Lecture-3:
Shell

Tongping Liu
Tongping.Liu@utsa.edu

Administration Related

Recitation time:

- Using your registered time slot

Project Assignments

- Submit script (Told next recitation time)
- Auto-grading system

Announcement

Two Forums:

- Student Internet Café
- Help Forum

Slides:

- Available at blackboard
- Try to put before classes

Utilities: not all of them are not tested.

What we learned last time

Overview

Utilities

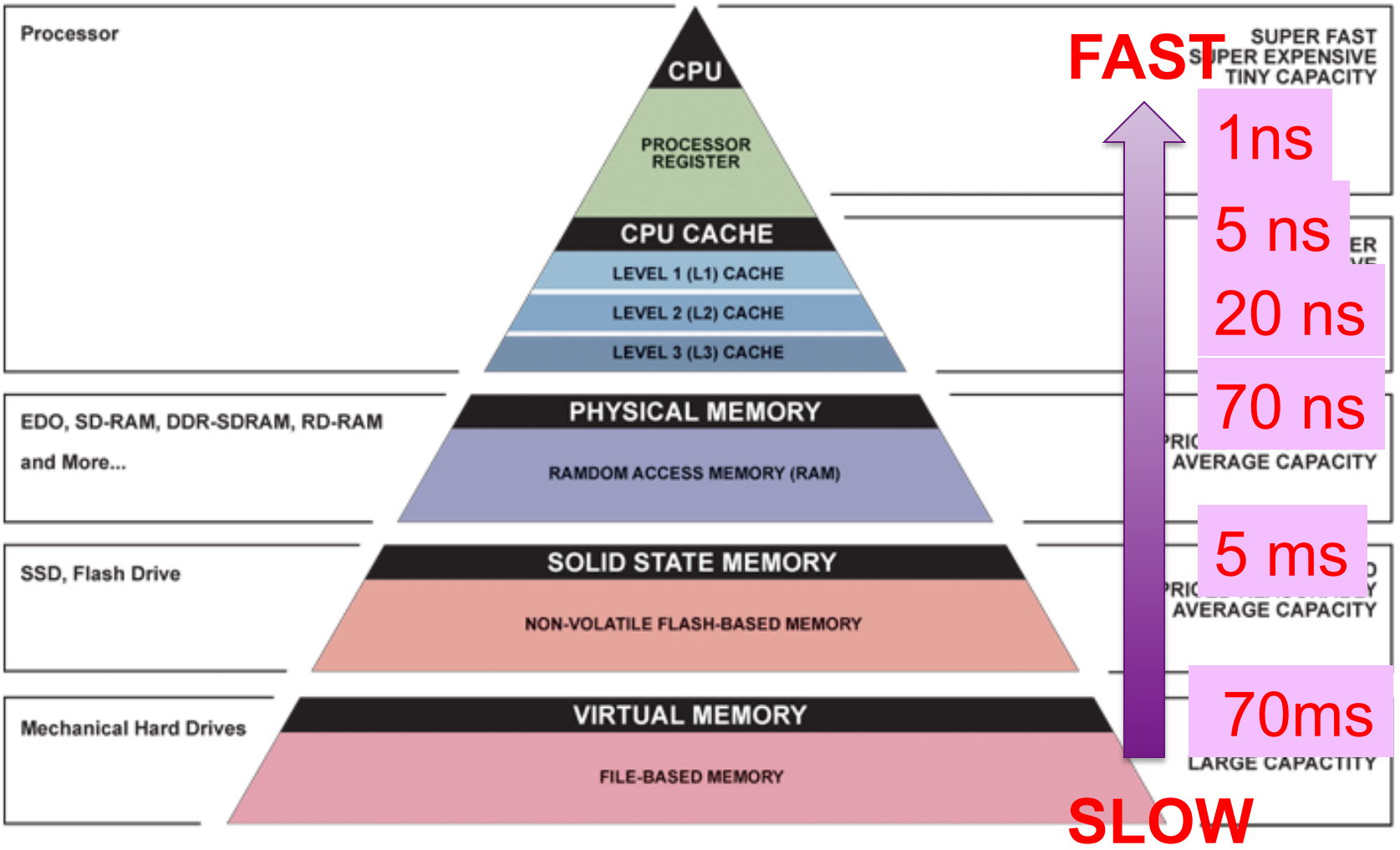
Redirection

Combining multiple commands

Other useful utilities

Text Editor

The Memory Hierarchy



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

User Space vs Kernel Space

Safety Reason:

- This separation serves to protect data and functionality from faults (by improving fault tolerance) and malicious behaviour (by providing computer security).

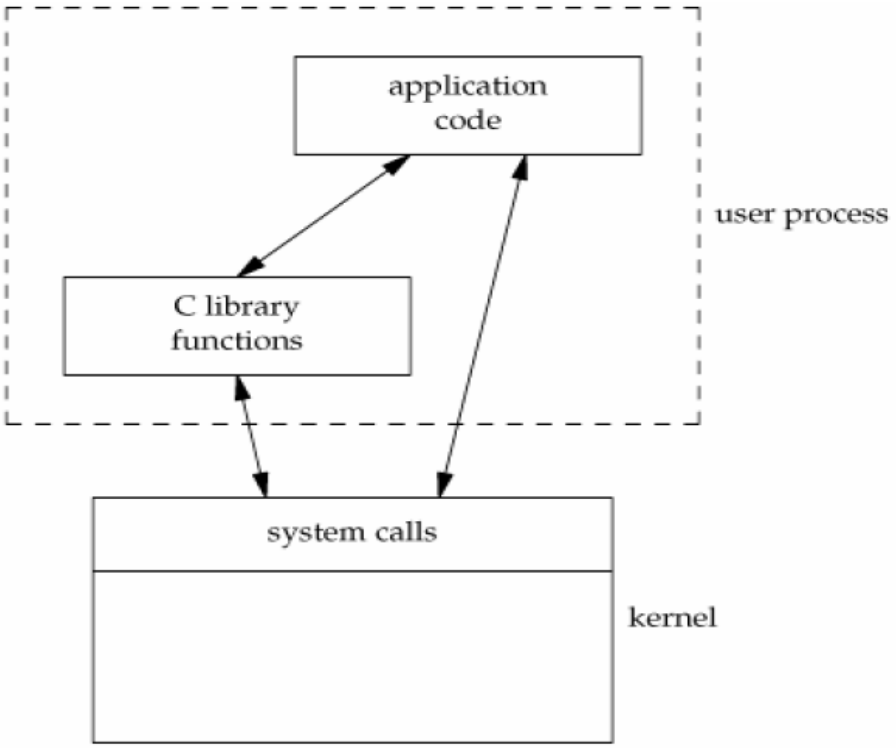
User space:

- Refers to all code which runs outside the operating system's kernel.
- User space usually refers to the various programs and libraries that interacting with the kernel.

Kernel space:

- Reserved for running privileged kernel, kernel extensions, and most device drivers.

System Calls



- A system call is how a program requests a service from an operating system's kernel.
- System calls provide an essential interface between a process and the operating system.

Utility

Definition:

- Some software tools that are in the system and help analyze, manage, configure, optimize or maintain a computer

Reason:

- DON'T need to write a program to do simple task. For example, we want to know the content of a file.

Check the command of a utility:

- man CMD # Check the manual of a command CMD

Directory

Home directory: A home directory is a directory on a multi-user system, containing files for a given user of the system.

Root directory: the first or top-most directory in a hierarchy(/).

Working directory: the directory associated with a particular process.

Absolute path: An absolute path is defined as the specifying the location of a file or directory from the root directory

Relative path: Relative path is defined as path related to the present working directory(pwd).

