

Plan for today

- Unix
 - Introduction
 - Utilities
 - Shell, Bourne shell command and shell script
 - Review
 - Leftover: small but useful
 - set, shift, cut, redirection, /dev/null, function
 - Review and new topics for C
 - enum....
 - midterm review

Utilities II – advanced utilities

grep/egrep	RE	<code>grep -w -i [Tt]he file123</code>
sort		<code>sort +3 -4 -r -n/M file</code>
uniq		<code>uniq -c</code>
cut		<code>cut -d" " -f2,3</code>
awk		<code>awk -F":" '{print \$1,\$2}'</code>
tr		<code>tr a-z A-Z < filename</code>
cmp/diff		<code>cmp/diff</code>
ln		<code>ln</code>
find		<code>find . -name "*.c" -exec</code>
		<code>cp {} {}.bak \;</code>

Start column 0, default delimiter blank/tab

Start column 1 Default delimiter tab

Start column 1 Default delimiter blank/tab

gREp, eREep RE: regular expression

Pattern	Maning	Example
c	Non-special, matches itself	'tom'
\c	Turn off special meaning	\"\$'
\$	End of line	'ab\$'
^	Start of line	^ab
.	Any single character	'..nodes'
[...]	Any single character in []	'[tT]he'
[^...]	Any single character not in []	'[^tT]he'
R*	Zero or more occurrences of R	'e*'
R+	One or more occurrences of R (egrep)	'e+'
R?	Zero or one occurrences of R (egrep)	'e?'
R1R2	R1 followed by R2	'[st][fe]'
R1 R2	R1 or R2 (egrep)	'the The'

Don't get confused with UNIX metacharacter (file name wildcards) →

```
ls file*.c *.java
cp file?.sh .
```

Find Utility

- find pathList expression
- finds files starting at pathList
- finds files descending from there
- **Allows you to perform certain actions**
 - e.g. deleting the files
 - e.g., copying , mv, rm

“Find all the c files and make a backup of them/rename to 2012“

```
find . -name "*.c" -exec cp {} {}.bak \;
```

```
find . -name "*.c" -exec mv {} {}.2012 \;
```

Finding Files: find

- `find` startingDir searchOptions commandToPerform

```
$ find . -name az.c -print      # print C source files
                                # in the current directory or
                                # any of its subdirectories.
```

```
./proj/fall.89/play.c
./proj/fall.89/rerefee.c
./proj/fall.89/player.c
./rock/guess.c
```

```
$ find /code -mtime -14 -ls    # list modified files during the last 14 days
```

```
$ find . -name '*.txt' -print  # find all text files in the current directory
```

Find Utility

- `-mtime count`
 - true if the file has been modified within count days
- `-atime count`
 - true if the file has been accessed within count days
- `-ctime count`
 - true if the contents of the file have been modified within count days or any of its file attributes have been modified
- `-exec command`
 - true if the exit code = 0 from executing the command.
 - command must be terminated by `\;`
 - If `{}` is specified as a command line argument it is replaced by the file name currently matched

Find Examples

- `$ find / -name x.c`
 - searches for file x.c in the entire file system
- `$ find . -mtime 14 -ls`
 - lists files modified in the last 14 days
- `$ find . -name '*.bak' -ls -exec rm {} \;`
 - ls and then remove all files that end with .bak
- `$ find . -name 'a?.c' -exec cp {} {}.bak \;`
 - Find all a?.c and then cp it to a?.c.bak
 - a1.c -> a1.c.bak
 - a2.c -> a2.c.bak

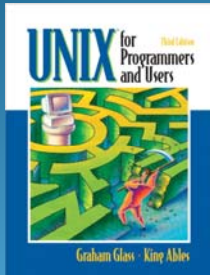
Utilities II – advanced utilities

Introduces utilities for power users, grouped into logical sets

We introduce about thirty useful utilities.

section	Utilities
Filtering files	<code>egrep</code> , <code>fgrep</code> , <code>grep</code> , <code>uniq</code>
Sorting files	<code>sort</code>
Comparing files	<code>cmp</code> , <code>diff</code>
Archiving files	<code>tar</code> , <code>cpio</code> , <code>dump</code>
Searching for files	<code>find</code>
Scheduling commands	<code>at</code> , <code>cron</code> , <code>crontab</code>
Programmable text processing	<code>awk</code> , <code>perl</code>
Hard and soft links	<code>ln</code>
Switching users	<code>su</code>
Checking for mail	<code>biff</code>
Transforming files	<code>compress</code> , <code>crypt</code> , <code>gunzip</code> , <code>gzip</code> , <code>sed</code> , <code>tr</code> , <code>ul</code> , <code>uncompress</code>
Looking at raw file contents	<code>od</code>
Mounting file systems	<code>mount</code> , <code>umount</code>
Identifying shells	<code>whoami</code>
Document preparation	<code>nroff</code> , <code>spell</code> , <code>style</code> , <code>troff</code>
Timing execution of commands	<code>time</code>

UNIX Shells



Ch 4 Unix shells
“UNIX for Programmers and Users”
Third Edition, Prentice-Hall, GRAHAM GLASS, KING ABLES

• INTRODUCTION



```
Command Prompt
Volume Serial Number is E814-2F58
Directory of C:\Users\hamie\Desktop\TRICOM\
03/18/2013  04:30 PM  (DIR)          .
03/18/2013  04:30 PM  (DIR)          ..
03/18/2013  04:30 PM  (DIR)          18 '
                1 Files(s)    18 bytes
                1 Dir(s)   96,432,253,148 bytes free

C:\Users\hamie\Desktop\TRICOM>type n a
C:\Users\hamie\Desktop\TRICOM>dir
Volume in drive C has no label
Volume Serial Number is E814-2F58
Directory of C:\Users\hamie\Desktop\TRICOM\
03/18/2013  04:31 PM  (DIR)          .
03/18/2013  04:31 PM  (DIR)          ..
03/18/2013  04:30 PM  (DIR)          18 '
                1 Files(s)    18 bytes
                2 Dir(s)   96,432,253,148 bytes free

C:\Users\hamie\Desktop\TRICOM>
```

A shell is a program that is an interface between a user and the raw operating system.

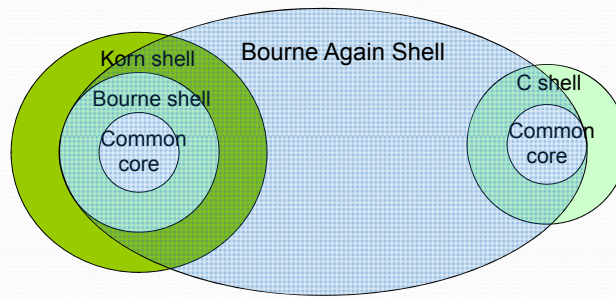
It makes basic facilities such as multitasking and piping easy to use, and it adds useful file-specific features such as wildcards and I/O redirection.

There are four common shells in use:

- the Bourne shell
- the Korn shell
- the C shell
- the Bash shell (Bourne Again Shell)

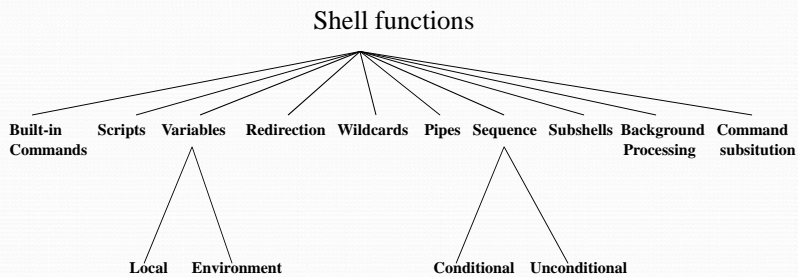
- SHELL FUNCTIONALITY

- This part describes the common core of functionality that all four shells provide.
- The relationship among the four shells:



- SHELL FUNCTIONALITY

- A hierarchy diagram is a useful way to illustrate the features shared by the four shells



Shell functions

- **Displaying Information : echo**

The **built-in echo command** displays **its arguments** to standard output and works like this:

Shell Command: **echo** {arg} *

echo is a **built-in shell command** that displays all of its arguments **to standard output**.
By default, it appends a new line to the output.

use **-n** to fix

```
sh-3.00$ echo hello world    # echo "hello world"
hello world
sh-3.00$ echo -n hello
hellosh-3.00$
```

- **METACHARACTERS**

Some characters are processed specially by a shell and are known as **metacharacters**.

All four shells share a core set of **common** metacharacters, whose meanings are as follow:

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 the brackets.

Shell functions

Built-in Commands Scripts Variables (Local, Environment) Redirection Wildcards Pipes Sequence (Conditional, Unconditional) Subshells Background Processing Command substitution

Symbol	Meaning
<code>`command`</code>	Command substitution; replaced by the output from command.
<code>\$</code>	Variable substitution. Expands the value of a variable.
<code>&</code>	Runs a command in the background.
<code> </code>	Pipe symbol; sends the output of one process to the input of another.
<code>;</code>	Used to sequence commands. % <code>echo hello; wc lyrics</code>
<code> </code>	Conditional execution; executes a command if the previous one fails.
<code>&&</code>	Conditional execution; executes a command if the previous one succeeds.
<code>(...)</code>	Groups commands.
<code>#</code>	All characters that follow up to a new line are ignored by the shell and program(i.e., used for a comment)
<code>\</code>	Prevents special interpretation of the next character.
<code><<tok</code>	Input redirection; reads standard input from script up to tok.

