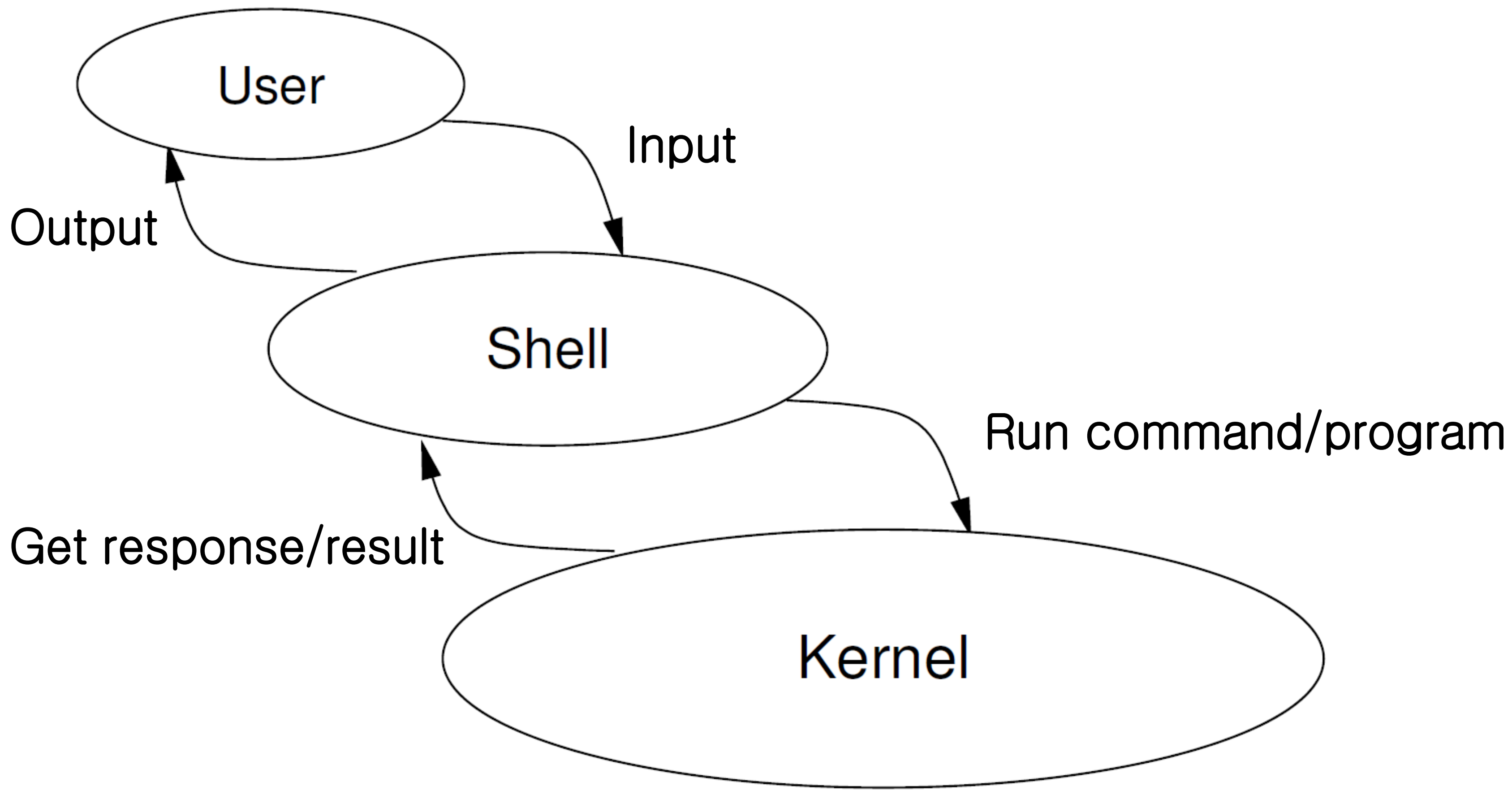
Shell

Chonnam National University
School of Electronics and Computer Engineering

Kyungbaek Kim

Hierarchy of Linux program



Kernel and User Program

- Kernel
  - The core of operating system
  - Initialize and control all the resources in a computer machine
    - Management of processes
    - Management of memory
    - Management of files
    - Management of devices
- User Program
  - Every program generally used by normal users
  - They use the resources of a machine
  - Editors, Browsers, Games, and etc.

Shell

- Provides an interface between the user program and the operating system kernel
  - Analyze the user commands and pass the interpreted command to kernel
- Either a **command interpreter** or a **graphical user interface**
- Traditional Unix/Linux shells are command-line interfaces (CLIs)
- Usually started automatically when you log in or open a terminal

Shells in UNIX/LINUX

Name	Description	Location	In Ubuntu
bash	The Bourne Again SHell	/bin/bash	
ksh	The Korn shell	/bin/ksh, /usr/bin/ksh	
pdksh	A symbolic link to ksh	/usr/bin/pdksh	
rsh	The restricted shell (for network operation)	/usr/bin/rsh	
sh	A symbolic link to bash	/bin/sh	
tcsh	A csh-compatible shell	/bin/tcsh	
zsh	A compatible csh, ksh, and sh shell	/bin/zsh	