- **/home/tongpingliu** (Absolute path; Home path)
- **../mydir** (Relative path)

Special symbols

Asterisk (*) - a wildcard matching folder or file names

- **cp notes*.txt <dir> # Copy notes1.txt, notes-xyz.txt to a <dir>**

Dot (.) - represents the current directory

```
elk01> pwd
/home/twood/folder-1

elk01> ls ..
folder-1/ folder-2/

elk01> cp * /tmp
```

list contents of the parent directory

copy all files into folder

Standard input/output

Stdin: standard input. Fd: 0

Stdout: standard output. Fd: 1

Stderr: standard error. Fd: 2

Output to a file

"Redirect" output of a command to a file

- Useful for commands that produce many lines of output
- Save results for later, or to use with another command

Syntax: [command] **>** [filename]

```
> sort days.txt > sorted.txt

> head -n 3 sorted.txt
Friday
Monday
Saturday
```

Warning! This will REPLACE any file with the same name

To APPEND to a file, use >>

- [command2] **>>** [filename]

Stderr to a file

grep da * 2> grep-errors.txt

Check whether there is an error of grep command.
For this output, this file has no content.

grep da * abcd 2> grep-errors.txt

$ cat grep-errors.txt
grep: abcd: No such file or directory

Only standard errors will be output to a file. Those normal output won't be redirected to this file!!!

Redirect stdout and stderr to a file

./run.sh 2>&1 > mylog

Redirects stderr to stdout and then saves all of these to a file mylog.
However, we can't see those results on the screen.

./run.sh 2>&1 | tee mylog

Redirects stderr to stdout and then saves all of these to a file mylog.
Also, we can see those results on the screen.

tee: copies standard input to standard output, making a copy in zero or more files. The output is unbuffered.

Command: tee [file ...]

| : pipe to connect two commands

Pipe – combining multiple commands

Pipe allow you to combine multiple commands

Syntax: [command  **[command 2]**

Example:

- **Sort a file, then print the top 3 entries**

```
> sort days.txt | head -n 3
Friday
Monday
Saturday
```

Output of the first command is the input of the second command

Internal of the pipe

command_a [arguments] | command_b [arguments]

REDIRECTION ?

command_a [arguments] > temp
command_b [arguments] < temp
rm temp

xargs

Execute command lines from standard input.

