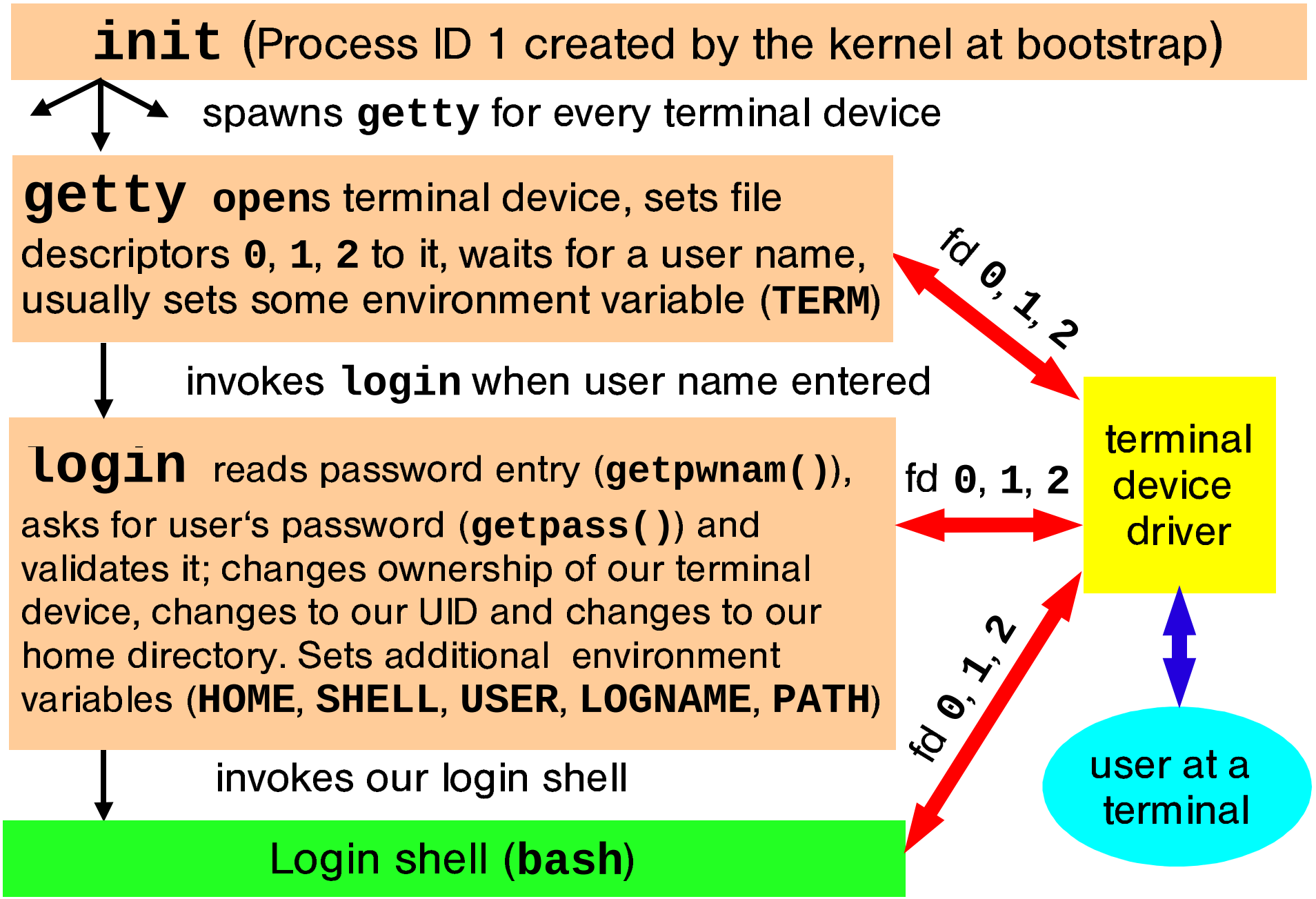
CS3281
UNIX Programming

Part 7

Shell Programming

Zoltan KATO (cs3281@comp.nus.edu.sg)
S16 #06–12

Logging in to a UNIX System



What is a Shell?