/etc/shells → List of available shell in the machine

The Bash Shell

- Linux's most popular command interpreter
  - The Bourne-Again Shell (from 1988)
  - More sophisticated than the original sh by Steve Bourne (the original Unix shell)
- Gives you a prompt and waits for a command to be entered
- Alternative shell → tcsh

Shell commands

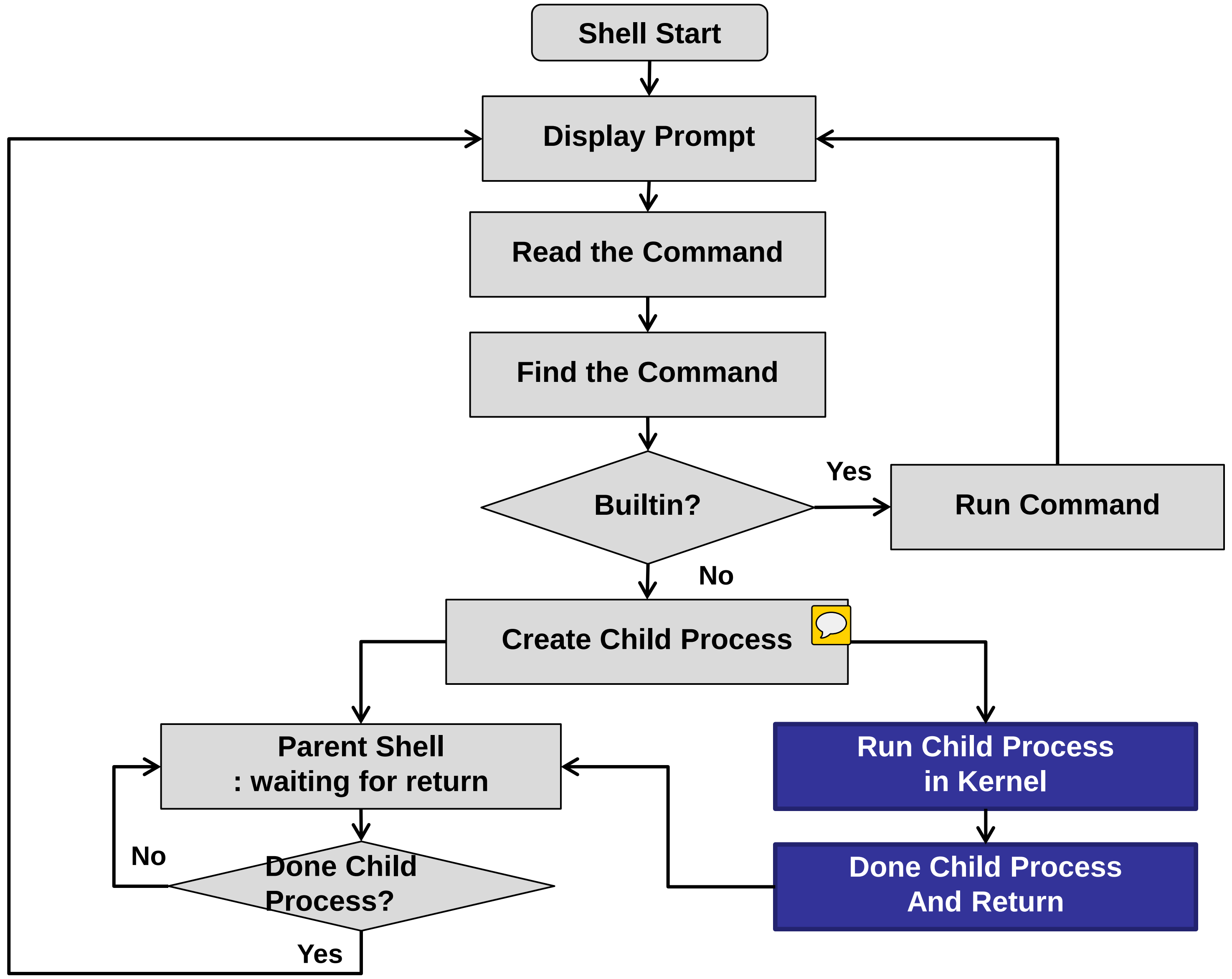
- Shell commands entered consist of words
  - Separated by spaces (whitespace)
  - The first word is the command to run
  - Subsequent words are options or arguments to the command
- Some commands are built into the shell
  - Called builtins
  - Only a small number of commands are builtins
    - E.g.) echo, printf, read, cd, pwd, let, eval, set, unset, exec, help, logout, exit, true, false, local, export, source …

Bulit-in commands

Built in Commands	Action
alias	Pint or set the alias of a command.
echo	Output the given argument, separated by spaces.
printf	Output the formatted data
read	Read value from user
exec	Run the program which is given by argument
bg	Run the program in background mode
cd	Change directory
pwd	Check the current directory
exit	Exit from the shell
source	Read from the file and run it
set	Set variable
export	Set shell environment variable

And So on………

Flowchart of processing a shell command



Child/Parent Process

- Shell is a program
  - When invoke a new shell, it create a new process
    - Child process
  - The old shell became a parent process of the new shell process
    - C.f.) PPID variable
- If you want to run the command in background, add "&" on the last of command
  - Shell will not wait for the termination of its child process

```
kbkim@ubuntu:~$ echo $PPID
3324
kbkim@ubuntu:~$ /bin/bash
kbkim@ubuntu:~$ echo $PPID
3724
kbkim@ubuntu:~$ heavy_work &
kbkim@ubuntu:~$
```