- 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 by a **backslash(\)** character. also ' ' " " (later)

Here's an example:

```

$ echo hi > file ---> store output of echo in "file".
$ cat file ---> look at the contents of "file".
hi
$ echo hi \> file ---> inhibit > metacharacter.
hi > file ---> > is treated like other characters.
$ cat file ---> look at the file again.
hi

$ echo 3 \* 4 = 12
$ echo 3 + 2 = 5 # this is a comment

```


Shell functions

Redirection > >> < <<

The shell redirection facility allows you to:

- 1) store the output of a process to a file ([output redirection](#))
- 2) use the contents of a file as input to a process ([input redirection](#))

Output redirection

To redirect output, use either the ">" or ">>" metacharacters.

The sequence

```
$ 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 does already exist.

• Input Redirection

Input redirection is useful because it allows you to [prepare a process input](#) beforehand and store it in a file for later use.

To [redirect input](#), use either the '<' or '<<' metacharacters.

The sequence

```
$ 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.

- When the shell encounters a sequence of the form

`$ command << word`

- it copies its standard input up to, but not including, the line starting with word into a buffer and then executes command using the contents of the buffer as its standard input.

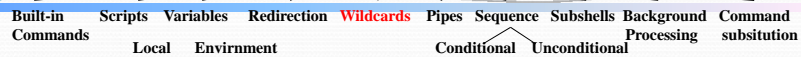
- that allows shell programs(scripts) to supply the standard input to other commands as in-line text,

```
$ cat << eof
> line 1
> line 2
> line 3
> eof
line 1
line 2
line 3
$_
```

'Here' document
Useful for output large info

```
cat <<ENDOFMENU
-----
| (d)isplay db      (n)ew record      (u)pdate record  |
| (s)ort records   (a)verage age    (s)earch (r)ecord |
| (c)lear db       (c)lear (r)ecord (f)ormatted display |
| (v)iew courses  (co)unt enrollment (q)uit          |
-----
ENDOFMENU
```

Shell functions



• FILENAME SUBSTITUTION (WILDCARDS)

- All shells support a wildcard facility that allows you to select files that satisfy a particular name pattern from the file system.

- The wildcards and their meanings are as follows: `ls *.c` `cp a?.c .`

Wildcard	Meaning
*	Matches any string, including the empty string. 0-more char
?	Matches any single character. Exactly one
[..]	Matches any one of the characters between the brackets. A range of characters may be specified by separating a pair of characters by a hyphen.

Don't confuse with Regulation Expression → `grep a*b file123` `grep a?.c file123`

- Prevent the shell from processing the wildcards in a string by surrounding the string with single quotes (apostrophes) or double quotes. (talk shortly)

Here are some examples of wildcards in action:

```
$ ls *.c ---> list any text ending in ".c".
```

```
a.c b.c cc.c
```

```
$ ls ?.c ---> list text for which one character is followed by ".c".
```

```
a.c b.c
```

```
$ cp /cs/dept/course/2012-13/W/2031/file? .
```

```
$ cp /cs/dept/course/2012-13/W/2031/file[12] .
```

```
$ ls [ac]* ---> list any string beginning with "a" or "c".
```

```
a.c abcd a3.pdf cc.c ce3.doc
```

```
$ ls [A-Za-z]* ---> list any string beginning with a letter.
```

```
a.c b.c cc.c
```

```
$ ls dir*/*.c ---> list all files ending in ".c" files in "dir*"
---> directories ( that is, in any directories beginning
with "dir" ).
```

```
dir1/d.c dir2/g.c
```

```
$ ls */*.c ---> list all files ending in ".c" in any subdirectory.
```

```
dir1/d.c dir2/g.c
```

Shell functions

- **PIPES**
- Shells allow you to use the standard output of one process as the standard input of another process by connecting the processes together using the pipe() metacharacter.
- The sequence


```
$ command1 | command2
```

 causes the standard output of command1 to “flow through” to the standard input of command2.
- Any number of commands may be connected by pipes.

A sequence of commands changed together in this way is called a *pipeline*.

```

$ head -4 /etc/passwd ---> look at the password file.
root:eJ2S10rVe8mCg:0:1:Operator:/:/bin/csh
nobody:*:65534:65534::/:
daemon:*:1:1::/:
sys:*:2:2:/:/bin/csh
$ cat /etc/passwd | awk -F: '{ print $1 }' | sort
audit
bin
daemon
glass
ingres
news
nobody
root
sync
sys
tim
uucp
$ _
  
```

```

graph LR
  ls[ls] -- Pipe --> awk[awk]
  awk -- Pipe --> sort[sort]
  sort --- Terminal[Terminal]
  
```

Shell functions

Built-in Commands
Scripts
Variables
Local Environment
Redirection
Wildcards
Pipes
Sequence
Conditional Unconditional
Subshells
Background Processing
Command substitution

COMMAND SUBSTITUTION used heavily in script

A command surrounded by **grave accents** (`) - back quote - is executed, and **its standard output** is inserted in the command's place in the entire command line. Any new lines in the output are replaced by spaces.

For example:

```
$ echo the date today is `date`, right?
the date today is Mon Feb 2 00:41:55 CST 1998, right?
$_
```

```
$ echo there are `wc -l classlist` students in the class
there are 71 students in the class

$ echo there are `cat classlist | grep -w Wang | wc -l` students with name Wang
there are 1 students with name Wang

$x=`wc -l classlist` # x get value 71 (talk later)
```

Shell functions

Built-in Commands
Scripts
Variables
Local Environment
Redirection
Wildcards
Pipes
Sequence
Conditional Unconditional
Subshells
Background Processing
Command substitution

• **SEQUENCES** ; ;

If you enter a **series of simple commands or pipelines** separated by semicolons, the shell will **execute them in sequence, from left to right**.

This facility is useful for **type-ahead(and think-ahead)** addicts who like to specify an entire sequence of actions at once.

Here's an example:

```
$ date; pwd; ls ---> execute three commands in sequence.
Mon Feb 2 00:11:10 CST 1998
/home/glass/wild
a.c b.c cc.c dir1 dir2
$_
```

- Each command in a sequence may be individually I/O redirected as well:

```
$ date > date.txt; ls; pwd > pwd.txt
a.c b.c cc.c date.txt dir1 dir2
```

Shell functions

- **Conditional Sequences** `&&` `||`
- Every UNIX process terminates with an **exit value**.
By convention, an exit value of 0 means that the process completed **successfully**, and a nonzero exit value indicates **failure**. (opposite to C)
- All built-in shell commands return a value of 0 if they succeed and a non-zero value if they fail.
You may construct sequences that make use of this exit value:
 - 1) If you specify a series of commands separated by `"&&"` tokens, the next command is executed only if the previous command returns an exit code of 0 -- successful
 - 2) If you specify a series of commands separated by `"||"` tokens, the next command is executed only if the previous command returns a nonzero exit code -- fails

- For example, if the C compiler `gcc` compiles a program without fatal errors, it creates an executable program called `"a.out"` and returns an exit code of 0; otherwise, it returns a nonzero exit code.

```
$ gcc myprog.c && a.out # if gcc successful, do a.out
```

- The following conditional sequence compiles a program called `"myprog.c"` and displays an error message if the compilation fails:

```
$ gcc myprog.c || echo "compilation failed."
```

```
$ grep huiwang classlist && echo "find this guy"
```

return 0 if match, return 1 otherwise

- **GROUPING COMMANDS ()**

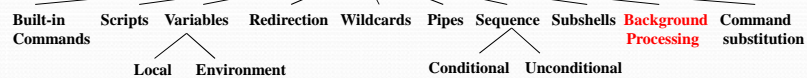
- Commands may be grouped by placing them between parentheses, which causes them to be executed by a child shell(subshell).
- The group of commands shares the same standard input, standard output, and standard error channels and may be redirected and piped as if it were a simple command.

```

$ date; ls; pwd > out.txt      ---> execute a sequence.
Mon Feb 2 00:33:12 CST 1998  ---> output from date.
a.c      b.c                  ---> output from ls.
$ cat out.txt                  ---> only pwd was redirected.
/home/glass
$ ( date; ls; pwd ) > out.txt  ---> group and then redirect.
$ cat out.txt                  ---> all output was redirected.
Mon Feb 2 00:33:28 CST 1998
a.c      b.c
/home/glass
$ _

```

Shell functions



- **Background Processing &**

- If you follow a simple command, pipeline, sequence of pipelines, or group of commands by the "&" metacharacter, a subshell is created to execute the commands as a background process
- The background process runs concurrently with the parent shell and does not take control of the keyboard.

Background processing is therefore very useful for performing several tasks simultaneously, as long as the background tasks do not require input from the keyboard.

Shell functions

- **SHELL PROGRAMS: SCRIPTS**
- Any series of shell commands may be stored inside a regular text file for later execution.
- A file that contains shell commands is called *a script*.
batch file in Windows
- Before you can run a script, you must give it execute permission by using the **chmod** utility.
`chmod u+x filename`

```
echo hello world
date
```

- to run it, you need only to type its name.
- Scripts are useful for storing commonly used sequences of commands, and they range in complexity from simple one-liners to fully blown programs.

- **SHELL PROGRAMS: SCRIPTS**
- The system decides which shell the script is written for by examining the first line of the script.
- Here are the rules that it uses to make this decision:
 - 1) If the first line of the script is just a pound sign(#), then the script is interpreted by the shell from which you executed this script as a command.
 - 2) If the first line of the script is of the form `#! path name`, then the executable program `pathName` is used to interpret the script.
 - 3) If neither rule1 nor rule2 applies, then the script is interpreted by a Bourne shell (`sh`).
Note: Bash on Linux, MacOS X is positioned as `/bin/sh`.

