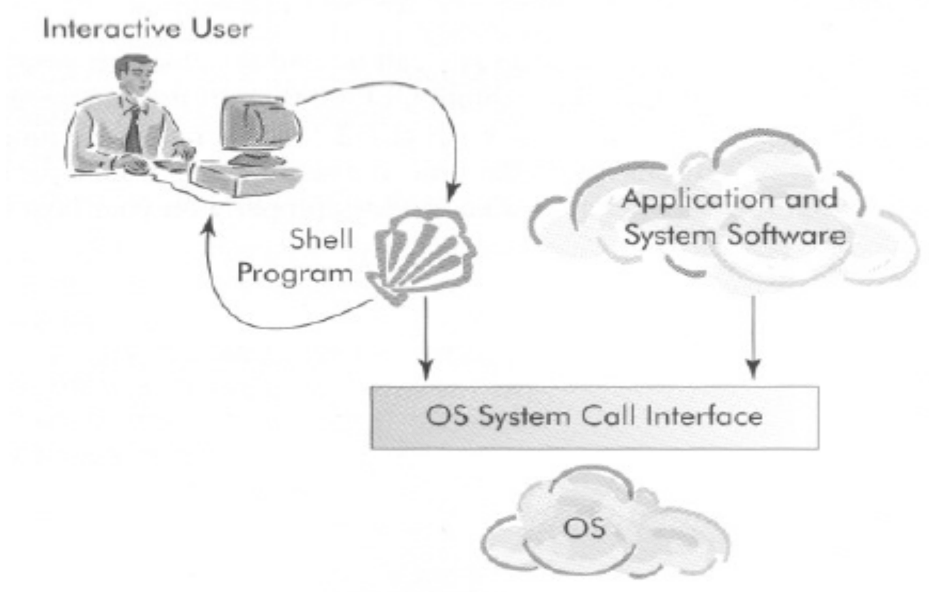
# Lecture-4:
# Bash Script

Tongping Liu
Tongping.Liu@utsa.edu

## Unix Shell

### As a commander interpreter

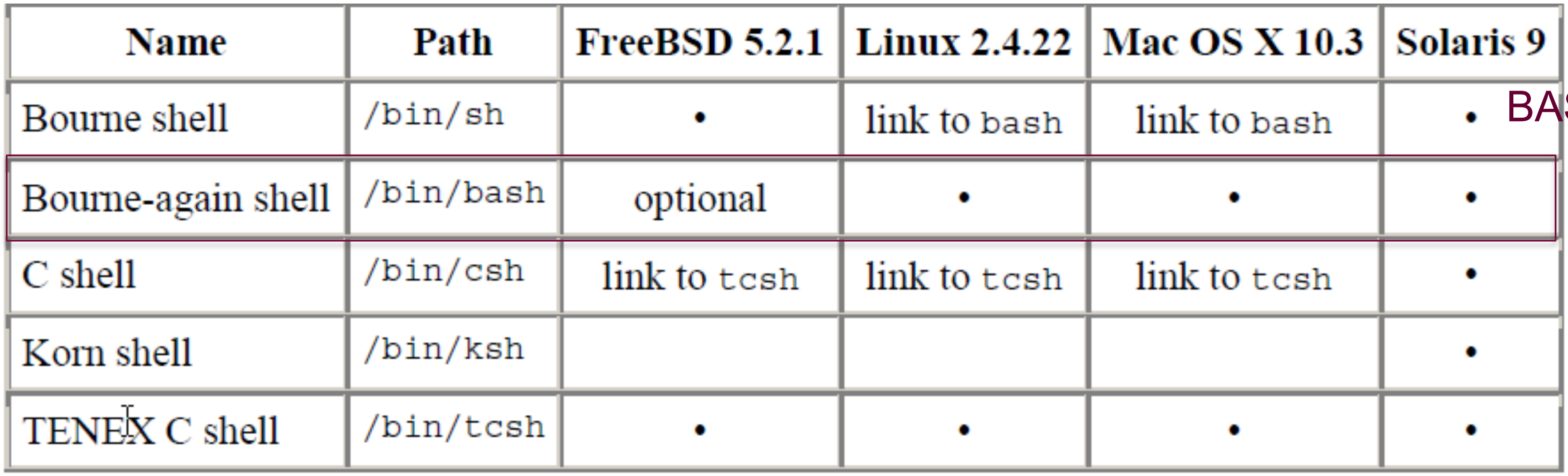
- Provides the user interface to many GNU utilities