Target: delete specific files having the "abc" inside

(1) rm `find . –name "*abc*"`

Fails with "Argument list too long" if too many files

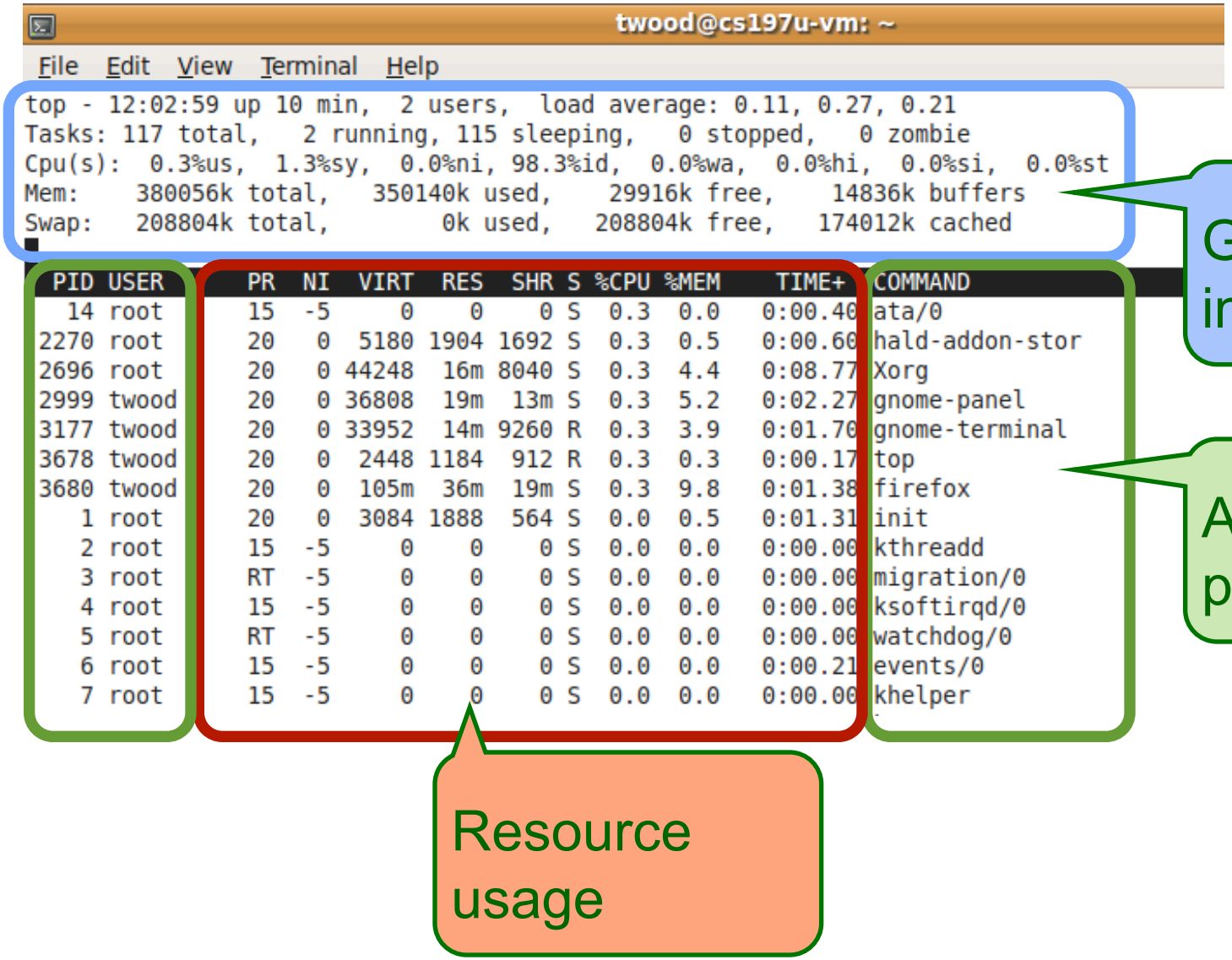
(2) find . –name "*abc*" | xargs rm -f

find utility feeds the input of **xargs** with a long list of file names. xargs then splits this list into sublists and calls rm once for every sublist. However, it does not correctly handle files or directories with a space in the name.

(3) find . –name "*abc*" –print0 | xargs -0 rm -f

Delimit results with NUL (\0) characters (by supplying -print0 to find), and to tell xargs to split the input on NUL characters as well (-0).

`top` - process and system info



Utilities review

Utilities	Description
`cp, mv, rm`	copy, move, and delete files
`cat`	print files to console
`head, tail`	print tops and bottoms of files
`sort`	sort files
`uniq`	Remove duplicate adjacent lines
`less, more`	view long files
`man`	provide help about commands
`>, >>`	Write command output to a file
`|`	Send command output to another command

More utilities: http://en.wikipedia.org/wiki/List_of_Unix_utilities

Today:

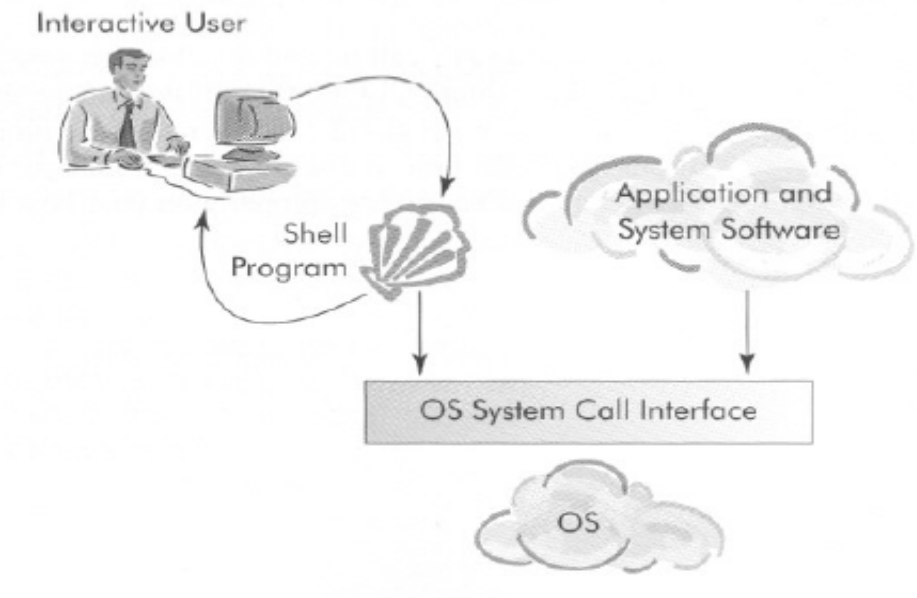
Shell

BASH script

Unix Shell

As a commander interpreter

- Provides the user interface to many GNU utilities



As a programming language

- Allows utilities to be combined. For example, files containing commands can be created, and become commands themselves.

Common Unix Shells

Name	Path	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Bourne shell	/bin/sh	•	link to bash	link to bash	•
Bourne-again shell	/bin/bash	optional	•	•	•
C shell	/bin/csh	link to tcsh	link to tcsh	link to tcsh	•
Korn shell	/bin/ksh				•
TENEX C shell	/bin/tcsh	•	•	•	•

BASH

What SHELL is used in elk01 machine?