• SHELL PROGRAMS: SCRIPTS

one for the Bash shell and the other for the Korn shell.

```
$ cat > script.sh ---> create the bash script.
```

```
#!/bin/sh
# This is a sample sh script.
echo "hello world"
echo -n "the date today is " # in sh, -n omits new line
date # output today's date.
^D
---> end of input.
```

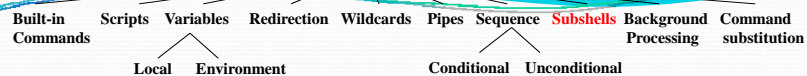
```
$ chmod u+x script.sh ---> make the scripts executable.
```

```
$ script.sh ---> execute the C-shell script.
```

```
hello world
```

```
The date today is Sun Feb 1 19:50:00 CST 2004
```

Shell functions



• SUBSHELLS

When you log into a UNIX system, you execute an initial login shell.

This initial shell executes any simple commands that you enter.

- current(parent) shell creates a new(child) shell to perform some tasks:

- 1) When a grouped command, such as (ls; pwd; date), is executed, the parent shell creates a child shell to execute the grouped commands.

If the command is not executed in the background, the parent shell sleeps until the child shell terminates.

2) When a script is executed, the parent shell creates a child shell to execute the commands in the script.

If the script is not executed in the background, the parent shell sleeps until the child shell terminates.

3) When a background job is executed, The parent shell creates a child shell to execute the background commands.

The parent shell continues to run concurrently with the child shell.

• VARIABLES \$

- A shell supports two kinds of variables:
local and environment variables.

local:
user defined, x=4
positional \$0 ... \$9

Both kinds of variables hold data in a string format.

the child shell gets a copy of its parent shell's environment variables, but not its local variables.

Environment variables are therefore used for transmitting useful information between parent shells and their children.

• **Environment VARIABLES** (for your reference)

- Here is a list of the predefined environment variables that are common to all shells:

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

Built-in local variables (must know!)

-several common built-in local variables that have special meanings:

Name	Meaning
\$\$	The process ID of the shell.
\$0	The name of the shell script (if applicable).
\$1..\$9	\$n refers to the nth command line argument (if applicable).
\$*	A list of all the command-line arguments.

\$ myscript we are the world

\$0 \$1 \$2 \$3 \$4


```

#!/bin/sh

echo the name of this script is $0
echo the first argument is $1
echo a list of all the arguments is $*
echo this script places the date into a temporary file called $1.$$
date > $1.$$ # redirect the output of date.
ls $1.$$ # list the file.
rm $1.$$ # remove the file.
x=5
echo the value of variable x is $x


```

\$variables.sh paul ringo george john --> execute the script.
 the name of this script is variables.sh
 the first argument is paul
 a list of all the arguments is paul ringo george john
 this script places the date into a temporary file called paul.2431
 paul.2431
 value of variable x is 5.

\$ _

\$ variable substitution

• QUOTING



There are often times when you want to inhibit the shell's wildcard-replacement * ? [], variable-substitution \$, and/or command-substitution ` ` mechanisms.

The shell's quoting system allows you to do just that.

- Here's the way that it works:

- 1) Single quotes(' ') inhibits wildcard replacement, variable substitution, and command substitution.
- 2) Double quotes(" ") inhibits wildcard replacement only.

• QUOTING

- The following example illustrates the difference between the two different kinds of quotes:

```
$ echo 3 * 4 = 12    ---> remember, * is a wildcard.
3 a.c b b.c c.c 4 = 12
```



```
$ echo "3 * 4 = 12" ---> double quotes inhibit wildcards.
3 * 4 = 12
```

```
$ echo '3 * 4 = 12' ---> single quotes inhibit wildcards.
3 * 4 = 12
```

another way?

- By using **single quotes (apostrophes)** around the text, we inhibit all **wildcarding and variable and command substitutions**:

```
$ name=Graham # assign value to name variable
```

```
$ echo 'my name is $name - date is `date`'
my name is $name - date is 'date'
$ _
```

- By using **double quotes around** the text, we inhibit **wildcarding**, but allow **variable and command substitutions**:

```
$ echo "my name is $name - date is `date`"
my name is Graham - date is Mon Feb 2 23:14:56 CST 1998
$ -
```

- TERMINATION AND EXIT CODES

Every UNIX process **terminates with an exit value**.

By convention, **an exit value of 0** means that the process **completed successful**, and **a nonzero exit value** indicates **failure**.

All built-in commands return an **exit value of 0** if they succeed return an non-zero is they fail.

In the Bash, Bourne and Korn shells, the special shell variable **\$?** always contains the value of the previous command's exit code.



date, ls, grep, find Every command has a exit code (return status)
Look for man

Matching 0	No matching 1	No such file 2
------------	---------------	----------------

-In the following example, the **date** utility succeeded, whereas the **gcc** and **awk** utilities failed:

```
$ date          ---> date succeeds.  
Mon Feb 2 22:13:38 CST 1998  
$ echo $?      ---> display its exit value.  
0          ---> indicates success.  
  
$ gcc prog.c   ---> compile a nonexistent program.  
cpp: Unable to open source file 'prog.c'.  
$ echo $?      ---> indicates failure.  
1
```

-In the following example, the `date` utility succeeded, whereas the `gcc` and `awk` utilities failed:

```
$ grep Wang classlist
$ echo $?
0          ---> indicates success.

$grep Hui classlist
$ echo $?
1          ---> indicates failure (not matching)

$grep Hui classlistX
grep: classlistX: No such file or directory
$ echo $?
2          ---> indicates failure (not matching)
```

- Any script that you write should always explicitly **return an exit code**.

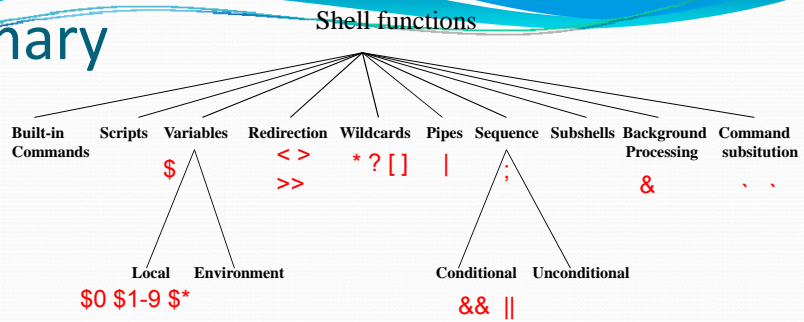
To terminate a script, use the built-in `exit` command, which works as follows:

Shell Command: `exit number`

`exit` terminates the shell and returns **the exit value number** to its parent process.

If **number is omitted**, the **exit value of the previous command** is used.

summary



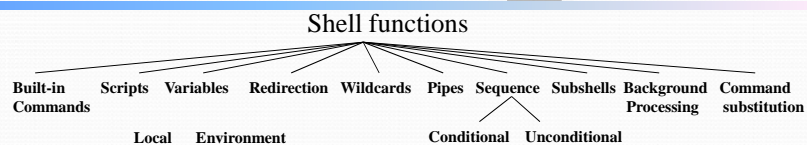
- Covered core shell functionality

- Built-in commands
- Redirection
- Wildcards
- Pipes
- Background processing
- Scripts
- Variables

The Bourne Shell and its script

Ch5 Bourn shell
"UNIX for Programmers and Users"
Third Edition, Prentice-Hall, GRAHAM GLASS, KING ABLES

Ch. 4. The Bourne Shell



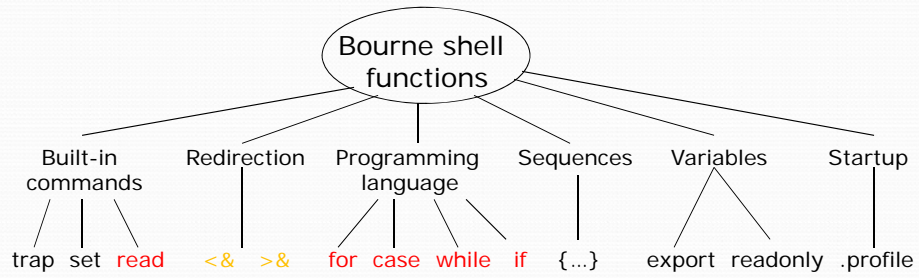
• Introduction

- The Bourne shell supports **all of the core-shell facilities** describe earlier, plus the following new facilities:
 - several ways to **set and access variables**
 - **a built-in programming language** that supports conditional branching, looping, and interrupt handling
 - **extensions** to the existing redirection and command-sequence operations
 - several **new built-in commands**

Ch. 4. The Bourne Shell

• Introduction

- These new facilities are described by this chapter and are illustrated by the following **hierarchy diagram**:



51

CONTENTS

- The Bourne shell, written by Stephen Bourne, was the first popular UNIX shell and is available on all UNIX systems.
- This section introduces the following utilities,

`expr` `test`

- Shell commands

This section introduces the following shell commands,

<code>break</code>	<code>for..in..do..done</code>	<code>set</code>
<code>case..in..esac</code>	<code>if..then..elif..fi</code>	<code>trap</code>
<code>continue</code>	<code>read</code>	<code>while..do..done</code>
<code>export</code>	<code>readonly</code>	

52

Ch. 4. The Bourne Shell

- **STARTUP**

- The Bourne shell is a regular C program whose executable file is stored as `"/bin/sh"`.

If your chosen shell is `"/bin/sh"`, an interactive Bourne shell is invoked automatically when you log into UNIX.

You may also invoke a Bourne shell manually from a script or from a terminal by using the command `sh`.

53

Ch. 4. The Bourne Shell

- **VARIABLES (setting and getting)**

- The Bourne shell can perform the following variable-related operations:

- simple assignment and access
- testing a variable for existence
- reading a variable from standard input
- making a variable read only
- exporting a local variable to the environment

- **Creating/Assigning a Variable**

The Bourne-shell syntax for assigning a value to a variable is:

```
{name=value} +
```

54

Ch. 4. The Bourne Shell

• VARIABLES

```
$ firstName=Graham
$ lastName=Glass
$ age=29
```

No space!