Command-Line Arguments

- The words after the command name
- Two categories of arguments
  - Options, usually starting with “-” or “--”
  - Filenames, directories, etc., on which to operate
- The options usually come first, but for most commands they do not need to

```
echo [-neE] [arg …]
 -n : the tailing newline is suppressed
 -e : backslash-escaped characters is enabled ( ₩a, ₩b, etc.)
 -E : backslash-escaped characters is disabled
example) compare ‘echo -e 123₩n123’ and ‘echo -n 123₩n123’
```

Syntax of command-line options

- Syntax of options
  - Single Letter options start with a hypen "**-**"
    - E.g.) -n, -e, -E in echo command
  - Less cryptic options are whole words or phrases, and start with two hyphens "**--**"
    - E.g.) --all option in ls command (same to -a)
  - Some options themselves take arguments
    - Usually the argument is the next work
      - E.g.) sort -o output_file
- Some exceptions
  - Some programs use different styles of command-line options
  - E.g.) gcc (c compiler) uses "-" option rather than "--"

Example : Read and Echo

```
kbkim@ubuntu:~$ read var_value
Linux is interesting
kbkim@ubuntu:~$ echo $var_value
Linux is interesting
kbkim@ubuntu:~$
```

Read user input to "var_value"

Print value of "var_value" to output

```
kbkim@ubuntu:~$ read –p "Insert Name : " name_value
Insert Name : peterpan
kbkim@ubuntu:~$ echo $name_value
peterpan
kbkim@ubuntu:~$
```

Redirection

- Standard I/O
  - stdin : standard input (keyboard), described by 0
  - stdout : standard output (console), described by 1
  - stderr : error output (console), described by 2
- Redirection Symbols
  - "<" : "a < b" means reading contents of "b" as standard input of "a"
  - ">" : "a > b" means writing results of "a" through the standard output into file "b".
    - If "b" is not empty, its contents are truncated
  - ">>" : "a >> b" means appending results

Output Redirection

```
kyungbak@gamera% ls > result2
kyungbak@gamera% cat result2
result2
test1*
test2
kyungbak@gamera% ps >> result2
kyungbak@gamera% cat result2
result2
test1*
test2
  PID TTY          TIME CMD
14863 pts/7        0:00 ps
14705 pts/7        0:00 tcsh
kyungbak@gamera%
```

Output of "ls" is redirected to a file "result2"

Output of "ps" is redirected to a file "result2", in an appending manner

"cat"

- With argument : Printing the contents of a file
- Without argument : Printing the user input

Input/Output Redirection

```
kyungbak@gamera% cat > test2
ps
ls -l
^C
kyungbak@gamera% sh < test2 > result2
kyungbak@gamera% cat result2
   PID TTY          TIME CMD
 14844 pts/7        0:00 sh
 14845 pts/7        0:00 ps
 14705 pts/7        0:00 tcsh
total 16
-rwx------   1 kyungbak guest          13 Mar 14 02:29 test1
-rw-------   1 kyungbak guest           9 Mar 14 02:39 test2
kyungbak@gamera%
```

Output of "cat" is redirected to a file "test2"

- Input of "sh" is redirected from a file "test2"
- Output of "sh" is redirected to a file "result2"

Output/Error Redirection

```
kyungbak@gamera% ( ls -l > file ) > & errfile
kyungbak@gamera% cat file
total 24
-rw-------   1 kyungbak guest        0 Mar 14 05:25 errfile
-rw-------   1 kyungbak guest        0 Mar 14 05:25 file
-rw-------   1 kyungbak guest      105 Mar 14 02:43 result2
-rwx------   1 kyungbak guest       13 Mar 14 02:29 test1*
-rw-------   1 kyungbak guest        9 Mar 14 02:39 test2
kyungbak@gamera% cat errfile
kyungbak@gamera% ( ls -z > file ) > & errfile
kyungbak@gamera% cat file
kyungbak@gamera% cat errfile
ls: illegal option -- z
usage: ls -1RaAdCxmnlhogrtuvVcpFbqisfHLeE@ [files]
kyungbak@gamera%
```

> : stdout
>& : stderr

Multiple commands

- “;” → run the commands sequentially
- “&” → run the commands simultaneously

```
kyungbak@gamera% date; ls –l test1 ; whoami
Wed Mar 14 02:32:36 PDT 2012
-rwx------   1 kyungbak guest        13 Mar 14 02:29 test1*
kyungbak
kyungbak@gamera% date & ls –l test1 & whoami
[1] 14788
Wed Mar 14 02:32:52 PDT 2012
[1]    Done                          date
[2] 14789
kyungbak
-rwx------   1 kyungbak guest        13 Mar 14 02:29 test1*

[2]    Done                          ls –F –l test1
kyungbak@gamera%
```

Pipe

- Use multiple command in serial
  - Sequentially processes with the results of the previous command as the standard input of the current command
  - prog1 arg1 arg2|prog2
    - "prog1" performs with "arg1" and "arg2"
    - "prog1" has "output1"
    - "prog2" performs with "output1" as the standard input
  - " ps | sort " is same to "ps > output ; sort < output"
    - c.f.) Latter case generate "output" file

