, csh, was the first attempt to enhance the original Bourne shell. The csh introduced the concept of command history list, job control and aliases.
- The extended C shell, tcsh has replaced csh entirely on some version of Unix. it retains all features of csh and add comment line editing.
- The korn shell, ksh developed at AT&T Bell Lab by David Korn.

Interpretive Shell cycle

After logging in, the UNIX system takes the form of a dialog with the cell. The dialog follows these simple sequence, repeated over and over.

1. The shell prompt you when it is ready for input, waits for you to enter a command.
2. You enter a command by typing in a command line.
3. The shell processes your command line to determine what action to take and carries out the actions required.
4. The program is finished, the shell prompt you for input, beginning the cycle again.

Argument Expansion

- The shell also interprets and replace certain shortcuts before sending argument on to the command, this is called expansion.
- $\$\ mv\ dance\ \sim$
- The shell will replace the task with your home directory before sending the ~ argument to mv
- Several comments can be entered at once on line by separating them with semicolon.
- $\$\ date;\ ls\ -l$

Pattern Matching

- Wildcards can be used for pattern matching in filename.
- Wildcards can be used to specify a whole set of files at once, or to search for a file when you know only parts of its name.

$*html$	**Matches any filenames ending in html**
$note*$	Matches any filenames start in note
$*kill*$	Matches any filenames containing the string kill

Pattern Matching

- The standard UNIX system shell provides two other filename wildcards: ? And [...]
- The question mark matches the single character.
- Bracket match any one of a set of characters that they enclose.
- We can indicate a range or sequence of characters in brackets with a −.

$email?$	Matches any filename consisting of email followed by one character
$[Jj]mf$	Matches Jmf, jmf
$Output[a-c]$	Matches Outputa, Outputb, and Outputc

Standard Input and Output

- The output of a command can be send to your screen, stored in a file, or used as the input to another command.
- Most command accept input from your keyboard from a stored file or from output of another command.
- This flexible approach to input and output is based on UNIX system concepts of standard input or standard I/O.
- The command does not need to know which of these resource the input comes from or where the output goes. It is the shell that set up the connection according to the instruction.

Standard Input and Output

Symbol	Example	Function
|	$cmd1 \mid cmd2$	Run cmd1 and send output to cmd2
>	$cmd > file$	Send output of cmd to file
>>	$cmd \gg file$	Append output of cmd to file
<	$cmd < file$	Take input for cmd from file
2>	$cmd\ 2 > errorfile$	Send standard error to errorfile (ksh, bash)
2>&1	$cmd > msgs\ 2 > \&1$	Send both output and standard error to msgs(sh, ksh, and ksh)
/dev/tty	$(cmd\ > /dev/tty)$ $> \ \& error$	Redirect output to screen, error to error(csh, tcsh, and bash)
>&	$cmd > \& msgs$	Send both output and errors to msgs

Output redirection

- The right arrow redirection operator reads the output of a command to a file.
- $ls\ -l > filelist$
- Cozy is the sale to save the output of LS – Ill Command to filelist.
- If a file filelist exist in current directory it will override.
- The >> Operator appends data to a file without overwriting it.

Input Redirection from a file

- we can use left arrow(<) to redirect standard input.
- The left arrow(<) symbol tells the shell to interpret the file name that follows it as the standard input to a command.
- $\$ sort < source > dest$
- Sort command takes information from the source file, alphabetizes it, and outputs it in the dest.
- The order of the operator does not matter. It provides the same result.
- $\$sort > dest < source$

Standard input from keyboard

- You can also force commands to accept input from keyboard.
- The logical file name $dev/stdin$ refer to the standard input when used as an argument to a command, it causes the shell to send the input from the keyboard to the command in place of a file.
- $ sort /dev/stdin > output

Standard error

- Standard error provides a second logical channel that a program can use to communicate with you, separate from standard output.
- Standard era is normally used for displaying error message.
- $\$\ ls\ -l\ 2 > errorfile$

Using pipes

- The pipe symbol tells the shell to take the standard output of 1 command and use it as the standard input of another command.
- Using pipes to join individual commands together in a pipeline in an easy way to use a sequence of simple commands to carry out a complex task.
- $\$who\ |grep\ anik$

Using tee

- The tee command is named after tee joint in plumbing.
- A tee joint split and incoming stream of water into two separate streams.
- tee split its standard input into two or more output streams, and each is sent to standard output; the others are sent to the file you specify.
- $file\ *|tee\ filetypes$

$/dev/tty$

- The logical filename $/dev/tty$ refers to your current terminal.
- The command
- \$ $(ls\ -l > /dev/$tty$)\ 2 > errorfile$ causes the shell to redirect the standard output from ls to the screen.
- Once the standard output has been redirected to /dev/tty the error file will cause the shell to redirect the standard error to error file.