---> assign variables.

```
$ echo $firstName $lastName is $age
Graham Glass is 29
```

---> simple access variable substitution

```
$ name=Graham Glass
Glass: not found
```

---> syntax error.

```
$ name="Graham Glass"
$ echo $name
Graham Glass
$ -
```

---> use quotes to built strings.
---> now it works.

No need to declare! If assigned does not exist, create!

55

Ch. 4. The Bourne Shell

• Accessing a Variable

- The Bourne shell supports the following access methods:

Syntax	Action
<code>\$name</code>	Replaced by the value of name.
<code>\${name}</code>	Replaced by the value of name.
<code>\${name-word}</code>	Replaced by the value of name if set, and word otherwise.
<code>\${name+word}</code>	Replaced by the word if name is set, and nothing otherwise.
<code>\${name=word}</code>	Assigns word to the variable name if name is not already set and then is replaced by the value of name
<code>\${name?word}</code>	Replaced by name if name is set. If name is not set, word is displayed to the standard error channel and the shell is exited. If word is omitted, then a standard error message is displayed instead.

56

Ch. 4. The Bourne Shell

- Example

```
$ verb=sing          ---> assign a variable.
$ echo I like $verbing ---> there's no variable "verbing".
I like

$ echo I like ${verb}ing ---> now it works.
I like singing
$ -
```

57

Ch. 4. The Bourne Shell

- Reading a Variable from Standard Input

The read command allows you to read variables from standard input and works like this:

Shell Command: `read {variable} +`

read reads one line from standard input and then assigns successive words from the line to the specified variables.

Any words that are left over are assigned to the last named variable.

58

Ch. 4. The Bourne Shell

- If you specify just one variable,
the entire line is stored in the variable.

Here's an example script that prompts a user for his or her full name:

```
$ cat readName.sh ---> view the script.
```

```
echo -n "Please enter your name: "  
read name # read just one variable.  
echo your input is $name # display the variable.
```

```
$ readName.sh
```

```
Please enter your name: Graham Walker Glass
```

```
your input is Graham Walker Glass ---> the whole line was read.
```

```
$ -
```

59

Ch. 4. The Bourne Shell

- Here's other example script that prompts a user for his or her full name:

```
$ cat readNames.sh
```

```
echo -n "Please enter your name: "  
read first last # read two variables.  
echo your first name is $first # display the variables.  
echo your last name is $last
```

```
$ readNames.sh
```

```
Please enter your name: Graham Walker Glass
```

```
your first name is Graham
```

```
your last name is Walker Glass ---> the whole rest line was read.
```

```
$ -
```

Try yourself →

```
sh-3.00$ read a b  
1 2 3 4 5 6 7  
sh-3.00$ echo $a  
1  
sh-3.00$ echo $b  
2 3 4 5 6 7
```

```
sh-3.00$ read a b c  
1 2  
sh-3.00$ echo $a  
1  
sh-3.00$ echo $b  
2
```

60

Ch. 4. The Bourne Shell

- Predefined Local Variables

In addition to the core predefined local variables ($$$, $0, $1..9, $*$) the Bourne shell defines the following local variables:

Name	Value
$@$$	an individually quoted list of all of the positional parameters
$ $#$	the number of positional parameters
$ $?$	the exit value of the last command
$ $$$	the process ID of this shell

61

-Recall: several common (core) built-in local variables that have special meanings:

Name	Meaning
$ $$$	The process ID of the shell.
$ 0	The name of the shell script(if applicable).
$ $1..$9$	$ n refers to the $ nth$ command line argument (if applicable).
$ $*$	A list of all the command-line arguments.

$$ myscrip we are the world`$

$$0 $1 $2 $3 4

$$# = 4$

$$*$

$@$$

Ch. 4. The Bourne Shell

- Here's a small shell script that illustrates the first three variables.

```
$ cat script.sh
echo there are $# command line arguments: $@
grep -w $2 $1
echo the last exit value was $?           # display exit code.
```

```
$ script.sh 2031classlist Hui
there are 2 command line arguments: 2031classlist Hui
the last exit value was 1
$ -
```

```
$ script.sh 2031classlist Wang
there are 2 command line arguments: 2031classlist Wang
cse*****      zi*****      *****      Wang, Zi*****
the last exit value was 0      ---> match find
$ -
```

63

Ch. 4. The Bourne Shell

- **ARITHMETIC**

- Although the Bourne shell doesn't directly support arithmetic, it may be performed by using the `expr` utility, which works like this:

Utility : `expr expression` `$expr 2 + 4`

`expr` evaluates *expression* and sends the result to standard output.

All of the components of expression must be separated by blanks,

The result of *expression* may be assigned to a shell variable by the appropriate use of **command substitution**.

```
x=`expr 2 + 4`
```

64

Ch. 4. The Bourne Shell

- **ARITHMETIC**

- **expression** may be constructed by applying the following binary operators to integer operands, grouped in decreasing order of precedence:

OPERATOR	RESPECTIVE MEANING
* / %	multiplication, division, remainder
+ -	addition, subtraction
=> >= < <= !=	comparison operators
&	logical and
	logical or

65

Ch. 4. The Bourne Shell

- The following example illustrates some of the functions of **expr** and makes plentiful use of command substitution:

```
$ x=1          ---> initial value of x.
$ x=`expr $x + 1`  ---> increment x.  x = x+1
$ echo $x
2

$ x=`expr 2 + 3 \* 5`  ---> * is conducted before +.
$ echo $x
17
```

Ch. 4. The Bourne Shell

• CONDITIONAL EXPRESSIONS

- The control structures often branch based on the value of a logical expression—that is, an expression that evaluates to true or false.

The `test` utility supports a substantial set of UNIX-oriented expressions suitable for most occasions and works like this:

Utility: test expression
[expression]

`test` returns a zero exit code if expression evaluates to true; otherwise, it returns a nonzero exit status.

```
sh-3.00$ test 3 -eq 3 ; echo $?    0   true
sh-3.00$ test 3 -eq 33 ; echo $?   1   false
```

The exit status is typically used by shell control structures for branching purposes. `if test $x -eq 3`

Some **Bourne shells** supports `test` as a built-in command, in which case they support the second form of evaluation as well. []

Ch. 4. The Bourne Shell

- The brackets of the second form must be surrounded by spaces in order for it to work. [2 -eq 4]

A `test` expression may take the following forms:

Form	Meaning
-b filename	True if filename exists as a block special file.
-c filename	True if filename exists as a character special file.
-d filename	True if filename exists as a directory.
-f filename	True if filename exists as an ordinary file
-g filename	True if filename exists as a "set group ID" file.
-h filename	True if filename exists as a symbolic link.
-k filename	True if filename exists and has its sticky bit set.
-p filename	True if filename exists as a named pipe.
-u filename	True if filename exists as a "set user ID" file.
-s filename	True if filename contains at least 1 char (none empty)

Ch. 4. The Bourne Shell

Form	Meaning
-t fd	True if file descriptor fd is associated with a terminal.
-r filename	True if filename exists as a readable file.
-w filename	True if filename exists as a writeable file.
-x filename	True if filename exists as an executable file.
-l string	True if length of string is nonzero.
-n string	True if string contains at least one character.
-z string	True if string contains no characters. empty string
str1 = str2	True if str1 is equal to str2.
str1 != str2	True if str1 is not equal to str2.
string	True if string is not null.
int1 -eq int2	True if integer int1 is equal to integer int2.
int1 -ne int2	True if integer int1 is not equal to integer int2.
int1 -gt int2	True if integer int1 is greater than integer int2.
int1 -ge int2	True if integer int1 is greater than or equal to integer int2.
int1 -lt int2	True if integer int1 is less than integer int2.
int1 -le int2	True if integer int1 is less than or equal to integer int2.
! expr	True is expr is False

69

Argument	Test is true if . . .
-d file	file is a directory
-f file	file is an ordinary file
-r file	file is readable
-s file	file size is greater than zero
-w file	file is writable
-x file	file is executable
! -d file	file is not a directory
! -f file	file is not an ordinary file
! -r file	file is not readable
! -s file	file size is not greater than zero
! -w file	file is not writable
! -x file	file is not executable
n1 -eq n2	integer n1 equals integer n2
n1 -ge n2	integer n1 is greater than or equal to integer n2
n1 -gt n2	integer n1 is greater than integer n2
n1 -le n2	integer n1 is less than or equal to integer n2
n1 -ne n2	integer n1 is not equal to integer n2
n1 -lt n2	integer n1 is less than integer n2
s1 = s2	string s1 equals string s2
s1 != s2	string s1 is not equal to string s2

70

Ch. 4. The Bourne Shell

- CONTROL STRUCTURES

- The Bourne shell supports a wide range of control structures that make it suitable as a high-level programming tool.

Shell programs are usually stored in scripts and are commonly used to automate maintenance and installation tasks.

Branch: **If then** **elif then** **else fi** **case..in..esac**

Loop: **While do done** **For do done** **until do done**



71

Ch. 4. The Bourne Shell

- **if...then...fi**

The *if* command supports **nested conditional branches** and has the following syntax:

```
if list1
then
  list2
elif list3 ---> optional,
               the elif part may be repeated several times.
then
  list4
else ---> optional,
               the else part may occur zero times or one time.
fi
```

72

Ch. 4. The Bourne Shell

- The `if` command works as follows:

The commands in `list1` are executed.

If the last command in `list1` succeeds, the commands in `list2` are executed.

If the last command in `list1` fails and there are one or more `elif` components, then a successful command list following an `elif` causes the commands following the associated `then` to be executed.

