CSCI 2132: Software Development

Lab 6: Exploring bash and C Compilation

Synopsis

In this lab, you will:

- Customize the behaviour your `bash` shell
- Write and compile some simple C programs
- Practice the edit-compile-fix cycle using `emacs` and the shell
- Learn to patch programs using `diff` and `patch`

Contents

Overview

You will learn a little more about using the `bash` shell and how to set it up. You will use the Unix `diff` command to compare two files and apply the changes in a `diff` file to a given file using the `patch` command. You will write some simple C programs using `emacs` and compile them using `gcc`. Along the way, you will get some experience in using the job control mechanism of the `bash` shell. You will use the `man` command to get information about available C library functions. Finally, you will put your knowledge together in a simple phone number formatter.

Step 1: Login and lab setup

Log into your bluenose account and create a directory lab6 under your directory ~/csci2132/svn/CSID and add it to SVN.

Step 2: The name of your shell

If you are ever unsure about which shell you are running, run

```
$ echo $SHELL
/bin/bash
```

SHELL is an environment variable. The expression $SHELL on the command line substitutes the value of the SHELL variable into the command line. So, `echo $SHELL` prints the value of the SHELL variable, which happens to store the path to the shell you are running. In this case, this is /bin/bash.

Step 3: The .bashrc file

There are a number of initialization steps that you want to perform every time you log into your account on bluenose. One such step is setting up your path, that is, the list of directories that are searched for programs you want to execute. Another possible step is setting up autocompletion options or customizing the prompt the shell displays to you.

The .bashrc file stores shell commands that are executed every time you log in, so you do not have to perform the same initialization steps manually every time you log in. In this step, you will edit your .bashrc file to perform some customizations.

Important warning: *Be extremely careful to carry out the steps in these notes exactly as described. If you make any mistake, this may prevent you from logging in in the future and you will need the help of the Help Desk to reset you .bashrc file to a sane state.*

Step 4: Customize `rm`

One thing you may want to customize is the behaviour of the `rm` command. The default behaviour of `rm` is that it removes its argument file(s) without question. This makes it easy to delete files by accident, without a safety net. The `-i` option of `rm` switches on `rm`'s "interactive mode", which asks for confirmation before deleting a file. We would like to make this the default so we do not have to remember to provide the `-i` option to turn on this safety net.

One way to do this is to use the built-in `bash` command `alias`. The command `alias cmd=XXX` ensures that every time we enter `cmd` on the command line, the text `XXX` is substituted for `cmd`. Thus,

```
$ alias rm="rm -i"
```

ensures that every time we type `rm`, this is replaced with `rm -i`, that is, interactive mode is active. Do not enter this command yet. If you did, you can delete the alias it created using

```
$ unalias rm
```

First, let us check whether `rm` is already an alias:

```
$ which rm
/bin/rm
```

We discussed the `which` command before. It tells us where the shell finds a given command, `rm` in this case. The output says that, whenever we type `rm` as a command to be run by the shell, it runs `/bin/rm`. In particular, `rm` is not an alias yet.

Being able to determine where a given program is installed can be useful for many reasons. One of them is that you may have multiple versions of a program installed, something that is common, for example, with Python interpreters. Even if you have the same version installed in different places, the place from where a given Python version is run usually determines where Python looks for installed packages, so these two versions may still behave differently. If your Python interpreter misbehaves, finding out which one you are running should be your first step.

Back to customizing `rm`. We do not want to set the alias for `rm` every time we log in, so we add the above alias line to our `.bashrc` file, so the alias is set up every time we log in. In order to keep track of your changes as part of this lab, start by copying you `~/.bashrc` file into your `lab6` directory and rename it to `bashrc.old`:

```
$ pwd
/users/cs/CSID/csci2132/svn/lab6
$ cp ~/.bashrc bashrc.old
```

Now make a copy of `bashrc.old` with the name `bashrc.new` and add both files to SVN. Open `bashrc.new` in `emacs`. Add the following lines to `bashrc.new` (after the already existing content, if any):

bashrc.new

```
umask 077
alias rm=”rm -i”
alias mv=”mv -i”
alias cp=”cp -i”
```

Warning: *Be careful to copy the contents of the file exactly as shown. You will replace your* `.bashrc` *file with it later. Any errors in this file may prevent you from logging into your account without the Help Desk first restoring your* `.bashrc` *file.*

All the commands you added to `bashrc.new` can also be run directly from the command line. Here is a brief explanation what they do:

- We already discussed that setting an alias from `rm` to `rm -i` ensures that, every time we remove a file, `rm` asks for confirmation. The aliases for `mv` and `cp` serve a similar purpose. If we try to move or copy a file to a destination file that already exists, the `-i` option of these commands ensures that they ask for confirmation because this replaces the contents of the destination file.

- Recall that, when you created the `csci2132` directory in Lab 1, I told you to change its permissions to `go-rwx` so nobody else can access the files in this directory. Setting `umask 077` ensures that, from now on, every file or directory you create is inaccessible to anybody but you. `umask` stands for "user mask".