### As a programming language

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

# Common Unix Shells



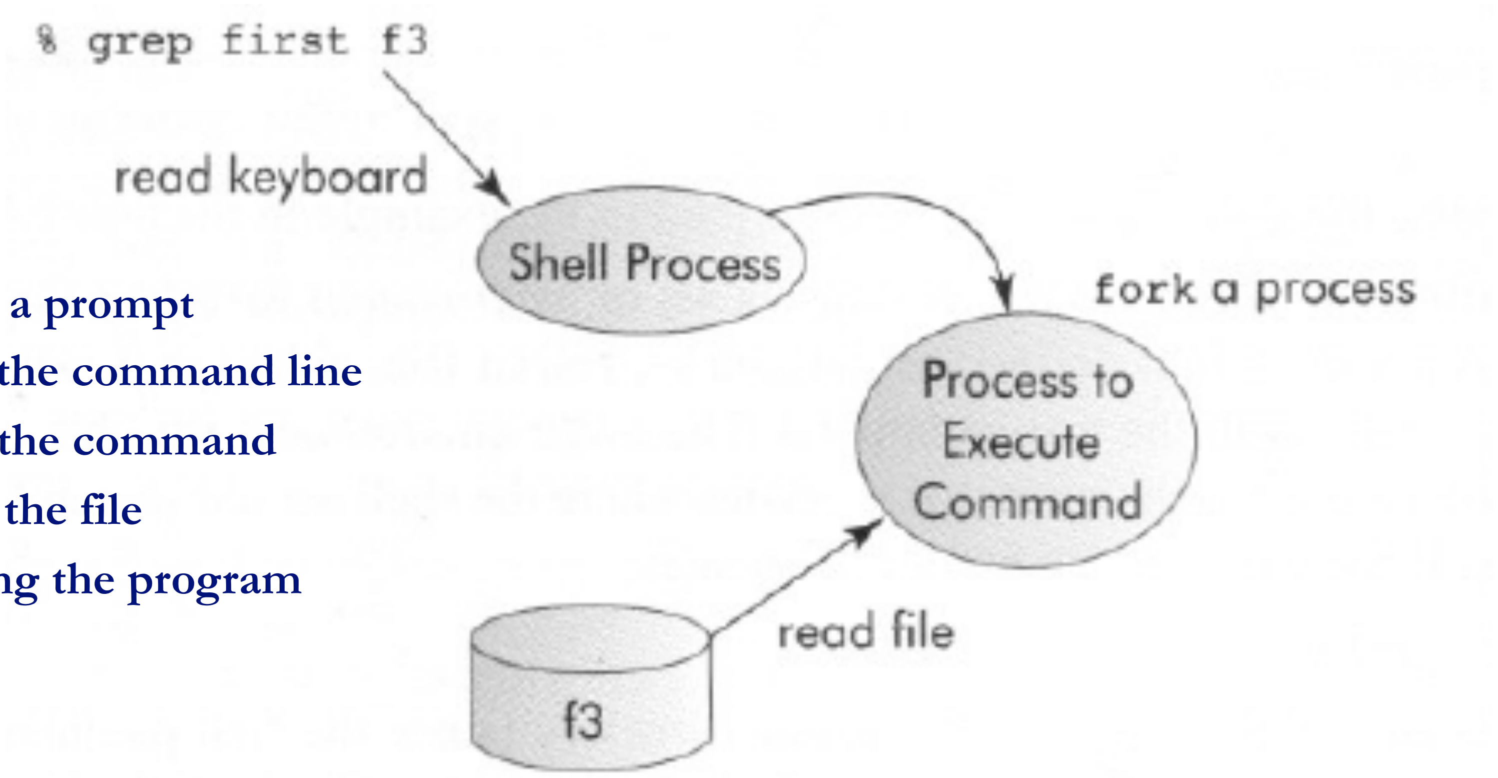
| Name | Path | FreeBSD 5.2.1 | Linux 2.4.22 | Mac OS X 10.3 | Solaris 9 |
|---|---|---|---|---|---|
| Bourne shell | `/bin/sh` | • | link to `bash` | link to `bash` | • |
| Bourne-again shell | `/bin/bash` | optional | • | • | • |
| C shell | `/bin/csh` | link to `tcsh` | link to `tcsh` | link to `tcsh`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