If no successful lists are found and there is an `else` component, the commands following the `else` are executed.

73

Ch. 4. The Bourne Shell

- Here's an example of a script that uses an `if` control structure:

```
$ cat numberScript.sh
echo -n "enter a number: "
read number
if [ $number -lt 0 ] # if test $number -lt 0
then # if ! ([ $number -gt 0 ] || [ $number -eq 0 ])
    echo negative
elif [ $number -eq 0 ]
then
    echo zero
else
    echo positive
fi
```

\$ `numberScript.sh` ---> run the script.

enter a number: 1
positive

\$ `numberScript.sh` ---> run the script again.

enter a number: -1
negative
\$-

74

Ch. 4. The Bourne Shell

- Here's another example of a script that uses an *if* control structure

```
$ cat week.sh
echo -n "enter a date: "
read date
if [ $date = Fri ]      # if [ $date = "Fri" ] compare string easy
then
    echo "Thank God it is Friday"
elif [ $date = Sat ] || [ $date = Sun ]
then
    echo "You should not be here working, go home!!!"
else
    echo "Not weekend yet. Get to work"
fi
```

Compare string is easier



```
$ week.sh      ---> run the script.
enter a date: Wed
Not weekend yet. Get to work
$ week.sh      ---> run the script again.
enter a date: Fri
Thank God it is Friday
$-
```

75

Ch. 4. The Bourne Shell

-Here's an example of a script that uses an *if* control structure:
-If a file empty, echo, else Is it

```
$ cat if.sh
echo -n "enter a file name: "
read name
if [ ! -s $name ]
then
    echo "File $1 is empty"
    exit 1
else
    ls -l $name
fi
```

Argument	Test is true if . . .
-d <i>file</i>	<i>file</i> is a directory
-f <i>file</i>	<i>file</i> is an ordinary file
-r <i>file</i>	<i>file</i> is readable
<u>-s <i>file</i></u>	<u><i>file</i> size is greater than zero</u>
-w <i>file</i>	<i>file</i> is writable
-x <i>file</i>	<i>file</i> is executable

-s filename True if filename contains at least 1 char (none empty)

76

Ch. 4. The Bourne Shell

- Here's an example of a script that uses an *if* control structure:

```
$ cat if.sh ---> list the script.
```

```
echo -n "enter a file name: "  
read name  
if [ -d $name ]          # if [ $number -lt 0 ] || [ $number -eq 0 ]  
then  
    echo $name is a directory  
elif [ -x $name ]  
then  
    echo "File $name is executable"  
else  
    echo "File $name is not executable"  
    chmod +x $name  
    echo "File $name is executable now"  
fi
```

Argument	Test is true if . . .
-d file	file is a directory
-f file	file is an ordinary file
-r file	file is readable
-s file	file size is greater than zero
-w file	file is writable
-x file	file is executable

77

Ch. 4. The Bourne Shell

- Here's an example of a script that uses an *if* control structure:
improved version of grep

```
$ cat grepNotFound.sh ---> list the script.
```

```
# arg1 search pattern  
# arg2 file to search  
egrep $1 $2  
if [ $? -ne 0 ]          # not 0 --- not successful [ $? gt 0 ]  
then  
    echo pattern $1 not found  
fi
```

```
$ grepNotFound.sh hui classlist2031  
pattern hui not found
```

How to check if found or not?

Ch. 4. The Bourne Shell

- CONTROL STRUCTURES

case..in..esac

The **case** command supports multiway branching based on the value of a single string and has the following syntax:

```
case expression in
  pattern { |pattern} *)
  list
;;
esac
```

79

Ch. 4. The Bourne Shell

- expression is an expression that evaluates to a string, pattern may include wildcards, and a list of one or more shell commands.

You may include as many pattern/list associations as you wish.

The shell evaluates expression and then compares it to each pattern in turn, from top to bottom.

When the first matching pattern is found, its associated list of commands is executed and then the shell skips to the matching **esac**.

A series of patterns separated by "or" symbols(|) are all associated with the same list.

If no match is found, then the shell skips to the matching **esac**.

80

Ch. 4. The Bourne Shell

- **while...done**

- The *while* command executes one series of commands as long as another series of commands succeeds.

Here's its syntax:

```
while list1
do
    list2
done
```

The *while* command executes the commands in *list1* and ends if the last command in *list1* fails; otherwise, the commands in *list2* are executed and the process is repeated.

81

Ch. 4. The Bourne Shell

- If *list2* is empty, the *do* keyword should be omitted.
A **break** command causes *the loop to end immediately*, and a **continue** command causes *the loop to immediately jump to the next iteration*.
- Here's an example of a script that uses a *while control structure* to generate a small multiplication table:

\$ **cat repeat.sh**

```
count=1
while [ $count -lt 5 ]
do
    echo Hello
    count=`expr $count + 1`
done
```

```
sh-4.1$ repeat.sh
Hello
Hello
Hello
Hello
sh-4.1$
```

82

Ch. 4. The Bourne Shell

Write a script `matrix.sh` to generate a matrix

```
$ matrix.sh 7
```

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28
5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

```
$ -
```

\$1

```
x = 1
while (x < 7)
{
  y = 1
  while (y < 7)
  {
    print y * x
    y = y + 1
  }
  x = x + 1
}
```

83

Ch. 4. The Bourne Shell

```
$ cat matrix.sh ---> list the script.
```

```
$ multi.sh 4
```

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16

```
if [ -z $1 ]; then
  echo "Usage: matrix number"
  exit
fi
x=1 # set outer-loop value
while [ $x -le $1 ] # outer loop
do
  y=1 # set inner-loop value
  while [ $y -le $1 ] # generate one table entry.
  do
    echo -n `expr $x \* $y` " " # Indent not mandatory
    y=`expr $y + 1` # y++ update inner-loop count
  done
  echo
  x=`expr $x + 1` # x++ update inner-loop count
done
```

```
x = 1
while (x < 7){
  y = 1
  while (y < 7){
    print y * x
    y = y + 1
  }
  x = x + 1
}
```

84

Ch. 4. The Bourne Shell

- Here's an example of a script called "menu.sh" that makes use of **while** and **case** control structure:

```
#!/bin/sh
echo menu test program

stop=0 # reset loop-termination flag.

while test $stop -eq 0 # while [ $stop -eq 0 ] loop until done.
do
  cat << ENDOFMENU # display menu.
  1 : print the date.
  2,3 : print the current working directory.
  4 : exit
  ENDOFMENU

  echo
  echo -n 'your choice? ' # prompt.
  read reply # read response.
  echo
```

85

Ch. 4. The Bourne Shell

```
case $reply in
  "1") # process response.
    date # display date.
    ;;
  "2"|"3") # display working directory.
    pwd
    ;;
  "4") # set loop termination flag.
    stop=1
    ;;
  *) # default.
    echo illegal choice # error.
esac
done
```

86

Ch. 4. The Bourne Shell

- Here's the output from a sample run of the "menu.sh" script:

```
$ menu.sh
menu test program
 1 : print the date.
2,3 : print the current working directory.
 4 : exit
your choice? 1
Thu Feb 5 07:09:13 CST 1998
 1 : print the date.
2,3 : print the current working directory.
 4 : exit
your choice? 2
/home/glass
 1 : print the date.
2,3 : print the current working directory.
 4 : exit
your choice? 5
```

87

Ch. 4. The Bourne Shell

```
illegal choice
 1 : print the date.
2,3 : print the current working directory.
 4 : exit
your choice? 4
$ -
```

88

Ch. 4. The Bourne Shell

- **for..do..done**

- The **for** command allows a list of commands to be executed several times, using a different value of the loop variable during each iteration.

Here's its syntax:

```
for name [ in {word}* ]
do
    list
done
```

The **for** command loops the value of the variable **name** through each word in the word list, evaluating the commands in list after each iteration.

89

Ch. 4. The Bourne Shell

- If no word list is supplied, **\$((\$1...))** is used instead. A **break** command causes the loop to immediately end, and a **continue** command causes the loop to immediately jump to the next iteration.

Here's an example of a script that uses a **for** control structure:

```
$ cat for.sh
for color in red yellow green blue
do
    echo current color is $color
done
```

```
$ for.sh
current color is red
current color is yellow
current color is green
current color is blue
```

```
$ -
```

90

Ch. 4. The Bourne Shell

- If no word list is supplied, `$(($1...))` is used instead.

Here's an example of a script that uses a *for* control structure:

```
$ cat for2.sh
for color in $*
do
    echo current color is $color
done
```

All the command line input

```
$ for2.sh red yellow green blue black
current color is red
current color is yellow
current color is green
current color is blue
current color is black
$ -
```

Example: Loop and branch

```
#!/bin/sh
echo -n "enter a number or 'q': "
read number

while [ $number != q ]           # while [ number != "q" ]
do
    if [ $number -lt 0 ]
    then
        echo "a negative number"
    elif test $number -eq 0
    then
        echo " this is zero"
    else
        echo " a positive number"
    fi

    # read again
    echo -n "enter a number: "
    read number
done
```

Ch. 4. The Bourne Shell

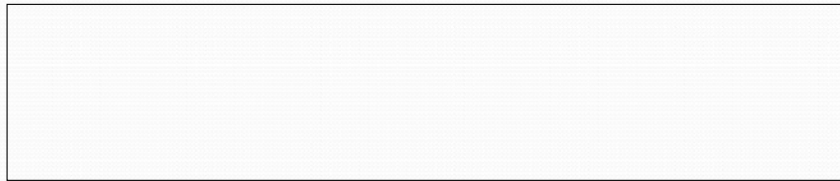
• CONTROL STRUCTURES

- The Bourne shell supports a wide range of control structures that make it suitable as a high-level programming tool.

Shell programs are usually stored in scripts and are commonly used to automate maintenance and installation tasks.