  Recall from class that we can write the permissions of a file in octal. Without `umask` (i.e., `umask 0`), every regular file would be created with permission `rw-rw-rw-` and every directory would be created with permission `rwxrwxrwx`, that is, everybody can read and write your files and directories and everybody can change into your directories. In octal, this means that your files have permission `666` and your directories have permission `777`. When setting a non-zero user mask using `umask XXX`, every new directory you create will have permission `777 &` $\overline{\texttt{XXX}}$, where `&` is the bitwise-and operation and $\overline{\texttt{XXX}}$ denotes the bitwise negation of `XXX`. So, with a `umask` of `077`, all newly created files have their permissions set to `777 & 700` = `700` = `rwx------`. All newly created files have their permissions set to `666 & 700` = `600` = `rw-------`.

  The default `umask` is `022`, which only takes write permission away from everybody but you but allows them to read your files and directories and navigate your directory hierarchy.

Verify that your `bashrc.new` file is valid, by running the command:

```
$ source bashrc.new
```

This executes all commands in `bashrc.new` as if you had typed them on the command line one by one. This should not report any errors. Check carefully once more that you copied the lines above *exactly* to your `bashrc.new` file. Next check that the aliases were created as expected:

```
$ which rm
alias rm='rm -i'
        /bin/rm
```

Check your `umask`:

```
$ umask
0077
```

If your output looks as in these two examples **and you did not get any errors from** `source bashrc.new`, everything should be fine. Add your `bashrc.new` file to SVN and commmit your changes.

Optional Step: Install `bashrc.new` as your new `.bashrc` file

This step is "dangerous"; it may break your ability to log in if there are errors in your `.bashrc` file.

Copy your `bashrc.new` file to `~/.bashrc`:

```
$ cp bashrc.new ~/.bashrc
```

Now test your newly installed file. Make sure you **do not log out** yet. Log into your `bluenose` account in a second window (by opening a second `PuTTY` session or opening a new terminal window and logging in using `ssh`; this is similar to what you did in Lab 3).

First of all, you should not see any errors in this second window, that is, you should have been able to log in without problems. If not, use your first window to restore your `.bashrc` file from `bashrc.old`:

```
$ cp bashrc.old ~/.bashrc
```

After doing this, verify that everything is back to normal by logging in a third time. Things should work smoothly again. You account is back in a sane state and you can investigate what is wrong with your `bashrc.new` file that it prevents you from logging in.

If your login attempt in the second window succeeded, you can check that the commands in `.bashrc` have the desired effect. Immediately after logging in, run

```
$ which rm
/bin/rm
$ umask
0022
```

This is the likely output you will see, that is, somehow your shell ignores your spiffy new `.bashrc` file. This is because `.bashrc` is not run by default when you log in. What is run on login is in fact controlled by your `.profile` file. You cannot just replace `.profile` with your `bashrc.new` file because `.profile` is run by many different shells, so `.profile` is used to triage to appropriate shell-specific files such as `.bashrc` for the bash shell. The next step shows you how to do this.

Step 5: Edit `.profile`

Follow the same steps as when editing your `.bashrc` file. Copy your `~/.profile` file to `profile.old` in your `lab6` directory and make a copy, also in the `lab6` directory, with name `profile.new`. Add both `profile.old` and `profile.new` to SVN. Now use `emacs` to edit `profile.new` to look *exactly* like this:

profile.new

```
case `basename $SHELL` in
  sh|jsh)
        . $HOME/.shrc
        ;;
  ksh)
        . $HOME/.kshrc
        ;;
  bash)
        . $HOME/.bashrc
        ;;
esac
```

In particular, make sure that you use backquotes (`` ` ``) instead of regular quotes in the first line. This finds the name of the shell you are running using `basename $SHELL` and then matches it to a number of different known shell names. Depending on which one matches, it runs the appropriate shell-specific script similar to the way a `switch` statement in Java would do.

It is possible that the file already contains these or similar lines, in addition to a few comment lines. In that case, the previous optional step should already have reported `rm` as being an alias and should have reported the desired umask of `077`. You can leave your `~/.profile` file intact in this case.

Test that the file behaves as expected:

```
$ source profile.new
```

If you performed the optional step of replacing your `.bashrc` file with `bashrc.new`, this should not produce any errors and running `which rm` and `umask` should now produce the following results:

```
$ which rm
alias rm='rm -i'
        /bin/rm
$ umask
0077
```

If you did not replace your `.bashrc` file with `bashrc.new`, then you may not have a `.bashrc` file and sourcing your `profile.new` script may report an error to this effect. To eliminate this error, you may create an empty `.bashrc` file by running

```
$ touch ~/.bashrc
```

Sourcing `profile.new` again should not raise any errors now.

Once you are able to source `profile.new` without errors, commit your work to SVN.

Optional Step: Install `profile.new` as your new `.profile` file

This step is "dangerous"; it may break your ability to log in if there are errors in your `.profile` *or* `.bashrc` *files.*

We will follow the same strategy as when installing a new `.bashrc` file, using two windows, one to try out the new files, one where we have the opportunity to restore old versions if things go wrong.