- Just a Unix program executed when you log in
- A command interpreter
  - provides the basic user interface to UNIX utilities
- A programming language
  - program consisting of shell commands is called a **shell script**
  - you can put commands in a file and execute it:
    - First, make the file executable (**`chmod u+x`** ***script-file***)
    - Lines starting with **#** are comments
    - Make use of **interpreter files** (kernel feature!): the first line of your script file must begin with a line:

    #!***pathname optional-arguments***

    where ***pathname*** is an absolute pathname (typically **`/bin/sh`**, or **`/bin/bash`**) of the interpreter.
- We will focus on the BASH shell (combines features of Bourne, Korn and C shell)

Command Execution

current shell (**bash**)

executes built-in commands (**echo**, **kill**, **pwd**,...) or shell scripts invoked by the **.** (dot) command: **.** ***shell−script***

wait

or

continue

if command is executed in the background

fork() (creates a new shell process)

sub-shell (**bash**)

executes a shell script or calls **exec()** to execute a program

variables defined in the parent are not passed (use the **export** command in parent to pass variables). Variables defined or changed in a sub-shell does not affect variables of the parent !

sub-shell terminates after script or program execution

List of Commands, Separation, Grouping

- **`newline`**: start execution of a command.
- Separation in command lists (**`|`** − pipe, discussed later)
  - Return status is the exit status of the **last command executed**
  - Commands separated by **`;`** are executed **sequentially**
  - **`&`** causes the execution of the command in **background**
    - Shell does not wait for termination, return status is **`0`**
    - Standard input is **`/dev/null`** unless otherwise specified.
  - **`cmd1 && cmd2`**: AND list. **`cmd2`** is executed if and only if exit status of **`cmd1`** is **zero**!
  - **`cmd1 || cmd2`**: OR list. **`cmd2`** is executed if and only if exit status of **`cmd1`** is **non−zero**!
- Grouping. Return status is the exit status of **`cmd-list`**:
  - **`( cmd-list )`** causes the execution in a ***subshell***.
  - **`{ cmd-list; }`** causes the execution in the ***current shell***

Redirection, Pipelines

- ***cmd* < *file*** : redirect standard input of ***cmd***
- ***cmd* > *file*** : redirect standard output of ***cmd*** (**>>**: append)
  - in general: ***n*<**, ***n*>** or ***n*>>** : redirect file descriptor ***n*** (**2>** redirect standard error)
  - redirecting both standard output **and** standard error: **&>** ***file***
- Duplicating file descriptors: ***n*>&*m*** file descriptor ***n*** is made to be a copy of file descriptor ***m***
- ***cmd1* | *cmd2*** : pipe. Redirect standard output of ***cmd1*** to standard input of ***cmd2***. Example:
  - **`ls -l | tail -50`**
  - **`tail`** as a **filter**: **`ls -l | tail -50 | less`**
- redirection operators are processed left to right:
  - **`ls > /tmp/dir 2>&1`** is not the same as
  - **`ls 2>&1 > /tmp/dir`**

Filename Expansion

- Special characters for **filename patterns**:
  - **?** – matches any **single character**:
    - **file?** matches **file1 filea** but does not match **file12**
  - ***** – matches **any string**, including the null string:
    - **file*** matches anything with a prefix **file** (**file**, **file12**)
  - **[]** – matches any one of the enclosed characters:
    - **file[12]** matches **file1** and **file2** only
    - **file[1-9]** matches **file1**, **file2** ,..., **file9**
- The shell generates a list of filenames matching the pattern. **NOTE**: a leading **.** does not match (**.profile**)

  assuming the filenames: a1.tex a.c a1.ps ab ab.txt
  - **ls a?.*** will be expanded by the shell to

    ls a1.tex a1.ps ab.txt
- a word beginning with a **~** prefix is expanded as:
  - **~** – the value of **$HOME**, **~fred** – the home dir. of user **fred**

Variables

- User−created variables: ***name*=*value*** (**no space** around **=**)
  - **`x=2  str=string  empty=  spaces=" s t r"`**
- Environment (aka. keyword or shell) variables:
  - inherited or initialized by the shell when started up (their value can be changed): **`HOME  PATH  TERM  PS1`**
- Manipulating variables
  - **`$`**, **`${}`** is used to substitute the value of a variable: **`$x ${x}`**
    - **`echo $str`** prints **`string`** but **`echo str`** prints **`str`** !!
  - **`unset`** − removes a variable
  - **`export`** − makes a variable available to child processes
- Read−only variables:
  - **`$1`**,...,**`$9`**,**`${10}`**,...: command line arguments, **`$0`**: prog. name
  - **`$#`**: number of arguments, **`$@`**: all of the arguments (**`$1`**, **`$2`**,...)
  - **`$?`**: exit status of last command, **`$$`**: PID of current process

Command Substitution, Quoting

- **$(*cmd*)** − command substitution: replaces with the standard output of ***cmd***.
- Quoting removes the special meaning of spec. characters:
  - **** − preserves the literal value of the next character (**\$1**)
  - **'** − literal value of enclosed characters (**'\$1'**)
  - **"** − literal value of enclosed characters except:
    - **$** retains its special meaning and
    - \ retains its special meaning when followed by the characters **$ " **
    - **"$@"** quotes the arguments individually

```
bash$ cat > script
echo $1 /tmp/q*
echo \$1 /tmp/q\*
echo '$1 /tmp/q\*'
echo "$1 /tmp/q\*"
```

```
bash$ . script arg
arg /tmp/q10 /tmp/q9 /tmp/quote
$1 /tmp/q*
$1 /tmp/q\*
arg /tmp/q\*
```