Shell Variables

- Variable Names: The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
- By convention, Unix shell variables will have their names in UPPERCASE.
- Variables are defined as follows : variable_name=variable_value
- To access the value stored in a variable, prefix its name with the dollar sign ($)

Shell Variables

- Read-only Variables
- Shell provides a way to mark variables as read-only by using the read-only command.
- After a variable is marked read-only, its value cannot be changed.
- PI=3.147
- readonly PI

Shell Variables

- Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks.
- Once you unset a variable, you cannot access the stored value in the variable.
- unset variable_name

Variable Types

- Local Variables − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- Environment Variables − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- Shell Variables − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Running command in Background

- When the shell runs a command, it is until the common is completed before it resumes its dialogue with you.
- During this time user cannot communicate with the shell. You may want to start the command and then run another immediately without waiting for the past to finish.
- An & symbol at the end of a common line directs the cell to execute the command in the background.
- $\$ \mathit{command}\ \&$

Running command in Background

- When a command is run in the background the system will continue to process eat while the cell from you for another command.
- Shell acknowledges your background command with a message that contains two numbers then the first number which is enclosed in the bracket is the job ID and this comment the other number is the process ID

Standard input/output and background jobs

- When you run a command in the background, the starts and prompts another command.
- It disconnects the standard input of the background command from your keyboard, but it does not automatically disconnect the standard output from your screen.
- When you run a command in the background you have to redirect its output to a file.
- $\$ ./prog1.sh > filename$

Standard Error

- When you run a command in the background, you should also consider whether you want to redirect your standard error.
- The find command can be used to search through an entire directory structure for the files with a particular name.
- It'll take lots of time to search. You can run it in the background.
- Find may generate messages to the standard error.
- To discard standard error entirely, redirecting it to $/dev/null$, which will cause it to finish.

$/dev/null$

- $\$\ troff\ big\_file\ > output\ 2 > /dev/null$
- /dev/null is a device that does nothing with information sent to it.
- It is like a black hole into which input vanishes.
- Sending output to /dev/null is a handy way to get rid of it

Job Control

- You can have multiple jobs in the background but only one job in the foreground.
- The job control commands allowed you to terminate a background job(kill it), suspend a background job temporarily(stop it), resume a suspended job in the background, move a background job in the foreground, and suspend a foreground job.
- The jobs command displays a list of all your jobs such as
- $ $jobs$
- [1] Running

Job Control

- The output shows your current foreground and background jobs as well as jobs that are stopped or suspended.

Command	Effect
$jobs$	List all jobs
$ctrl + z$	suspend current foreground process
$bg\ \%n$	resume stopped job in background
$fg\ \%n$	resume job in the foreground
$stop\ \%n$	suspend background job
$kill\ \%n$	terminate job

Job Control

- The command will cause a job to run in the background
- $ bg %1
- % sign introduce job identifier.
- You can terminate any of your background or suspended job with the kill command.
- $ $kill$ %2
- In addition to the job identifier you can use the name of the command to tell the sale which sale to kill.
- $ $kill\ troff$

Job Control

- The following command to halt the execution of a background job.
- $ $stop \ \%find$
- The following command will resume execution of the foreground job.
- $ $fg \ \%find$

Process

- A process is a program in execution
- A program can create new processes for a given program there may be one or more processes in execution.
- A program can create new processes for a given program there may be one or more processes in execution.

Process

- At the lowest level, a process is created by the fork system call.
- A system call is a routine that causes the kernel to provide some service for a program.
- A fork creates a separate but almost identical running process.
- The process that makes the fork system call is called the parent process; the process that is created by the fork is called the child process.
- The two processes have the same environment, the same signal handling settings, the same group and user IDs, and the same scheduler class and priority but different process ID numbers.

Process Creation

- There are three distinct phases in the creation of a process and uses three important system calls or functions- fork, exec, wait.
- $fork$: All process in Unix is created with the fork system call, which creates a copy of the process that invokes it.
- $exec$: the child process call $exec$ system call to run the program.
- $wait$: the parent then executes the wait system call to wait for the child process to complete.

Parent or Child Process

- When you type a command on your UNIX system the shell handles its execution.
- If the command is built in command, known by the shell, it is executed internally without creating a new process.
- If the command is not a built in, the shell treat it as an executable file.
- The current shell uses the system call $fork$ and create a child process, which executes the command.