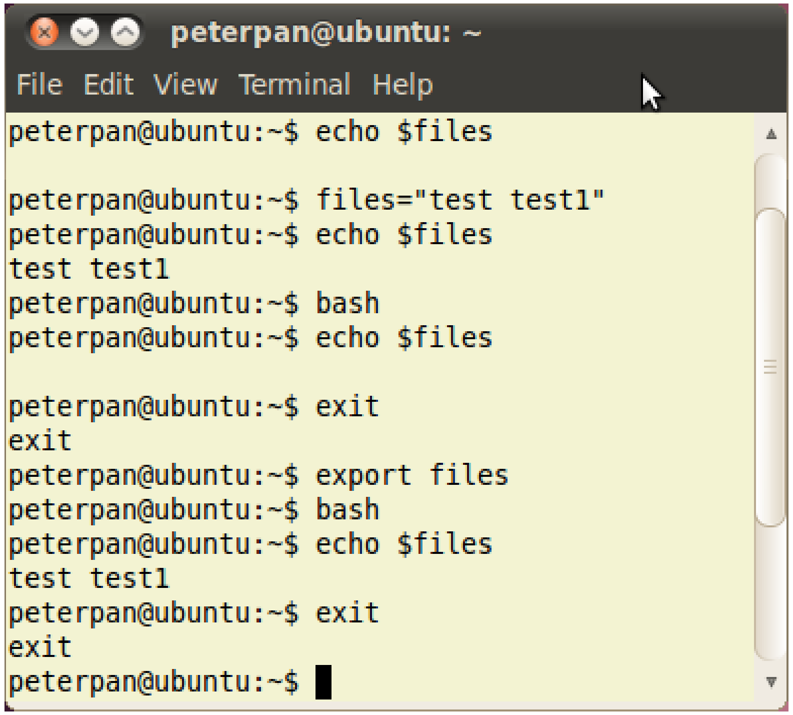
```
kyungbak@gamera% ps
   PID TTY          TIME CMD
 15376 pts/7       0:00 ps
 15258 pts/7       0:00 tcsh
kyungbak@gamera% ps | sort
   PID TTY          TIME CMD
 15258 pts/7       0:00 tcsh
 15377 pts/7       0:00 ps
 15378 pts/7       0:00 sort
kyungbak@gamera% ps | grep ps
 15415 pts/7       0:00 ps
kyungbak@gamera%
```

Shell Variables

- Store temporary "String" values
  - E.g) files="notes.text report.text"
    - The double quotes are needed because the value contains a space
    - **No white space before/after "=“**
- **Use the values of a shell variables with dollar mark ($)**
  - E.g.) echo $files
    - "$" mark tells the shell to insert the variable's value into the command line
- Use the "set" command (without argument) to list all the shell options and variables

Environment Variables

- **Shell variables are private to the shell**
  - Can not share the variable used in a shell with other shells
- A special type of shell variables called environment variables are passed to programs run from the shell
  - You need to "**export**" the variable
- The "env" command lists environment variables



```
peterpan@ubuntu:~$ echo $files

peterpan@ubuntu:~$ files="test test1"
peterpan@ubuntu:~$ echo $files
test test1
peterpan@ubuntu:~$ bash
peterpan@ubuntu:~$ echo $files

peterpan@ubuntu:~$ exit
exit
peterpan@ubuntu:~$ export files
peterpan@ubuntu:~$ bash
peterpan@ubuntu:~$ echo $files
test test1
peterpan@ubuntu:~$ exit
exit
peterpan@ubuntu:~$ 
```

Environment Shell Variables

Name	Value
PS1	The primary prompt string
HOME	The current user's home directory
PATH	A colon separated list of directories in which the shell looks for commands
PWD	The current working directory as set by the cd builtin
UID	The numeric real user id of the current user. Read-only
GROUPS	An array variable containing the list of groups of which the current user is a member
SECONDS	The number of seconds since the shell was started
HOSTNAME	The name of the current host
PPID	The process ID of the shell's parent process

And So on………

How to find a program

- The location of a program can be specified explicitly
  - "./a.out" runs the "a.out" program in the current directory
  - "/bin/ls" runs the "ls" command in the "/bin" directory
- Otherwise the shell looks in standard places for the program
  - Using "PATH" environment variable
  - Directory names are separated by colon
  - Running the program whichever is found first

```
kbkim@ubuntu:~$ echo $PATH
/bin:/usr/bin:/usr/local/bin
kbkim@ubuntu:~$ whoami
kbkim
kbkim@ubuntu:~$ PATH="/home/kbkim/bin":$PATH
kbkim@ubuntu:~$ echo $PATH
/home/kbkim/bin:/bin:/usr/bin:/usr/local/bin
```

/bin/whoami or
/usr/bin/whoami or
/usr/local/bin/whoami

Set “PS1” shell variable

Change Shell Prompt

Value	Prompt Present
anyString	Show the given “anyString”
₩d	Day of the week, Month, Day
₩H	Domain Name (jnu.ac.kr)
₩h	Host Name (myweb)
₩u	User name
₩w	Absolute path of the current directory
₩W	Last directory name of the absolute path of the current directory
₩t, ₩T	HH:MM:SS (24hours, 12hours)
₩$	If UID is 0 (root), present “#”. Otherwise present “$”