echo $0

echo $SHELL

```
➢ Echo $0
-tcsh
➢ Echo $SHELL
/bin/tcsh
```

Change the shell to /usr/local/bin/bash

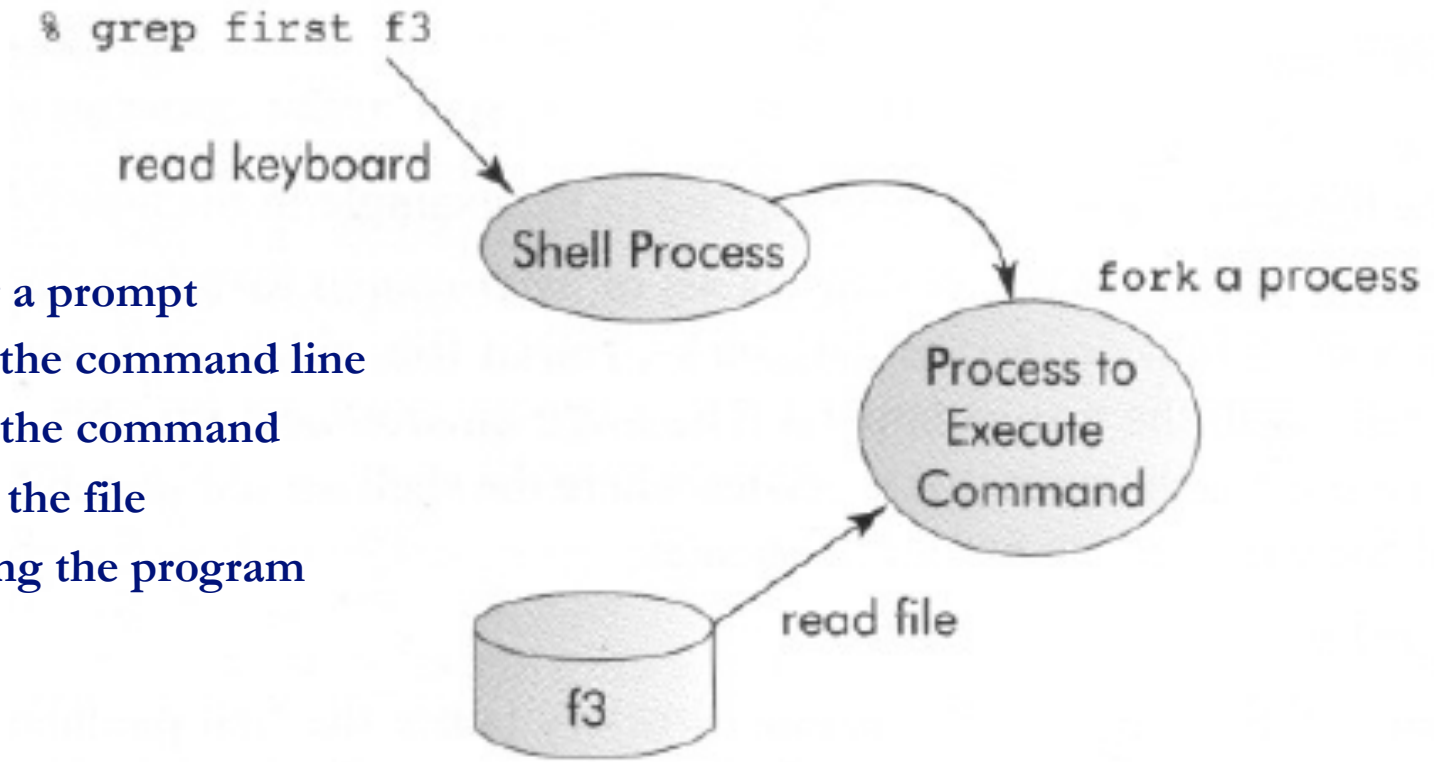
```
➢ elk01:~> cat ~/.cshrc
# New files are created without
group/other permissions

umask 077

setenv SHELL /usr/local/bin/bash
exec /bin/bash --login
```

Unix Shell

The shell isolates itself from program failures by creating a child process to execute each command/program

