Faculty of Computer Science, Dalhousie University *26-Sep-2018*

CSCI 2132 — Software Development

Lecture 10: Shells and Computing Environment

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Formatted Input and Output
  - printf function
  - scanf function

Let us look at some more examples. Suppose that we defined two variables as follows:

```
int i;
double x;
```

Then the following two scanf functions calls are equivalent:

```
scanf("%d %f", &i, &x);
scanf("%d%f", &i, &x);
```

The following two scanf function calls are not:

```
scanf("%d ", &i);
scanf("%d", &i);
```

This is because the first one will wait for the user to input a non-white space character after it reads the value of i (or until end of input is reached).

Next, suppose that we have two variables defined as:

```
double x, y;
```

These two function calls are not equivalent:

```
scanf("%f,%f", &x, &y);
scanf("%f ,%f", &x, &y);
```

This is because, as an example, the following user input would not be treated in the same way by them:

```
5.5 , 6e5
```

The first function would convert 5.5 and assign it to x, but then it would abort because it could not match the '`,`' from the format string with the white space in input. However, the second scanf function would succeed, and convert and assign $6e5$ (i.e., $600,000$) to y as well.

Example: Adding Fractions

In this example, we want write a program that prints out the string "`Enter expression: `" and waits for the user to enter an expression that represents addition of two factions in the following form:

```
2/3+1/6
```

Our program would calculate the result (without reducing the fraction to the lowest terms). Thus, the result for this input would be:

```
15/18
```

This is a solution to the problem:

```
#include <stdio.h>

int main() {
    int num1, denom1, num2, denom2, result_num, result_denom;

    printf(" Enter expression: ");
    scanf("%d / %d + %d / %d", &num1, &denom1, &num2, &denom2);

    result_num = num1 * denom2 + num2 * denom1;
    result_denom = denom1 * denom2;

    printf("%d/%d\n", result_num, result_denom);
    return 0;
}
```

You can notice how we put extra spaces in the scanf format in order to make it more robust and allow for a possibility that users enters extra spaces. The program will still work correctly with inputs that do not have any intermediate white spaces, as in the above example.

10 Shells and Computing Environment

Shells

- Reading: Unix book Chapter 4
- Separated program from kernel
    - not necessarily with all operating systems
- Advantages:
    - Crashing shell does not crash the system
    - Easy to replace or enhance shell without changing the kernel

Advantages of Separating a Shell from the Kernel. There are advantages of separating the shell from the kernel. First, if something goes wrong in the shell, it will not bring down the system. Other programs can still be running. Second, shells can be replaced without rewriting the kernel. Thus, it is easy to add new functionality by writing a new shell. This is good software development practice.

Shell Functionality

The following is a list of some major shell functionalities:

- Built-in commands
- Scripts
- Variables (local and environment)
- Redirection
- Wildcards (file name substitution)
- Pipes
- Sequences (conditional and unconditional)
- Subshells
- Background processing
- Command substitution

We learned some of these before, and we will learn some in this lecture. We will learn the rest when we learn shell scripting.

Popular Shells

There are four popular shells. Here we also give the common location of these shells when introducing them.

- Bourne shell (`/bin/sh`): This shell replaced the original Thompson shell (which was the first UNIX shell).
- Korn shell (`/bin/ksh`)
- C shell (`/bin/csh`)
- TC shell (`/bin/tcsh`)
- Bash shell (`/bin/bash`)
- Note: Use `cat /etc/shells` to see shells available
- We will focus on the Bash shell

Bash Shell

- We will use bash shell
  - widely available
  - includes many advanced features of other shells
- Default on bluenose
- Command `chsh` used to change default shell
  - not on bluenose, but can ask help-desk
- Command `finger` shows the default shell of a user
- File of interest: `/etc/passwd`

The Bash shell, or the Bourne Again shell, provides backward compatibility with the Bourne shell. It also includes most useful features of C/Korn shells. It is licensed under GPL, and is thus open-source. Most Unix systems have it. For this course, we will focus on Bash.

When you open a terminal and login, the default login shell is started and it can accept your commands. On bluenose, all new users are by default set up with the bash default login shell. This can be changed for each individual user. The command chsh can be used to change your login shell. On some servers (including bluenose) users are not allowed to change their default login shell by themselves. You can ask the root user to change this for you, or you can run your shell of choice each time after you login (or you can make use of a shell script). The command `finger` can be used to display the default shell of a user, among other information. For example:

```
finger test2132
```

may produce the output:

```
Login: test2132                         Name: Marker for CSCI 2132
Directory: /users/faculty/test2132      Shell: /bin/bash
Last login Fri Sep 20 15:22 (ADT) on pts/19 from test.cs.dal.ca
No mail.
No Plan.
```

from which we can see the default shell. Another place where the default shell information is stored is usually `/etc/passwd`, although due to use of LDAP, this information is not stored on bluenose. However, we can still see information about some users. If we use command `more /etc/passwd`, we can see the following first line:

```
root:x:0:0:root:/root:/bin/bash
```

where the line consists of the following fields separated by colon (:):

root — username

x — used to be encrypted password, but for security reasons it is stored elsewhere, which is indicated by 'x' (sometimes in the /etc/shadow file)