BASH

## Unix Shell

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

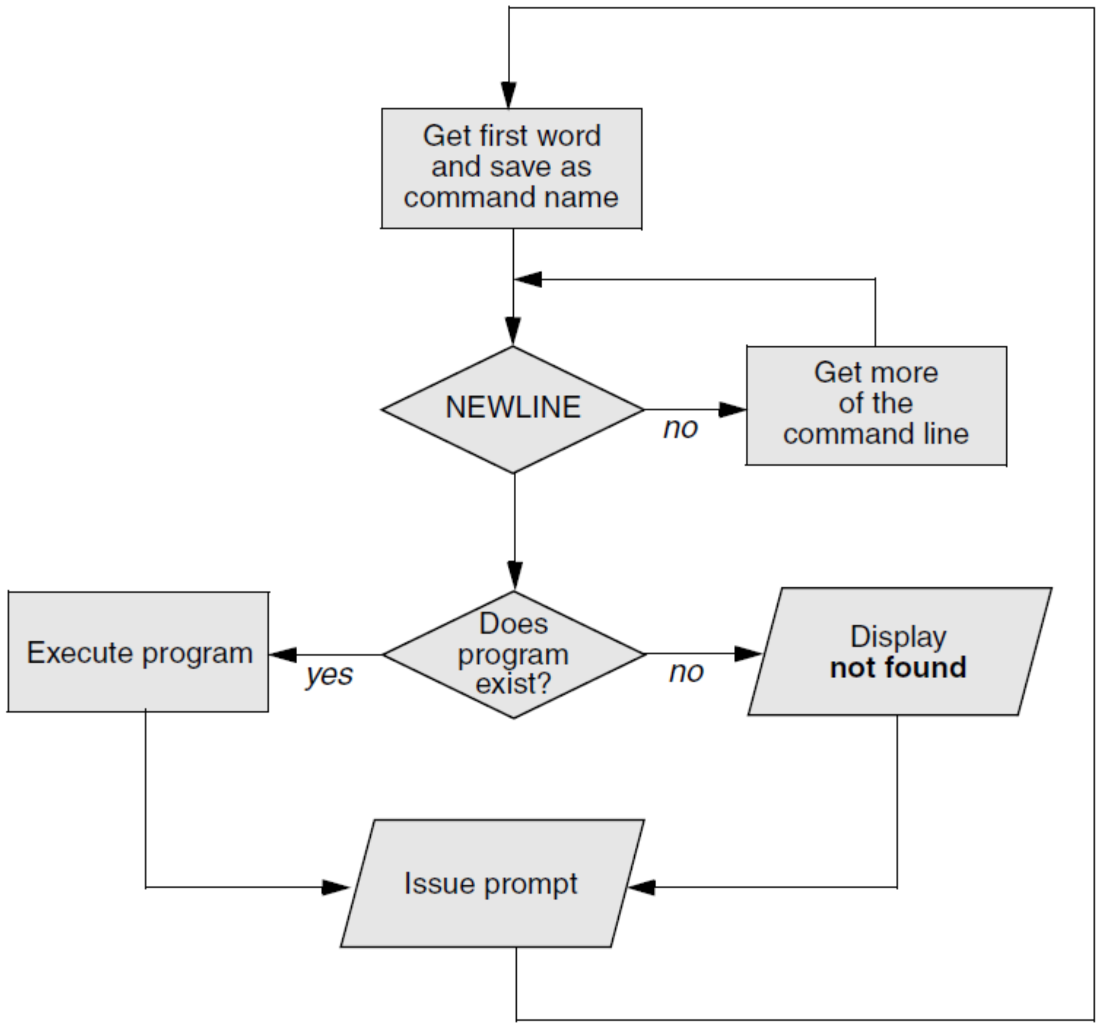
- Printing a prompt
- Getting the command line
- Parsing the command
- Finding the file
- Executing the program