Let’s change shell prompt

```
kbkim@ubuntu:~$ PS1=“₩$”
$ PS1=“₩u₩$”
Kbkim$ PS1=“₩u hi ₩t₩$”
Kbkim hi 23:09:21$
Kbkim hi 23:09:49$ exit
Then restart shell
kbkim@ubuntu:~$
```

When you come back, your change is reset

- The change is only effective to the current shell.
- When you start a new shell it is reset
- How can we set it for permanently?
  - Using configuration files

Configuration file

- The sequence of configuration files when a bash shell starts just after login
  - /etc/profile
    - Overall system environment
  - ~/.bash_profile or ~/.bash_login or ~/.profile
    - Call one of file whichever is first found
  - ~/.bashrc
    - Environment settings per user
    - Called by ~/.profile or other equivalents
- During login, for starting a bash
  - Use ~/.bashrc
- For Logout, using ~/.bash_logout

Edit ".bashrc" and deploy changes

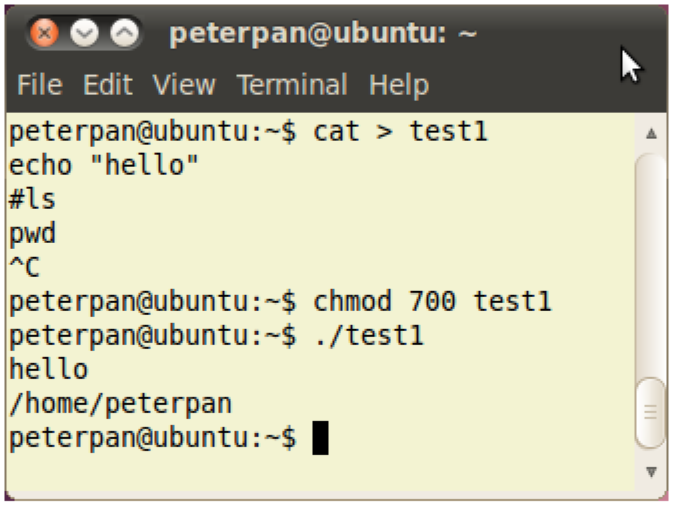
- Use vi or any text editor, and write the command in ".bashrc"
- But, it is not applied directly to the current shell.
- "source .bashrc" command for deploying change to the current shell

Example : alias

- alias : making an alias of other command
  - Examples of general usage (in .bashrc)
    - alias lf="ls –F"
    - alias ls="ls –l"
    - alias rm="rm –i"
    - alias mv="mv –i"
    - alias cp="cp –i"
- unalias : deleting the alias

Shell Programming

- Users can make an executable shell file, and run it
- Usually, this executable shell file is called, "Shell Script File"
- How to create
  - At the first line, state "#!/bin/bash"
  - List variables and commands
- How to run
  - Use "chmod" to give "x" (executable) permission to the file, then run
- In a script file, a line starting with "#" considered as comments
  - Except the line "#!/bin/bash"



```
peterpan@ubuntu:~$ cat > test1
echo "hello"
#ls
pwd
^C
peterpan@ubuntu:~$ chmod 700 test1
peterpan@ubuntu:~$ ./test1
hello
/home/peterpan
peterpan@ubuntu:~$
```

Variables

- A script file can have variables
  - Just same as shell variables
  - Do no have types
  - Only String
  - The name of variable can not start with a digit or a special character
  - Expanded by “$” with various options

```
kyungbak@gamera% cat > test_var
#!/bin/bash
a="hello world"
echo "a is $a"
^C
kyungbak@gamera% chmod 700 test_var
kyungbak@gamera% ./test_var
a is hello world
kyungbak@gamera%
```

Expansion of variables – assignment

Expansion of Variable	Meaning
${var_name}	Use the value of var_name
${var_name:=value}	• If var_name is **null**, set the value of var_name as **the given value**. • Otherwise, use the old one.
${var_name:+value}	• If var_name is **null**, set the value of var_name as **empty string**. • Otherwise, temporary use the given value for the var_name but not saved
${var_name:–value}	• If var_name is **null**, set the value of var_name as **the given value but not saved**. • Otherwise, use the old value
${var_name:?value}	• If var_name is **null**, **end the shell script** and return error with the given value. • Otherwise, use the old value
${#var_name}	The length of $var_name

Example of expansion – assignment

```
#!/bin/bash
#testx
a="xxy“
echo "$a"
echo "1:${a:="test1"}"
echo "1:$a"
echo "1n:${x:="test1"}"
echo "1n:$x"
echo "2:${a:-"test2"}"
echo "2:$a"
echo "2n:${b:-"test2"}"
echo "2n:$b"
echo "3:${a:+"test3"}"
echo "3:$a"
echo "3n:${c:+"test3"}"
echo "3n:$c"
echo "4:${a:?"test4"}"
echo "4:$a"
echo "4:${#a}"
echo "4n:${d:?"nonexist d"}"
```

Run

```
kyungbak@gamera% ./testx
xxy
1:xxy
1:xxy
1n:test1
1n:test1
2:xxy
2:xxy
2n:test2
2n:
3:test3
3:xxy
3n:
3n:
4:xxy
4:xxy
4:3
./testx: line 21: d: nonexist d
```