Parent or Child Process

- The parent process, the shell, waits until the child either complete execution or dies.
- When the shell create the child process it executes a $wait$ system call.
- These suspends the operation of the parent shell until it received a signal from kernel indicating the death of the child process.
- At that point parent process wake up and look for a new command.

The ps command

- The ps command list all of the active processes running on the machine.
- If you use ps without any option, information is printed about the process associated with your terminal.

ps Options

- ps is a highly variant command; it actual output varies across different UNIX flavors.

Options	Significance
$-f$	Full listing showing the PPID of each process
$-e\ or\ -A$	All processes including user and system process
$-u\ usr$	Process of user usr only
$-a$	Process of all users excluding system process
$-l$	Long listing showing memory related information
$-t\ term$	Process running on terminal $term$

Process Scheduling

- You can specify when comments should be executed by using at command, which read commands from its standard input and schedule them for execution.
- Normally standard output and error are mailed to you unless you redirect them elsewhere.
- Mix accept several ways of indicating the time.
- Examples of time and date are
- at $500\ Jan\ 18$
- at $noon$
- at $5{:}00\ PM\ tomorrow$

Process Scheduling

- The at command is handy for sending yourself reminder for important scheduled event.
- $\$\ at\ 6{:}00\ PM\ Friday$
- e$cho\ Don't\ forget\ my\ meeting$
- Will mail you the reminder early Friday morning
- You can redirect standard output back to your terminal and use at to interfere with a reminder.
- The $-f$ option to at allows you to run a sequence of command content in a file.

The corn facility

- The corn facility is a system demand that executes comments at specific times.
- It similar in some aspect to the at command, but is much useful for repetitive execution of a process.
- The command and scheduled information are kept in the directory /var/spool/corn/corntabs or /usr/spool/corn/corntabs
- Each user who is entitled to directly schedule command with code has a orntab file.
- corn wakes up and execute any job that are scheduled for that minute.

Process Priorities

- Process on a Unix system are sequentially assigned resources for execution.
- The kernel assigns the cpu to a process for a time slice; when the time has elapsed, the process is placed in one of several priority queues.
- The execution is scheduled depends on the priority assigned to the processes.
- System processes have higher priority then user processes.
- User process priorities depend on the amount of cpu time they have used.
- Processes that have larger amounts of cpu time are given lower priorities those using little cpu time are given higher priority.

The $nice$ command

- The $nice$ command allows a user to execute a command with a lower than normal priority.
- The process that is using the $nice$ command and the command being run must both below to the time scheduling class.
- The priority of a process is a function of its priority index and its nice value.
- $priority = priority\ index + nice\ value$
- you can decrease the priority of a command by using $nice$ to reduce nice value.
- If you reduce the priority of your command it uses less cpu time and runs slower.

The $nice$ command

- The reduction in $nice$ value can be specified as an increment to nice.
- valid values are from -1 to -19; if no increment is specified a default value of 10 is assumed.
- You do this by preceding a normal command with the nice command
- $\$\ nice\ proofit$
- Will run the proofit command with a priority value reduced by the default value of 10 unit.
- $\$\ nice\ -19\ proofit$

The $nice$ command

- A child process inherit the $nice$ value of its parent.
- Running a command in the background does not lower its priority.
- If you wish to run command in the background and at a lower priority place the command in a sequence in a file(script) and issue the command.
- $\$\ nice\ -10\ scrtipt\ \&$
- The priority of a command can be increased by the superuser.
- A higher nice value is assigned by using a double minus sign.
- $\$nice\ --19\ draft$

The sleep command

- The sleep command does nothing for a specific time.
- You can have a shell script suspended operation for a period by using sleep.
- The command $sleep\ time$ included in a script will delay for $time$ second.
- $\$\ (sleep\ 3600; who \gg log)\ \&$
- it creates a process in the background that sleeps for 3600 sec and then wake up, run the who command and place its output in a file name log.

The wait command

- When a shell uses the system call $fork$ to create a child process, it suspends operation and waits until the child process terminates.
- when a job is run in the background, the shell continues to operate while other jobs are being run.
- it is important in shell script to be able to run simultaneous processes and wait for their conclusion before proceeding with other commands.
- The wait command allows this degree of scheduling control within shell script.

The wait command

```
Command1 >> file1 &
Command2 >> file2
Wait
Sort file1 file2
```

- Rum the two commands Simultaneously in the background. Eat weights until both background process terminates and then eat sorts the two output files.