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**

## 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.

Script

## 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
```

## Variable Creation and Local Variables

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

**Two commands:**

**echo ls**

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

**echo $(ls)**

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

## How to check parameters

```
# check the number of parameters
if [ "$#" -ne 2 ]; then
      # Printing the usage of the command line
      echo "The usage of this script:";
      echo "  ./test.sh INPUT OUTPUT";
      exit 1;
fi

sourceInput=$1;
destinedOutput=$2;

# Check difference to avoid overlapping
if [ "$sourceInput" = "$destinedOutput" ]; then
  echo "INPUT must be different with OUTPUT";
  exit 1;
fi

# Check whether the input file exists or not
if [ ! -f "$sourceInput" ]; then
  echo "$sourceInput does not exist";
  exit 1;
fi
```

Check the number of input parameters.

meaningful name

Is INPUT different with OUTPUT?

INPUT is existing?

## 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
```

## Conditionals(continued)

```
if [ "$1" = "cool" ]; then
    echo "Cool Beans”;
elif [ "$1" = "neat" ]; then
    echo "Neato cool”
else
  echo "Not Cool Beans”;
fi
```

=

```
if [ "$1" = "cool" ]; then
    echo "Cool Beans";
else
  if [ "$1" = "neat" ]; then
    echo "Neato cool";
  else
    echo "Not Cool Beans";
  fi
fi
```

## “and” vs “or”

“and” relation --- &&

“or” relation --- ||

```
#!/bin/bash
# Prompt for a user age...
echo "Please enter your age:"
read AGE
if [ "$AGE" -lt 20 ] || [ "$AGE" -ge 50 ]; then
  echo "Sorry, you are out of the age range."
elif [ "$AGE" -ge 20 ] && [ "$AGE" -lt 30 ]; then
  echo ”You are in your 20s"
elif [ "$AGE" -ge 30 ] && [ "$AGE" -lt 40 ]; then
  echo ”You are in your 30s"
elif [ "$AGE" -ge 40 ] && [ "$AGE" -lt 50 ]; then
  echo ”You are in your 40s"
fi
```

## String comparison

| | |
|---|---|
| (1) S1 matches S2 | (1) “S1” = “S2” |
| (2) S1 does not match S2 | (2) “S1” != “S2” |
| (3) S1 is less than S2 | (3) “S1” < “S2” |
| (4) S1 is not NULL | (4) –n “S1” |
| (5) S1 is NULL | (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 |

## for loop

**Let you to iterate over a series of “words” within a string**

```
#!/bin/bash

for i in $(ls); do
  echo “item: $i”;
done
```

Prints each item from the “ls” results.

```
item: a
item: b
item: c
item: run.sh
```

```
#!/bin/bash

for i in `seq 1 10`; do
  echo “item: $i”
done
```