Expansion of variables – pattern searching

Expansion of variables	Meaning
${var_name%pattern}	• Find the first matched pattern from the end of the value • Return the value without the string after the founded pattern (including)
${var_name%%pattern}	• Find the last matched pattern from the end of the value • Return the value without the string after the founded pattern (including)
${var_name#pattern}	• Find the first matched pattern from the head of the value • Return the value without the string before the founded pattern (including)
${var_name##pattern}	• Find the last matched pattern from the head of the value • Return the value without the string before the founded pattern (including)

Working with wild card “*”

Example of expansion – pattern searching

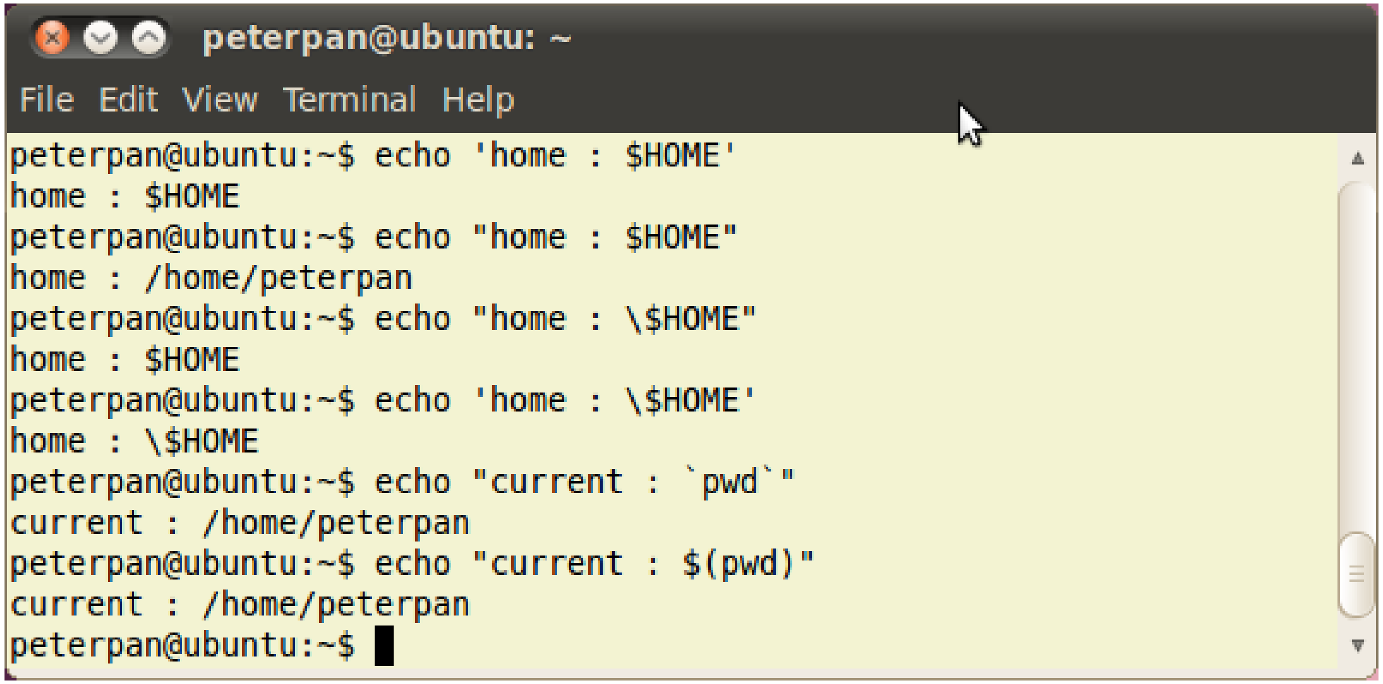
```
#!/bin/bash
#./test_expvar
a=/home/kbkim/test
echo "correct usage"
echo ${a%/*}
echo ${a%%/*}
echo ${a#*/}
echo ${a##*/}
```

```
kbkim@ubuntu:~/test$ ./test_expvar
correct usage
/home/kbkim

home/kbkim/test
test
kbkim@ubuntu:~/test$
```

Quotes for variables

Quotes	Meaning
Single quotes (‘string‘)	String
Double quotes (“string”)	String, but $,` and ₩ which are special characters are working
Back quotes (\`string\`)	Command, alternative way → $(command)



```
peterpan@ubuntu:~$ echo 'home : $HOME'
home : $HOME
peterpan@ubuntu:~$ echo "home : $HOME"
home : /home/peterpan
peterpan@ubuntu:~$ echo "home : \$HOME"
home : $HOME
peterpan@ubuntu:~$ echo 'home : \$HOME'
home : \$HOME
peterpan@ubuntu:~$ echo "current : `pwd`"
current : /home/peterpan
peterpan@ubuntu:~$ echo "current : $(pwd)"
current : /home/peterpan
peterpan@ubuntu:~$ 
```

Exporting variables

- Export a variable to environment variables
- Child process can share the exported variable

```
kyungbak@gamera% cat main
#!/bin/bash
name=Peterpan
land=Neverland
echo main:name is $name
echo main:land is $land
export name
./sub
kyungbak@gamera% cat sub
#!/bin/bash
echo sub:name is $name
echo sub:land is $land
kyungbak@gamera%
```

```
kyungbak@gamera% ./main
main:name is Peterpan
main:land is Neverland
sub:name is Peterpan
sub:land is
kyungbak@gamera%
```