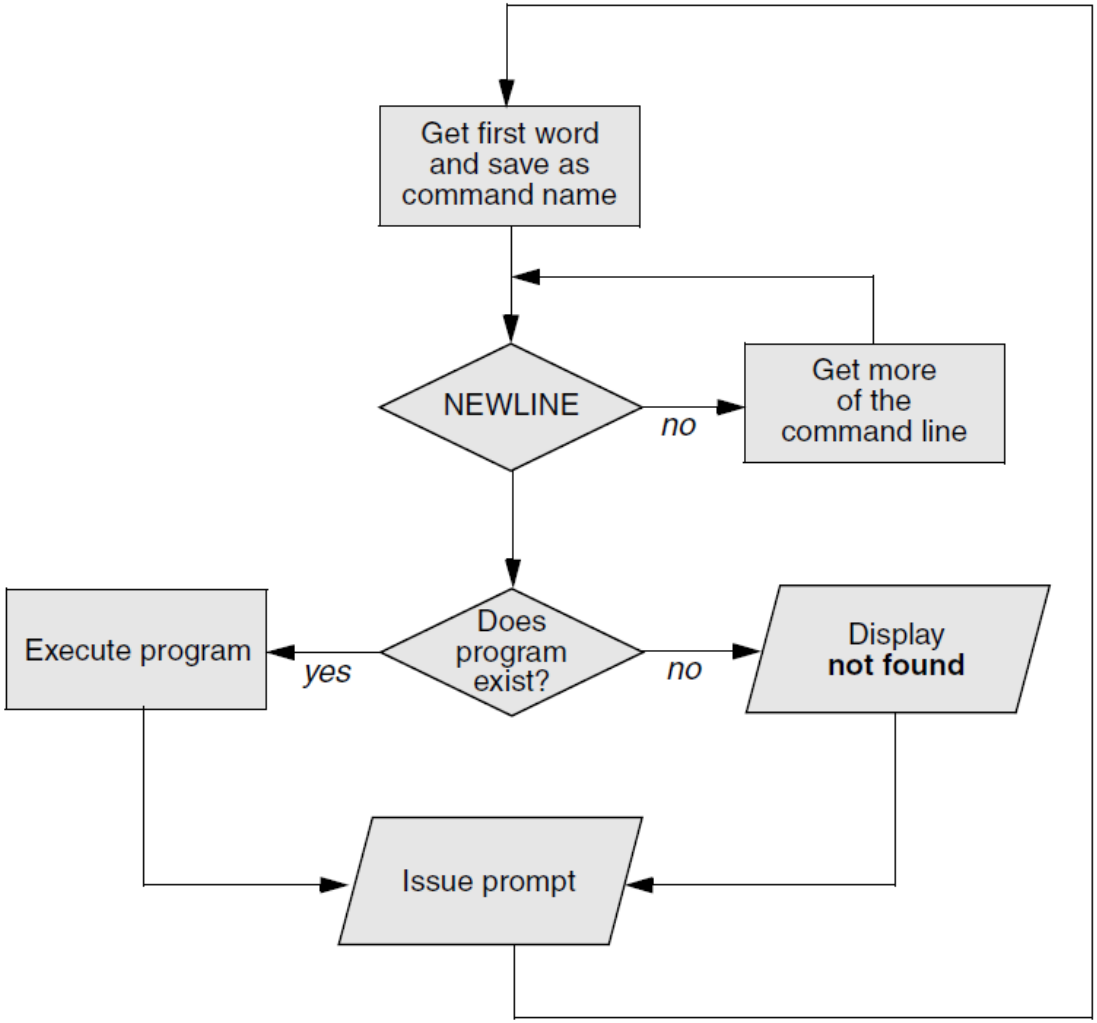


- Printing a prompt
- Getting the command line
- Parsing the command
- Finding the file
- Executing the program

Why creating a child process to execute a command, instead of by itself?

Protect itself from any fatal errors that might arise during execution

Processing the Command Line



Where does the Shell check for the existence of the given command ?

↓

Depends on whether Absolute or Relative Path is provided. (PATH) environment

ls → /bin/ls

./myprogram

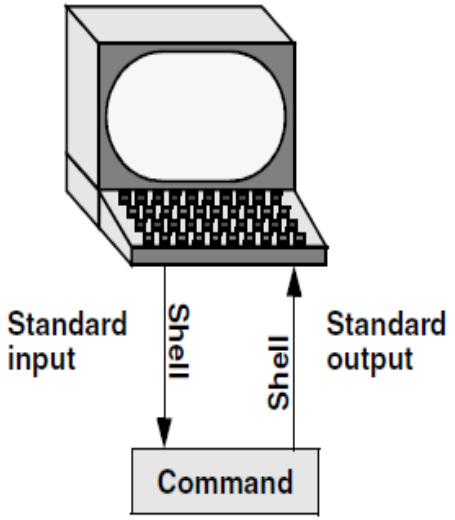
System Calls for Shell Design

A stripped down shell:

```
while (TRUE) {                                  /* repeat forever */
  type_prompt( );                               /* display prompt */
  read_command (command, parameters)           /* input from terminal */
  if (fork() != 0) {                            /* fork child process */
          /* Parent code here ……*/
          waitpid( -1, &status, 0);             /* wait for child to exit */
  } else {
          /* Child code here */
          execve (command, parameters, 0);      /* exec commands */
  }
}
```

Input and Output

At login, the shell directs the default standard input and output.



By default, standard input comes from the keyboard and standard output goes to the screen

Today

Shell

BASH script

Motivation of a script

Interactive mode:

- User types a command at a time, then gets an immediate execution and feedback.