Branching: `if` `elif` `else` `fi` `case` `..in` `..esac`

Looping: `While` `do` `done` `For` `do` `done` `until` `do` `done`



93

Ch. 4. The Bourne Shell

□ `until...do...done`

- The `until` command executes one series of commands as long as another series of commands fails and has the following syntax:

```
until list1
do
  list2
done
```

- The `until` command executes the commands in `list1` and ends if the last command in `list1` succeeds; otherwise, the commands in `list2` are executed and the process is repeated.
- If `list2` is empty, the `do` keyword should be omitted. A `break` command causes the loop to immediately end, and a `continue` command causes the loop to immediately jump to the next iteration.

94

Ch. 4. The Bourne Shell

- Here's an example of a script that uses an `until` control structure:

```
$ cat until.sh ---> list the script.
```

```
x=1
until [ $x -gt 3 ]
do
  echo x=$x
  x=`expr $x + 1`
done
```

```
$ until.sh ---> execute the script.
```

```
x=1
x=2
x=3
$ -
```

95

Some small but useful topics in shell scripting

- Looping: [1] / true, break, continue, ;
- Command line: set, shift
- function
- dev/null/ and redirection

Ch. 4. The Bourne Shell

-while true, break, continue, and :

```
#!/bin/sh
echo menu test program

while [ 1 ] # while true
do
  echo -n 'your choice?' # prompt.
  read reply # read response.
  if [ $reply = q ]
  then
    break
  fi
  .....
done
```

```
x=0
while true
do
  x=`expr $x + 1`
  if [ $x -eq 100 ]
  then
    break
  else
    : # do nothing
  fi
done
echo $x
```

99

97

Command Line Arguments -- shift, set

- Command line arguments stored in variables called **positional parameters**.
- These parameters are named **\$1** **\$9**.
- Command itself is in parameter **\$0**.
- In diagram format:

command	arg1	arg2	arg3	arg4	arg5	arg6	arg7	arg8	arg9
\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9
- Note: **\$#** does NOT include the program name (unlike **argc** in C programs)

98

Command Line Arguments

- What if the number of arguments is more than 9?
How to access the 10th, 11th, etc.?
- Use **shift** operator.

99

shift Operator

- **shift** promotes each argument one position to the left.
- Operates as a conveyor belt.
- Allows access to arguments beyond \$9.
 - shifts contents of \$2 into \$1
 - shifts contents of \$3 into \$2
 - shifts contents of \$4 into \$3
 - etc.
- Eliminates argument(s) positioned immediately after the command.
- Syntax:
shift # shifting arguments one position to the left
- After a shift, the argument count stored in \$# is automatically decremented by one.

100

Example 1

```
% cat args
#!/bin/sh
echo "arg1 = $1, arg2 = $2, arg9 = $9, ARGC = $#"  
echo $@  
shift  
echo "arg1 = $1, arg2 = $2, arg9 = $9, ARGC = $#"  
echo $@  
  
% args 1 2 3 4 5 6 7 8 9 10 11 12  
arg1 = 1, arg2 = 2, arg9 = 9, ARGC = 12  
1 2 3 4 5 6 7 8 9 10 11 12  
arg1 = 2, arg2 = 3, arg9 = 10, ARGC = 11  
2 3 4 5 6 7 8 9 10 11 12
```

101

Example 2

```
% cat show_shift
#!/bin/sh
echo "arg1=$1, arg2=$2, arg3=$3"  
shift  
echo "arg1=$1, arg2=$2, arg3=$3"  
shift  
echo "arg1=$1, arg2=$2, arg3=$3"  
  
% show_shift William Richard Elizabeth  
arg1=William, arg2=Richard, arg3=Elizabeth  
arg1=Richard, arg2=Elizabeth, arg3=  
arg1=Elizabeth, arg2=, arg3=
```

102

Example 3

```
% my_copy dir_name filename1 filename2 filename3 ...
• mycopy lab8b f1 f2 f3 f4

# This shell script copies all the files to
  directory "dir_name" --- 1st argument

% cat my_copy
#!/bin/sh
directory=$1
shift
files=$* # f1 f2 f3 f4
cp $files $directory # or cp $* $directory
```

103

Shifting Multiple Times

Shifting arguments three positions: 3 ways to write it

```
shift
shift
shift
```

```
shift; shift; shift
```

```
shift 3
```

104

set: Changing Values of Positional Parameters

- Positional parameters `$1`, `$2`, ... normally store command line arguments.
- Their values can **not** be set or changed using `=` command, `$1=2` wrong
- Their values can be changed using `set` command, for example, `set `date``
- The new values are the output of `date` command.

105

Example

```
sh-3.00$ set a b c d e
sh-3.00$ echo $@
a b c d e
sh-3.00$ echo $1 $2 $3 $5
a b c e
sh-3.00$ shift
sh-3.00$ echo $@    ?
b c d e
```

106

Example set: A more descriptive we

- `%wc filename`
`5 17 84 filename`

- `cat myWC.sh`

```
set `wc $1`      # 5:$1 17:$2 84:$3 filename:$4
echo "File: $4"
echo "Lines: $1"
echo "Words $2"
echo "characters: $3"
```

```
%myWC.sh filename
```

```
File: filename
```

```
Lines: 5
```

```
Words: 17
```

```
Characters: 84
```

Shell Functions

- Similar to shell scripts.
- Stored in shell where it is defined (instead of in a file).
- Executed within **sh**
 - no child process spawned

- Syntax:

```
function_name()  
{  
    commands  
}
```

- or `function function_name`

- Allows structured shell scripts

Example

```
#!/bin/sh
# function to sample how many users are logged on

log()
{
    echo "Users logged on:" >> users
    date >> users
    who >> users
    echo "\n\n" >> users
}

# taking first sample
log

# taking second sample (30 min. later)
sleep 1800
log
```

109

Itenable example

```
#!/bin/sh
# Enable labtest.
error() {
    echo "error: $1"
    exit 1
}
usage() {
    echo "usage: $0 <full path to labtest dir>"
    exit 1
}

if [ $# -ne 1 ]; then
    usage
fi
```

How to pass and receive parameters/arguments ?

Itenable example (con't)

```
LTDIR=$1
/xsys/pkg/pcmode/bin/ltfix -q $LTDIR

if [ $? -ne 0 ]; then
    error "Labtest \"$LTPATH\" is NOT enabled."
fi

chmod 750 $LTDIR
if [ $? -ne 0 ]; then
    error "chmod on $LTDIR failed. Labtest unabled."
fi

echo "Labtest \"$LTDIR\" is enabled."
```

Argument to the function
in function, this is referred to as \$1

```
#!/bin/sh
```

```
sum() {
x=`expr $1 + $2`
echo $x
}
```

How to pass and receive
parameters/arguments ?

```
x=1
sum 5 3
echo "The sum of 4 and 7 is `sum 4 7`"
a=`sum 2 10`
echo a is $a
```

Arguments to the function
in function, 5 is referred to as \$1, and 3 is referred to as \$2

8

The sum of 4 and 7 is 11

12

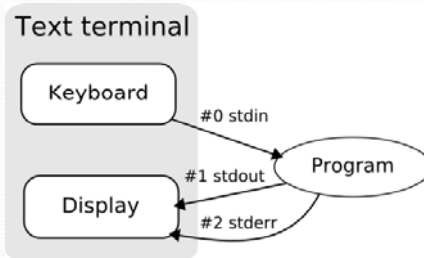
a is 12

/dev/null - bit bucket, black hole, Dave Null

- Special file found in device directory (/dev)
 - Discard data written to (but report writing succeed)
 - Write to it -- everything disappears (absorbed)
 - Provide no data to reading process (EOF immediately)
 - Read from it – get nothing
-
- `ls -l > /dev/null` # standard output is suppressed
 - Create an empty file / clear a file
 1. `cat < /dev/null > emptyF`
 2. `cat /dev/null > emptyF`
 3. `echo /dev/null > emptyF` ???
 4. Another way: `touch emptyF`

Enhanced I/O Redirection of Control Structures

- `ls xx 1> logfile`
- `ls xx 2> logfile`
- `ls xx 2> /dev/null`
- `ls xx 2>&1`
 - Send 2 to same place as 1
- `ls xx > logfile 2>&1`
- `ls xx > /dev/null 2>&1`
- `ls xx 2>&1 > /dev/null`



Handle	Name	Description
0	stdin	Standard input
1	stdout	Standard output
2	stderr	Standard error

114

```
# arg1 search pattern
# arg2 file to search
egrep $1 $2
if [ $? -ne 0 ]      # not 0 --- not successful
then
  echo pattern $1 not found
fi
```

↓ Suppress all the outputs from grep

```
# arg1 search pattern
# arg2 file to search
egrep $1 $2 > /dev/null 2>&1
if [ $? -ne 0 ]      # not 0 --- not successful
then
  echo pattern $1 not found
else
  echo pattern $1 found
fi
```

New small topics about script (summary)

- while [1] break
- set and shift
- function in scripting
- dev/null,
- redirection 2>&1

Quick review and addition of C

- Topics that is small but it is better that you know
 - const
 - enum
 - union
 - Library funciton, e.g., memset
 - self-referential structures
 - system calls (C + Unix)

Type, size, variables,

- Two more ways for constant
 - `# define MAX_LENGTH 30`

By default, start from 0
 - `enum boolean {NO, YES} /* NO 0 YES 1 */`
 - `enum months {Jan=1, Feb, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC}`
 - JAN = 1, FEB is 2, MAR is 3
- You can specify values if you want - C fills in the rest
 - `enum col { RED = 1, BLUE, GREEN = 16, BROWN};`
 - `enum DAY { saturday, sunday = 0, monday, tuesday, wednesday, thursday, friday } workday;`