Special Shell Variables for Shell Programming

Variable names	Values
$	PID of the current process
?	Return value of the last command
!	PID of the child process
#	Number of arguments
0	Command String
1, 2, …	Argument String (1,2,…)
* Or @	Array of argument string
_	Last argument or command if there is no argument

Example of Special Variables

```
#!/bin/bash
#File : test
echo 1:$$
echo 2:$?
echo 3:$!
echo 4:$#
echo 5:$0
echo 6:$1
echo 7:$2
echo 8:$*
for x in $*
do
      echo 9:$x
done
Echo 10:$@
for x in $@
do
      echo 11:$x
done
echo 12:$_
```

```
$ chmod 700 test
```

```
$ ./test a b c
1:4065
2:0
3:
4:3
5:./test
6:a
7:b
8:a b c
9:a
9:b
9:c
10:a b c
11:a
11:b
11:c
12:c
$
```

Lets calculating : expr

- Calculate the basic mathematical operations (+,-,*,/,%) and put the result to stdout
- Integer-based
- **Whitespace** is required before and after the operator
  - C.f.) expr 3 + 4 (ok), expr 3+4 (won't work)
- Recently $((equation)) is generally used

Example of expr

```
#!/bin/bash
#test_expr
a=10
b=3
echo `expr $a + $b`
echo `expr $a – $b`
echo `expr $b – $a`
echo `expr $a / $b`
echo `expr $a % $b`
echo `expr $a ₩* $b`
echo $(($a*$b))
echo `expr $a ₩* $b`
```

```
kbkim@ubuntu:~/test$ ./test_expr
13
7
-7
3
1
30
30
30
kbkim@ubuntu:~/test$
```

If statement

```
If [condition];
then
        command1;
fi
```

```
If [condition];
then
        command1;
else
        command2;
fi
```

```
If [condition1];
then
        command1;
elif [condition2];
then
        command2;
else
        command3;
fi
```

- If-then : If condition is true, then do command1
- If-else : If condition is false, then do command2
- If-elif : if condition1 is true, then do command1, otherwise if condition2 is true, then do command2, else do command3

How to test?

	condition	testing
String comparison	[string]	If string is not empty, true
	[string1 = string2]	If string1 is same to string2, true
	[string1 != string2]	If string1 is not same to string2, true
	[–n string]	If string is not null, true
	[–z string]	If string is null, true
Arithmetic Comparison	[expr1 –eq expr2]	If expr1 is same to expr2, true
	[expr1 –ne expr2]	If expr1 is not same to expr2, true
	[expr1 –gt expr2]	If expr1 > expr2, true
	[expr1 –ge expr2]	If expr1 >= expr2, ture
	[expr1 –lt expr2]	If expr1 < expr2, true
	[expr1 –le expr2]	If expr1 <= expr2, true
	[!expr]	If expr is false, true
	[expr1 –a expr2]	If expr1 AND expr2 is true, true
	[expr1 –o expr2]	If expr1 OR expr2 is true, true

How to test? (cont’)

condition	testing
[–b FILE]	If FILE is a block device (disk), true
[–c FILE]	If FILE is a character device (keyboard), true
[–d FILE]	If FILE is a directory, true
[–e FILE]	If FILE is exist, true
[–f FILE]	If FILE is exist and regular file, true
[–h FILE]	If FILE is a symbolic link file, true
[–r FILE]	If FILE is readable, true
[–s FILE]	If FILE is not empty, true
[–S FILE]	If FILE is a socket device (network socket), true
[–w FILE]	If FILE is writable, true
[–x FILE]	If FILE is executable, true
[–O FILE]	If FILE is owned by the current user, true
[–G FILE]	If FILE’s group is same to the current user’s group, true

List of commands

- AND List
  - com1 && com2 && com3
  - Do commands sequentially until the result is false
- OR List
  - com1 || com2 || com3
  - Do command sequentially until the result is true
- AND list and OR list can be mixed
  - [cond] && {com1 com2} || com3
  - If cond is true, do com1 and com2, otherwise cond is false, do com3

Example with .bashrc

```
# enable color support of ls and also add handy aliases
if [ -x /usr/bin/dircolors ]; then
    test -r ~/.dircolors && eval "$(dircolors -b ~/.dircolors)" || eval "$(dircolors -b)"
    alias ls='ls --color=auto'
    #alias dir='dir --color=auto'
    #alias vdir='vdir --color=auto'

    alias grep='grep --color=auto'
    alias fgrep='fgrep --color=auto'
    alias egrep='egrep --color=auto'
fi
```

"eval" command : execute the given command string

"dircolors" command : return the command string for setting up the configuration of directory colors

Case statement

```
case var_name in
        pattern1) command1;;
        pattern2) command2;;
esac
```

If “var_name” is matched “pattern1”, do “command1”, and so on.