C-like for loop, prints different numbers between 1 and 10

```
item: 1
item: 2
……
item: 10
```

## While loop

**Execute the code if the control expression is true. Only stops when it is false or a break inside.**

```
#!/bin/bash

Index=0
while [ $Index -lt 10  ]; do
          echo “now index is $Index”
          let Index=$Index+1;
done
```

## Until loop

**Similar to "while" loop, but execute the code while the control expression equals FALSE.**

```
#!/bin/bash

Index=0
while [ $Index -lt 10  ]; do
  echo "now index is $Index"
  let Index=$Index+1;
done
```

while

```
  now index is 0
  now index is 1
  …….
  now index is 8
  now index is 9
```

```
#!/bin/bash

Index=20
until [ $Index -lt 10  ]; do
   echo "now index is $Index"
   let Index=$Index-1;
done
```

until

```
now index is 20
now index is 19
…….
now index is 11
now index is 10
```

## “continue” and “break”

**These two keywords has the same meaning as C language.**

- **continue statement resumes iteration of an enclosing for, while, until or select loop.**
- **break statement is used to exit the current loop before its normal ending.**

```
LIST=$(ls);
for name in $LIST; do
  # if it is not a valid word
  if ! [[ $name =~ ^[A-Za-z-]+$ ]]; then
    continue;
  fi
  echo "$name need to be renamed";
  ORIG="$name";
  NEW=`echo $name | tr 'A-Z' 'a-z'`;
  mv "$ORIG" "$NEW";
done
```

Is this a valid word?

Change all letters to lower cases

## Cases

```
echo "Translate a number to a word";
echo "1. ONE";
echo "2. TWO";
read choice
case $choice in
  1) echo "ONE";;
  2) echo "TWO";;
  *) echo "INVALID, try 1 and 2.";;
esac
```

## Functions

As other language, a function is used to group code in a more logical way and avoid repeat

Definition:

```
function my_func_name {
    YOU CODE HERE
}
```

Calling a function by its name:

```
my_func_name
```

## Function Example

```
#!/bin/bash

function quit {
    exit;
}

function hello {
   echo “Hello World”
}

hello
quit;
echo “What is now”;
```

Output of this program?

```
Hello World
```

## Functions with parameters

```
#!/bin/bash

function quit {
    exit;
}

function newFunc {
   echo $1; // print the parameter
}

newFunc “Hello World”
quit;
echo “What is now”;
```

Output of this program?

```
Hello World
```

## Return value of a function

**Return value: can only return numeric value between 0 and 255.**

- If you return -1, then you get 255
- If you return 256, then you get 0

**How to get the return value:**

- (1) myFunc myParameter;

  retvalue = $?;

- (2) retvalue =$(myFunc myParameter)

## Return value of a program

**$? Is used to fetch the return value.**

```
#!/bin/bash

cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

If /dada is not existing, then we will get the result like this:

```
rv: 1
rv: 0
```

## Limitation of return value

**We can't return a value larger than 255.**

- **Portable shell is requires to pass an unsigned decimal integer, no greater than 255, for defined behavior.**
- **More: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#return**

## Array

http://www.thegeekstuff.com/2010/06/bash-array-tutorial/

### Initialization:

- myArray=()   ; empty array
- myArray[0] = ‘first’; myArray[1]=‘second;’
- declare -a myArray=(‘first’ ‘second’);

### Add an element or elements:

- myArray+=(‘third’ ‘four’);

### Get the value of an element

- Echo “${myArray[1]}”

### Increment the value of an element

- ((myArray[$index]++))

## Two Dimension Array?

No two dimension array

## More about Shell

http://www.gnu.org/software/bash/manual/bashref.html

http://www.gnu.org/software/bash/manual/bash.pdf (166)

## Homework today

Writing a script to compare two directories, lists:

Files that are in directory DIR1, but not DIR2

Files that are in both directories.

Files that are in directory DIR2, but not DIR1.