2

17

1

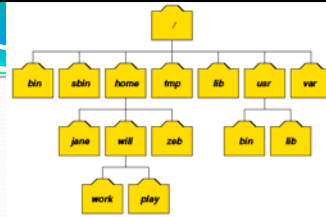
2

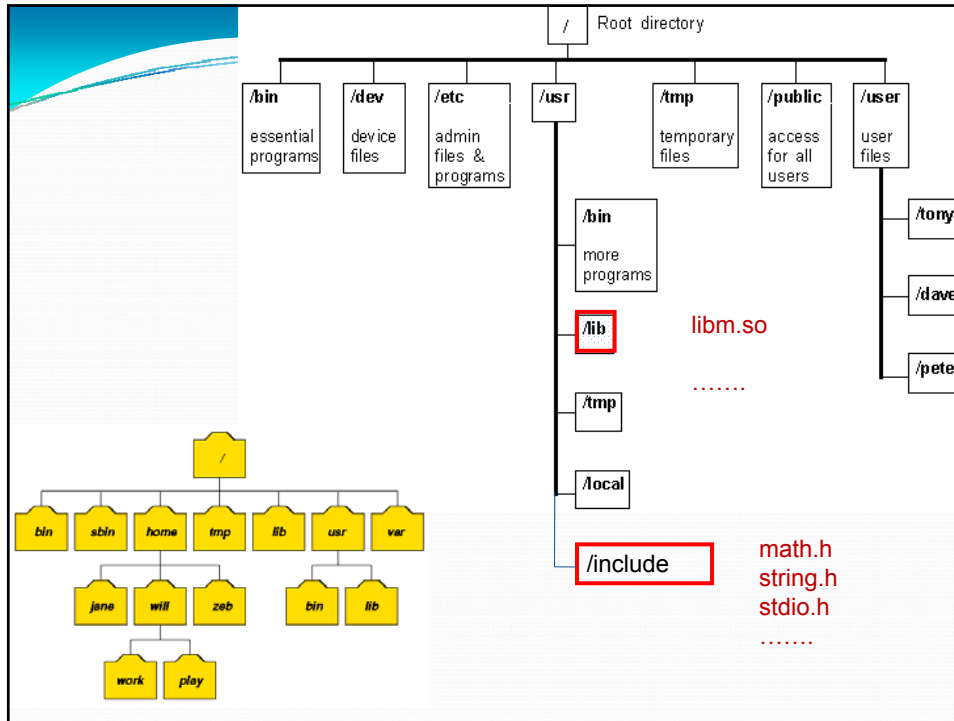
Type, size, variables,

- Two more ways for constant
- `# define MAX_LENGTH 30`
- `enum boolean {NO, YES} /* NO 0 YES 1 */`
- `enum months {Jan=1, Feb, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC}`
 - JAN = 1, FEB is 2, MAR is 3
- `const int MAX_LENGTH = 30;`
- `const double e = 2.71828182835905;`
- `int strlen (const char *s) # fun will not change s`

function

- To use a function, need two things
 - Declaration (prototype)
 - Have implementation (the code)
- `#include <math.h> # /usr/include/math.h`
 - header file contains declaration (prototype)
- `gcc file.c -lm`
 - link math library `# /usr/lib/libm.so`





Function -- old style definition --- good to know

```

type fun (parm1, parm2, .. parmN)
type parm1
type parm2
.
type parmN
{ function code
}

```

```

main(argc,argv)
int argc;
char *argv[];
{...}
}

```

```

# float func (int a, int b, char ch){...}
float func (a, b, ch)
int a, b;
char ch;
{ ...
}

```

```

explore(world, myworld, nbeacons)
Graph *world, *myworld;
int nbeacons;
{
RobotState state, mystate;
int beacons[MAXBEACONS], beaconlink[MAXBEACONS];
int shouldvisit[MAXDEGREE];
char buf[80];

/*-*/
int dists[MAXNODES];
}

```

Array and functions

- `int arr[];` **X**
- `int arr[] = {1,1,2,3,3,9};`
- `int arr2D[][3] = {1,2,3,4,4,5}`

Give some or all
the sizes !

- `function(int [2])`
- `function(int [])` **X**

- `function (int [2][3] str)`
- `function (int [][][3] str)`
- `function (int [][] str)` **X**

Pointer, structure

- Array (string) can not be operated as whole. `arr1 = arr2` **X**
- Others could be copied
 - E.g., structure, and pointer `s1 = s2` `p1 = p2`
 - Copy a structure is different from Java -- a separate copy

- Beware when accessing members:

`*d.width` --- incorrect **X**

- '·' is higher precedence than '*'

`(*d).width` --- correct

`d->width`

Trouble with Pointers

- Pointers (vs. arrays) and dynamic memory management are the largest source of crashes and errors in C programs
- All of the problems deal with losing track of pointers and dynamically allocated memory

segmentation fault
core dump



127

Be extra careful with pointers!

Common errors:

- Uninitialized pointers
- Overruns and underruns
 - Occurs when you reference a memory beyond what you allocated.

128

Problems with pointers

```
int *ptr; /* I'm a pointer to an int */  
ptr= &rate; /*I got the address of rate */  
*ptr = 5 /* contents of the pointee*/
```



```
int *ptr; /* I'm a pointer to an int */  
*ptr = 5; /* contents of the pointee is 5*/
```



- Ptr is uninitialized. Has some value but don't know
- Pointing to sth **unknown**, may be your os!
- Always make it point to sth! Ptr = & rate

129

problem with pointers

```
char name[6];  
char *name2;  
int age; double wage;  
  
printf("input name, name2, age, wage");  
scanf("%s %s %d %f",name,name2,age,wage);  
while( strcmp(name, "xx") )  
.....  
}
```



segmentation fault ! Why?

130

Whenever you need to access its "pointee"

Ask you self: Have you done one of this

1. `ptr = & var. /* direct */`
 - **var must not be a local variable**
2. `ptr = ptr2 /* indirect */`
3. `ptr = (..)malloc(....)`

- You do need to do one of above for these
 - `*ptr = var`
 - `prt[2] = var`
 - `scanf("%s", ptr)`
 - `strcpy(ptr, "hello")`
 - `fgets(ptr, .)`
 -

131

Whenever to use malloc ?

- When you need to allocate memory in run time.
- When you need memory space throughout the program running

1. `ptr = & var. /* direct */`

- **var is a local variable**



2. `ptr = ptr2 /* indirect */`

- **Ptr2 point to a local variable**



- var is in **stack**. Not in heap.

132

```

void setArr (int);

int * arr[10]; // array of 10 int pointers

setArr2
int main(int argc, char *argv[])
{
    int i;

    setArr(0);
    setArr(1);

    for(i=0; i<2;i++)
        printf("arr [%d] = %d\n", i, *arr[i]); /*
return 0;
}

/* set arr[index], which is a pointer, to point to
void setArr (int index){
    int i = 2 * index;
    arr[index] = &i;
}


```

X

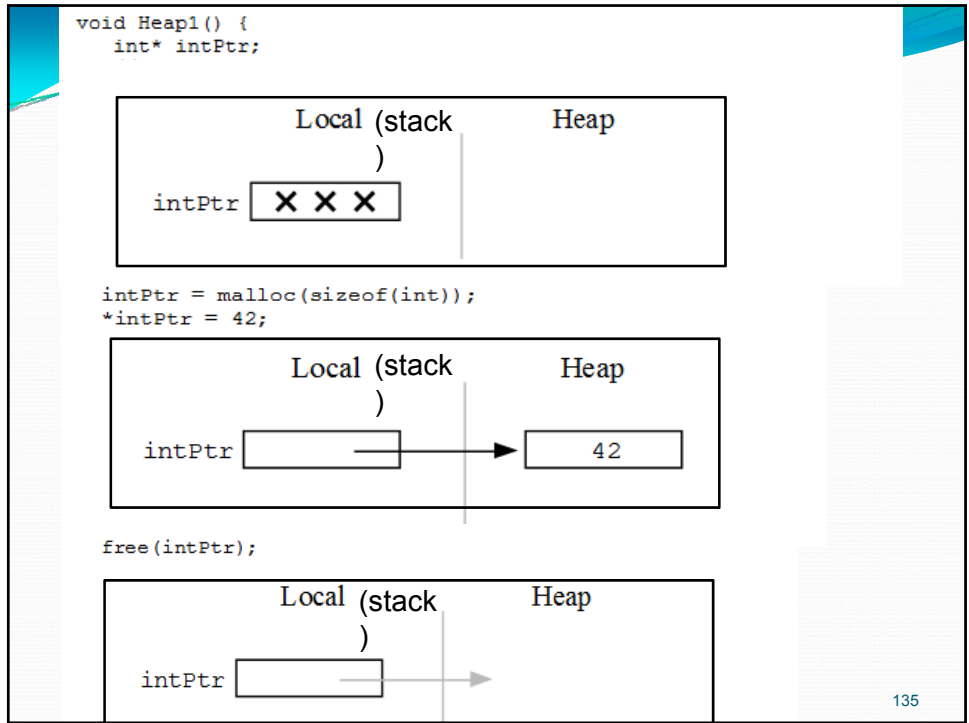
133

Stack vs. heap

- Local (**stack**) memory, automatic
 - Allocated on function call, and deallocated automatically when function exits
- Dynamic **heap** memory
 - Not deallocated when function exits!
 - Request a heap memory: **malloc/calloc** in C, **new** in Java
 - **free()** to deallocated in C, garbage collection in Java
- The heap is an area of memory available to allocate areas ("blocks") of memory for the program.