Example : part of .bashrc file

```
# If this is an xterm set the title to user@host:dir
case "$TERM" in
xterm*|rxvt*)
   PS1="₩[₩e]0;${debian_chroot:+($debian_chroot)}₩u@₩h: ₩w₩a₩]$PS1"
   ;;
*)
   ;;
esac
```

For statement

```
for var_name in val1 val2 …
do
        command
done
```

For every "val*n*" as the value of "var_name", do command

```
#!/bin/bash
#test_for
for var in "apple" "banana"
do
        echo $var
done
for file in $(ls)
do
        echo file:$file
done
```

```
kbkim@ubuntu:~/test$ ./test_for
apple
banana
file:test
file:test_expr
file:test_expvar
file:test_for
kbkim@ubuntu:~/test$
```

Othre expression of for statement

List of Numbers

```
for VARIABLE in 1 2 3 4 5 .. N
do
        command1
        command2
        …
        commandM
done
```

List of Files

```
for VARIABLE in file1 file2 file3
do
        command1 on $VARIABLE
        command2
        …
        commandM
done
```

Return value of command (e.g. ls)

```
for OUTPUT in $(Linux-Or-Unix-Command-Here)
do
        command1 on $OUTPUT
        command2
        …
        commandM
done
```

Expression for ranges

- {start..end}

```
#!/bin/bash
for i in {1..5}
do
   echo "Welcome $i times"
done
```

- {start..end..increment}

```
#!/bin/bash
echo "Bash version ${BASH_VERSION}..."
for i in {0..10..2}
  do
     echo "Welcome $i times"
 done
```

While and Until Statement

```
while condition
do
        command
done
```

While condition is true, do command

```
until condition
do
        command
done
```

While condition is fa lse, do command

- break → exit from while, until and for
- continue → ignore the command after “continue” command and do again while, until and for

Example of while

```
#!/bin/bash
#test_while
echo "Enter password: "
read passwd1
echo "Retype password: "
read passwd2

while [ "$passwd1" != "$passwd2" ]
do
        echo "Mismatched!! Try again"
        echo "Retype password: "
        read passwd2
done
echo "OK password matched"
```

Select Statement

```
select var_name in val1 val2 …
do
        command
done
```

- Shell gives a selection prompt with the given values (val1, val2,…)
- Command is done with the selected variable
- break can be used to end the select statement

Example of Select

```
#!/bin/bash
echo "What is your Linux?"
select var in "Redhat" "Fedora" "Ubuntu" "SUSE"
do
        if [ "$var" = "Ubuntu" ]
        then
                option="You are Rock!!"
        else
                option="Please use Ubuntu"
        fi
        break;
done
echo "Your Linust is $var"
echo "$option"
```

Function

- User can define explicit function
  - Do the statements
  - Then return the value to the caller
- Export function
  - User “–f” option
  - e.g.) export –f user_print

```
func()
{
        statement
        return value
}
```

Example

```
#!/bin/bash
user_print()
{
        echo “user print done”
}

user_print
echo “user print is used”
```

Practical Example : exec command

- Very unutilized unix command, but here and there it is used as a part of shell scripts
- Replaces the current shell process with the specified command
- In shell programming, it is generally used to redirect fire descriptors
  - exec fd<file : open file for input with file descriptor fd
  - exec fd>file : open file for output with file descriptor fd
  - Excample :
    - exec 9<&0 : copy standard input descriptor to file descriptor 9
    - exec 0< /proc/mounts : redirect standard input to “/proc/mounts”
    - exec 0<&9 9<&- : copy file descriptor 9 to standard input file descriptor and close the file descriptor 9

&n : indicate “n” is a file descriptor, “-” file descriptor means null

Stdin Redirection with exec

```
#!/bin/bash

exec 9<&0
exec 0<< EOF     1
one
two
three
four
five
EOF

for I in 1 2; do
  read LINE
  echo "number: ${LINE}"
done
```

"exec [fd]<< [string]" : Use the following string which ends in [string] as a file with file descriptor [fd]

```
exec 8<&0
exec 0<< EOF1     2
a
b
c
EOF1
while read LINE; do
  echo "letter: ${LINE}"
done

exec 0<&8         3
while read LINE; do
 echo "number: ${LINE}"
done

exec 0<&9
exec 8<&- 9<&-    4

echo -n "term: "
read TERMINAL
echo "${TERMINAL}"
```

- "read" command read data from standard input (0)
- Standard input keeps changing

	0	9	8
1	EOF	0	
2	EOF1	0	EOF
3	EOF	0	EOF
4	0	-	-

Stdout Redirection with exec

```
#!/bin/bash

NUMBERS="$(tempfile)"
exec 9>&1
exec 1> "${NUMBERS}"
```

1

```
for WORD in "one" "two"; do
  echo "number: ${WORD}"
done

LETTERS="$(tempfile)"
exec 8>&1
exec 1> "${LETTERS}"
```

2

“tempfile” command : Generate a temp file under “/tmp” folder

```
for WORD in "a" "b" "c"; do
  echo "letter: ${WORD}"
done
```

3

```
exec 1>&8
for WORD in "three" "four" "five";
do
 echo "number: ${WORD}"
done

exec 1>&9
exec 8>&- 9>&-
```

4

```
echo "--NUMBERS--"
cat "${NUMBERS}"

echo "--LETTERS--"
cat "${LETTERS}"

rm "${NUMBERS}"
"${LETTERS}"
```

- “read” command read data from standard input (0)
- Standard output keeps changing

	1	9	8
1	NUM	1	
2	LET	1	NUM
3	NUM	1	NUM
4	1		