0 — UserID (UID) of the user

0 — GroupID (GID) of the primary user group

/root — home directory of the user

/bin/bash — the absolute path of the user shell, or command

Commands

There are two types of commands. The built-in commands are commands that a shell recognizes and executes internally. They are called built-in commands or internal commands. The code for these commands is executed within the shell program. This program is already in memory when shell is running so it is fast to execute them. However, it is difficult to replace these commands with new versions as they are part of the shell. Therefore, these commands typically execute simple tasks that we use frequently, or tasks which cannot be easily executed from a another process, or 'child' process in this case. Some examples of internal commands are cd, logout, and echo, although echo can also be an external command.

The command `echo` displays all of its arguments to stdout. This is useful when we need to create a file with a single line (output redirection). We also use this command to display the values of shell variables, and it is used by shell programs extensively to print output. If there are no arguments, echo will just produce one empty line at the output.

External commands are executable programs that are stored in the directory hierarchy, separate from the shell. Some examples are: ls, grep, sort, cut, and uniq. (uniq and cut are covered in a lab.)

Shell Variables

A shell behaviour is influenced by several string-valued shell variables. These variables are sometimes called *environment* variables, but, precisely speaking, the shell variables are typically divided into local and environment variables. The difference between them is that when the shell starts another process, called child process, the environment variables are accessible from the child process while local variables are not. The metacharacter $ is used to expand the value of a variable. We can define our own variables, and we will learn how to create a variable when learning shell scripting. For now, let us learn some built-in variables that are used by the shell:

- `SHELL` stores the pathname of the shell
  If we enter
  `echo $SHELL`
  on bluenose, it will output the line:
  `/bin/bash`

- `HOME` stores home directory
  The variable HOME stores the absolute pathname of your home directory. Thus the following three commands are equivalent:
  ```
  cd
  cd ~
  cd $HOME
  ```
- `PATH` stores list of directories to search for commands
  The variable PATH stores a list of directories for the shell to search for external commands. When we run a command without giving its full pathname, the shell first checks whether the command is built-in. If not, the shell searches the directories whose names are stored in `$PATH` for this command. That's how the shell locates the programs for external commands. The directories are separated by the colon character (`:`).
- `PS1` stores default prompt (there are also `PS2`, `PS3`, and `PS4`) — this is bash-specific
  The variable PS1 is used to set the current shell prompt. Similar variables are PS2, which is used as the continuation prompt, PS3 used for the select command, and PS4 used with script debugging. At this point, we will just take a brief look at the PS1 variable. Let us try the following on bluenose:
  `echo $PS1`
  We should get the output:
  `\u@\h:\w\\$`
  which describes the prompt where `\u` is replaced with the username, `@`, and `:`, are literal characters, `\h` is replaced with the hostname, `\w` is replaced with the current working directory, and `\\$` is replaced with the literal `$`. More details about the special sequences for defining the prompt can be found by using the command '`man bash`', and by searching for the string `PS1`. After running the man command (`man bash`) you are viewing the contents using the `more` command and within it, you search for a string by pressing `/` character. So, try the following:
  `man bash`
  then search with:
  `/PS1` $\boxed{\text{Enter}}$
  After finding the first occurence of `PS1` you can press:
  `n`
  to find other occurences. And, of course, you press `q` to exit the man page.
  As an example, if we want to set a very simple prompt `$`, we can use the command:
  `PS1='\\$ '`
  If we wanted to restore the previous prompt, we can either logout and login again, or we can use the command:
  `PS1='\u@\h:\w\\$ '`
  We will learn later more about setting and using shell variables.

Common Shell Variables

- The following are usually common variables among different shells:
- `SHELL` is the full pathname of the login shell
- `HOME` is the full pathname of the home directory
- `PATH` is the list of directories searched for a command
- `USER` is the username
- `MAIL` is the full pathname to the mailbox
- `TERM` is the type of the terminal

11 Processes and Job Control

Programs and Processes

When we run a program, we create a process. Thus programs and processes are related. They are however different. A program is a piece of code which is inactive and stored in a file, while a process is a running program which is active. A process is instantiated from a program.

Process Memory Composition

A process occupies memory space, which is a chunk of memory consisting of the following four parts:

- Code, which stores the program
- Data, which stores static data, i.e., data whose values are maintained throughout the execution
- Heap, which is used for dynamic allocation
- Stack, which is used to store temporary data

When we learn C programming, we will see exactly what types of variables are stored in which part of the memory, and how exactly each part of the memory is maintained.