cool

134



```

void setArr (int);

int * arr[10]; // array of 10 int pointers

setArr2
int main(int argc, char *argv[])
{
    int i;

    setArr(0);
    setArr(1);

    for(i=0; i<2;i++)
        printf("arr [%d] = %d\n", i, *arr[i]); /*
    return 0;
}

/* set arr[index], which is a pointer, to point to
void setArr (int index){
    int i = 2 * index;
    arr[index] = &i;
}

```

X

i is in stack --
deallocated
when fun exits

136

setArr3

```

void setArr (int);

int * arr[10]; // array of 10 int pointers

int main(int argc, char *argv[])
{
    int i;

    setArr(0);
    setArr(1);

    for(i=0; i<5;i++)
    | printf("arr [%d] = %d\n", i, *arr[i]);
    return 0;
}

/* set arr[index], which is a pointer, to point to
void setArr (int index){
    int i = 2 * index;
    arr[index] = (int *) malloc (sizeof(int));
    * arr[index] = i;
}

```

137

```

void enter()
{
    char name[LEN];
    int age;
    char cour1[LEN], cour2[LEN];

    printf("name: "); scanf("%s", name);
    printf("age: ");   scanf("%d", &age);
    printf("course-1: "); scanf("%s", cour1);
    printf("course-2: "); scanf("%s", cour2);

    record newStudent;
    strcpy(newStudent.name, name);
    newStudent.age = age;
    strcpy(newStudent.course1, cour1);
    strcpy(newStudent.course2, cour2);

    database[count++] = &newStudent;
}

```

```

void enter()
{
    char nam[LEN];
    int ag;
    char cour1[LEN], cour2[LEN];

    printf("name: ");      scanf("%s", nam);
    printf("age: ");       scanf("%d", &ag);
    printf("course-1: ");  scanf("%s", cour1);
    printf("course-2: ");  scanf("%s", cour2);

    arr[count] = (record *)malloc (sizeof(record)) ;

    strcpy( arr[count] -> name, name)
    arr[count] -> age = ag;
    strcpy(arr[count] -> course1, cour1);
    strcpy(arr[count] -> course2, cour2);
}

```

Assignment 2

```

void writeDisk(void){
    FILE *fp; int i;
    if ( (fp=fopen(diskFile,"ab")) == NULL){
        fprintf(stderr, "cannot open file\n");
        return;
    }
    for (i=0; i< SIZE ; i++){
        if ( database[i] != NULL)
            { if (fwrite( database[i], sizeof(struct
people), 1, fp) != 1)
                { fprintf(stderr, "file write error\n");
                }
            }
    }
    fclose(fp);
}

```

```

void loadDisk(void){
    FILE *fp; int i;
    struct inv_type * tmp;

    if ( (fp = fopen(diskFile,"r")) == NULL){
        fprintf(stderr,"cannot open file\n"); return; }

    for (i=0; i < SIZE ; i++){
        tmp = (struct inv_type *) malloc (sizeof(struct
inv_type));
        if (fread( tmp, sizeof(struct inv_type), 1, fp) != 1)
            {
                if (feof(fp))
                { fclose(fp);
                return;
                }
                printf("file read error\n");
            }
        else invtry[i] = tmp;
    }
}

```

Binary I/O fseek, ftell, rewind...

- For binary files there is also
 - `fseek(FILE *stream, long offset, int origin);`
- where origin is one of
 - SEEK_SET (relative to beginning of file),
 - SEEK_CUR (relative to current position),
 - SEEK_END (relative to end of file).
- `ftell(FILE *stream)`
 - Tell the offset (from beginning, in byte)
- `rewind(FILE *stream)`
 - is equivalent to `fseek(f,0,SEEK_SET);`

```

int main(int argc, char *argv[]){
    FILE *fp;
    long p; char line [20];
    if ( (fp = fopen("lines.txt","rt")) == NULL ){
        fprintf(stderr, "cannot open file");
        exit(1)}

    fgets(line, 20, fp);
    fgets(line, 20, fp);
    p = ftell(fp); printf("%d\n", p); // 30

    fseek(fp, 20, SEEK_CUR);
    p = ftell(fp); printf("%d\n", p); // 50

    fgets(line, 20, fp);
    fputs(line, stdout); // is line 4

    fseek(fp, 0, SEEK_SET); // rewind();
    p = ftell(fp); printf("%d\n", p); // 0

    fgets(line, 20, fp); fputs(line, stdout); // this is line 1

    fseek(fp, 15, SEEK_CUR);
    fgets(line, 20, fp); fputs(line, stdout); // this is line 3
    fclose(fp);

```

Each line 15 chars

```

this is line 1
this is line 2
this is line 3
this is line 4
this is line 5
this is line 6
this is line 7
this is line 8
this is line 9
this is line 10

```

fseek + ftell rewind

```

#include <stdio.h>

int main(int argc, char **argv) {
    FILE *file_handle;
    long int file_length;

    file_handle = fopen("file.bin","rb");
    if(fseek(file_handle, 0, SEEK_END)) {
        puts("Error while seeking to end of file");
        return 1;
    }
    file_length = ftell(file_handle);
    if (file_length < 0) {
        puts("Error while reading file position");
        return 2;
    }

    printf("File length: %d bytes\n", file_length);
    fclose(file_handle);
    return 0;
}

```

Other topics that we did not get to cover

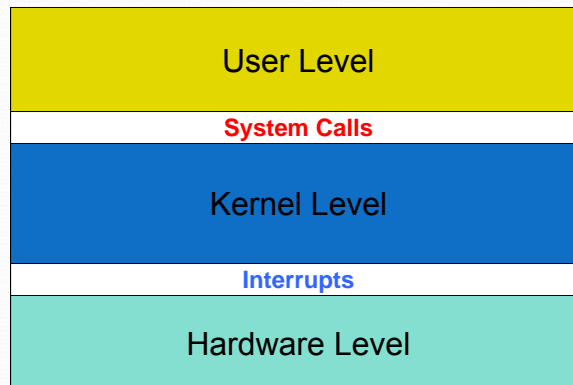
-- might be useful in your future studies

- Union
- Pointer to whole arrays, `int (*arr) []` `[][]` decayed to
 - `int (*arr) [8];` // A pointer to an array of integers
 - We have seen: `int* arr[8];` // An array of int pointers.
- Pointer to functions `void (*fptr) (int)`
- System calls (fork, pipe ... read, write)
 - You will see if you take CSE3221 operating systems.
- Others
 - Make file
 - dbg

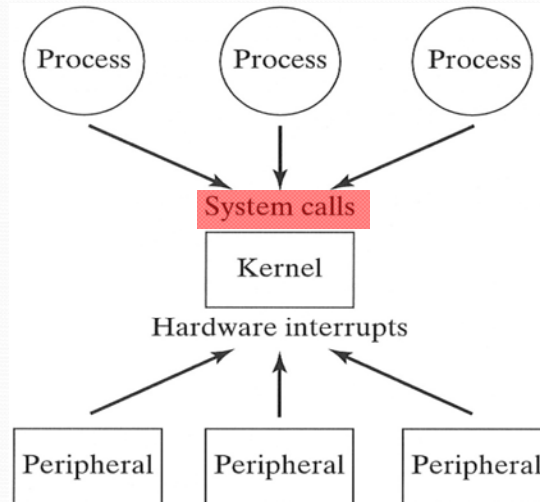
last topic:

C + Unix -> System calls (ch8)

- You may never use it, but good to know
- Processes accesses kernel facilities via **system calls**
- Peripherals communicate with the kernel via **hardware interrupts**



Talking to Kernel



System calls vs. library subroutines

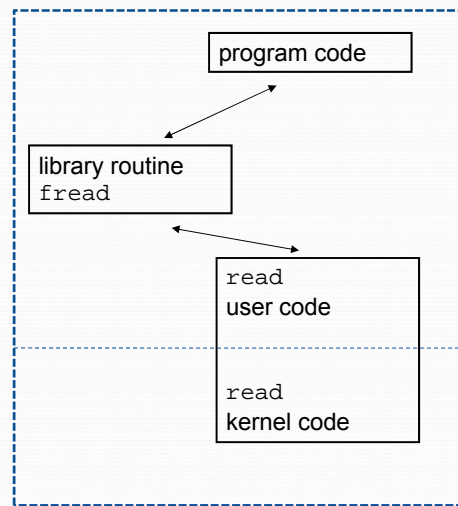
Read data, two ways

```
fopen (...)  
fread (...)
```

```
open (...)  
read (...)
```

Standard IO (e.g., `fopen/ fread`)
eventually call system call (e.g.,
`open/read`)
Act as an interface for system calls

Unix utilities/ shell command call
system call
`cat, ls, |`



High-Level vs. Low-Level

Your program	<code>read_dict("words")</code>
C Library	<code>fopen("words","r")</code>
Unix interface	<code>open("words",O_RDONLY)</code>
kernel	<i>actual work is done here</i>

System calls are functions under Unix which are implemented in the kernel
See section 2 of the manual pages

149 your program calls this function in C, but the kernel does the real work

High-Level vs. Low-Level

Your program	
C Library	<code>fopen, fread, fwrite, fclose, remove, fseek</code>
Unix interface (System call)	<code>open, read, write, close, unlink, lseek</code> and more
kernel	<i>actual work is done here</i>

150

Some things to discuss

- Lab exercise for Bourne shell script posted (lab 9)
- Sample shell scripting programs
- a3
- Give me your lab 8.
- Final exam...?
- Midterm papers
- I am still around -- email me if you need my help

That all for the course

- Sometimes you may have suffer, I tried my best to avoid or at least minimize this, but
- Learning C is not an easy task, and this course has never been an easy course.
- Hope you find the course useful
- Thank you for your supports
- And enjoy the coming spring & summer!



That all for the course

- Thank you for your supports
- And enjoy the coming spring & summer!



so long ...