Arithmetic Expressions

- Evaluation is done in **long integers**, only division by 0 is trapped. Expressions may be evaluated by
  - the **`let`** builtin
  - **`$(( `*`expr`*` ))`** − arithmetic substitution: evaluates and substitutes the result of the arithmetic expression ***expr***
- Operators, precedence rules are basically the same as in C
  - **`- + * / % = += *=`** ....
  - **`== != < > <= >= && || ?:`** (conditional expression)
    - 0 means false, any other value means true
  - **`**`** exponentiation

```
bash$ echo $((5+1))
6
```

```
bash$ i=4
bash$ echo $(( i+=1 )) $(( i+=1 ))
5 6
```

Shell variables are allowed as operands. Be careful:

- **`echo $(( $i+1 ))`** and **`echo $(( i+1 ))`** are OK
- but **`echo $(( $i+=1 ))`** **is wrong!**

Control−Flow Commands

- Looping (**zero** exit status means **true**):
  - **`until`** ***test−cmd*****`;`** **`do`** ***cmd*****`;`** **`done`** : ***cmd*** is executed as long as ***test−cmd*** has non−zero exit status
  - **`while`** ***test−cmd*****`;`** **`do`** ***cmd*****`;`** **`done`** : ***cmd*** is executed as long as ***test−cmd*** has zero exit status
  - **`for`** ***name*** **`in`** ***words*****`;`** **`do`** ***cmd*****`;`** **`done`** : ***cmd*** is executed once for each member in ***words***. If "**`in`** ***words***" is omitted then "**`$@`**" is used.
- Conditional constructs:
  - ***pattern*** is analogous to file name patterns, only the first matching commands are executed!

```
case test−string in
    pattern1)  cmd1;;
    pattern2)  cmd2;;
      . . . . .
esac
```

```
if test−cmd1; then cmd1;
elif test−cmd2; then cmd2;
      . . . . .
else cmd3;
fi
```

Conditional Expressions, **`test`**

- Conditional expressions are used by the **`test`** and **`[]`** built−in commands

```
if [ -d /tmp ] ; then cd /tmp; fi
if test -d /tmp; then cd /tmp; fi
```

- File status:
  - **`-e`** ***file*** true if ***file*** exists
  - **`-d`** ***file*** true if ***file*** exists and is a directory
  - **`-f`** ***file*** true if ***file*** exists and is a regular file
- String operators:
  - **`-z`** ***string*** true if length of ***string*** is zero
  - **`-n`** ***string*** true if length of ***string*** is non−zero
  - ***string1*** ***`OP`*** ***string2*** where ***`OP`*** is one of **`== != < >`**
- Numeric comparison:
  - ***arg1*** ***`OP`*** ***arg2*** where ***`OP`*** is one of **`-eq`** (==) **`-ne`** (!=) **`-lt`** (<) **`-le`** (<=) **`-gt`** (>) **`-ge`** (>=). Arguments may be positive or negative integers.

Variable Expansion

- Set value of an unset or null variable:
  - **${*name*:-*default*}** − use ***default*** when ***name*** is unset or null
  - **${*name*:=*default*}** − set ***name*** = ***default*** when it is unset/null
  - **${*name*:?*msg*}** − print ***msg*** on standard error and exit script when ***name*** is unset or null
- Pattern matching & removal. Patterns are similar to filename patterns:
  - **${*name*#*pattern*}**, **${*name*##*pattern*}** − produces the expansion of ***name*** with the shortest (**#**) or longest (**##**) matching **prefix** removed.
  - **${*name*%*pattern*}**, **${*name*%%*pattern*}** − produces the expansion of ***name*** with the shortest (**%**) or longest (**%%**) matching **suffix** removed.

```
bash$ a=pacific; echo ${a%i*} ${a%%i*} ${a#p*i} ${a##p*i}
pacif pac fic c
```

Useful Builtins

- **`help`** − display help about builtin commands (like **`man`**)
- **`echo`** − Output its arguments separated by spaces, terminated by a newline unless **`-n`** is specified.
- **`let`** ***expr*** − performs arithmetic evaluation on ***expr***
- **`printf`** ***format*** − writes its formatted arguments to standard output. ***format*** is the same as for **`printf()`**.
- **`read`** − One line is read from standard input and the first word is assigned to the first argument, ...
  - If there are more input words then arguments => leftover words are assigned to the last argument including spaces.

```
bash$ read -p "this is a prompt: " a b
this is a prompt: user input words
bash$ printf "a=%s; b=%s\n" "$a" "$b"
a=user; b=input words
```