Inconveniency: It might take a long time to finish.
Complicatedness: some temporary results.
Debuggability: hard to debug

Batch mode:

- Put all commands or related parts in a file.
- Then run all of them in a batch mode.

Initial line and permission of a script

1. **!/bin/sh : Shebang/hashbang**

http://en.wikipedia.org/wiki/Shebang_(Unix)

#!interpreter [optional-arg]

When a shell executes the script, it will use the specified interpreter.
Then will pass ”/path/to/script" as the first argument to this interpreter.

2. **Permission**

```
# chmod u+x myscript.sh
# ./myscript.sh
```

BASH Script

Bourne-**A**gain **SH**ell:

- Created by Shephen Bourne
- Largely compatible with *sh*
- Incorporates useful features of Korn shell csh and C shell csh
- A conformant of POSIX shell and IEEE POSIX specification (1003.1)

More information:

- www.gnu.org/s/bash/manual/bash.pdf
- www.gnu.org/software/bash/manual/bashref.html
- http://www.tldp.org/HOWTO/pdf/Bash-Prog-Intro-HOWTO.pdf

“Hello World” program

```
#!/bin/bash
```

Tells which program to interpret

```
echo “Hello World”
```

Actual line to print “Hello World”

```
$ ./hello.sh
```

Variables

Rule 1: there are no data types

Rule 2: a variable can be a number, a character or a string of characters

```
#!/bin/bash

STR="Hello World"
echo $STR
```

STR is defined as a variable

Value of a variable, put "$" before it

Variable Creation and Local Variables

In a directory with "a", "b" and "c" file.

Two commands:

echo ls

echo $(ls)

Result is "ls" since ls is a string here.

Result is "a b c" since we are echoing variables of executing "ls" command

```
#!/bin/bash
HELLO=Hello
function hello {
        local HELLO=World
        echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

$HELLO is "World" here.
Using LOCAL to mark local variables

$HELLO is "Hello" here.

Input parameters

The number of arguments:

$1: First parameter

$2: Second parameter

$#: the number of input parameters

Conditionals

Different forms:

- if EXPRESSION; then STATEMENT;
- if EXPRESSION; then STATEMENT1;
  else STATEMENT2
- if EXPRESSION1; then STATEMENT1;
  else
  if EXPRESSION2; then STATEMENT2;
  else STATEMENT3;

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expression evaluated as true
fi
```

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expression evaluated as true
else
   echo expression evaluated as false
fi
```

String comparison

(1) S1 matches S2

(2) S1 does not match S2

(3) S1 is less than S2

(4) S1 is not NULL

(5) S1 is NULL

(1) S1 = S2

(2) S1 != S2

(3) S1 < S2

(4) –n S1

(5) –z S1

Number comparison

<	–lt
>	–gt
<=	–le
>=	–ge
==	–eq
!=	–ne

String Comparison	Description
Str1 = Str2	Returns true if the strings are equal
Str1 != Str2	Returns true if the strings are not equal
-n Str1	Returns true if the string is not null
-z Str1	Returns true if the string is null
Numeric Comparison	**Description**
expr1 -eq expr2	Returns true if the expressions are equal
expr1 -ne expr2	Returns true if the expressions are not equal
expr1 -gt expr2	Returns true if expr1 is greater than expr2
expr1 -ge expr2	Returns true if expr1 is greater than or equal to expr2
expr1 -lt expr2	Returns true if expr1 is less than expr2
expr1 -le expr2	Returns true if expr1 is less than or equal to expr2
! expr1	Negates the result of the expression
File Conditionals	**Description**
-d file	True if the file is a directory
-e file	True if the file exists (note that this is not particularly portable, thus -f is generally used)
-f file	True if the provided string is a file
-g file	True if the group id is set on a file
-r file	True if the file is readable
-s file	True if the file has a non-zero size
-u	True if the user id is set on a file
-w	True if the file is writable
-x	True if the file is an executable