In your current window, copy your `profile.new` file to `~/.profile`:

```
$ cp profile.new ~/.profile
```

Now test your newly installed file. Make sure you **do not log out** yet. Log into your `bluenose` account in a second window.

You should not see any errors in this second window, that is, you should have been able to log in without problems. If not, use your first window to restore your `.profile` file from `profile.old`:

```
$ cp profile.old ~/.profile
```

After doing this, verify that everything is back to normal by logging in a third time. Things should work smoothly again. You account is back in a sane state and you can investigate what is wrong with your `profile.new` file that it prevents you from logging in.

If your login attempt in the second window succeeded, you can check that the commands in `.profile` and `.bashrc` have the desired effect. Immediately after logging in, run

```
$ which rm
alias rm='rm -i'
        /bin/rm
$ umask
0077
```

This is the output you should see. If so, everything went well. Otherwise, you have to do more debugging as in the case when you get errors logging in.

Step 6: Write a simple C program

Write a simple “Hello world” program in C, using emacs:

hello0.c

```
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
```

Make a copy hello1.c of it and change the line `return 0;` so it returns 1 instead.

Add both hello0.c and hello1.c to SVN.

For good measure, you should also compile the programs and run them:

```
$ gcc -o hello0 hello0.c
$ gcc -o hello1 hello1.c
$ ./hello0
Hello, world!
$ ./hello1
Hello, world!
```

Recall that `gcc hello0.c` and `gcc hello1.c` would place their output into a file a.out. Using the -o option, we tell gcc where to place its output. Thus, the first invocation of gcc produces a program hello0. The second invocation produces a program hello1.

Step 7: Explore exit codes

Recall that a program's exit code is the way it signals various outcomes to the shell, with 0 meaning success and any non-zero value meaning any result that may require your attention.

A C program has two ways to provide an exit code to the shell. The first one is the value returned by the `main` function. Try this out:

```
$ ./hello0; echo $?
Hello, world!
0
$ ./hello1; echo $?
Hello, world!
1
```

The other option is to call the C standard library function `exit()` from anywhere in your program. This makes the program exit immediately and provide as exit code the argument passed to `exit()`. To use `exit()`, you need to make it available by including the `stdlib.h` header file.

Create a third program `hello2.c` that looks like this:

hello2.c

```
#include <stdio.h>
#include <stdlib.h>

void do_stuff() {
  printf("Just doing stuff.\n");
  exit(2);
  printf("Done doing stuff.\n");
}

int main() {
  printf("Hello, world!\n");
  do_stuff();
  printf("Hello again!\n)";
  return 0;
}
```

What output do you expect if you compile and run this program? What exit code do you expect to see? Try to answer this question before running the program. The answer is on the next page.

```
$ gcc -o hello2 hello2.c
$ ./hello2; echo $?
Hello, world!
Just doing stuff.
2
```

The last line in `do_stuff()` and the last two lines in `main()` are never reached because of the call `exit(2)`. And, as expected, the exit code of the program is 2.

Step 8: `diff`

The `diff` tool is useful for comparing the contents of two files. Later in this course, you will use it to check that your code produces exactly the specified output. It is also useful to compare the changes you and a colleague made to a source code file and to merge these changes.

To compare two files, run

```
$ diff hello0.c hello0.c
```

As you should have expected, this produces no output because the two input files are obviously identical. The more interesting case is

```
$ diff hello0.c hello1.c
5c5
<   return 0;
---
>   return 1;
```

This shows that line 5 is different between the two files, as expected. Read pages 93 and 94 in the UNIX textbook to learn more about the `diff` output.

If you provide the `-u` option to `diff`, you get an output that is analogous to the output produced by `svn diff`:

```
$ diff -u hello0.c hello1.c
--- hello0.c 2018-12-31 13:21:54.351701970 -0400
+++ hello1.c 2018-12-31 13:09:55.933713226 -0400
@@ -2,5 +2,5 @@

 int main() {
   printf("Hello, world!\n");
-  return 0;
+  return 1;
 }
```

It displays the files that were changed, the location of each change, and some context around each change.

When trying to use `diff` as part of a source code management workflow, `diff -u` is very useful. The file produced by

```
$ diff -u hello0.c hello1.c > hello0.patch
```

is known as a *patch file*. The next step will teach you a little about how to use these files to manage changes to source code.

Step 9: `patch`

Patch files play a crucial role in software management systems such as `quilt`, used by some Linux distributions to maintain patched software versions, or early versions of Git.

The key to their power is that they can be “applied” to source files to make the changes stored in these patch files. The patch file `hello0.patch` we created in the previous step stores the differences between the `hello0.c` and the `hello1.c` files. More precisely, it stores the edits that need to be applied to transform `hello0.c` into `hello1.c`.

By running

```
$ patch < hello0.patch
patching file hello0.c
```

we change the contents of the `hello0.c` file according to the changes stored in the patch file. It is now identical to `hello1.c`:

```
$ diff hello0.c hello1.c
```

produces no output.

It is not hard to imagine why patch files are useful:

- When you make changes to large software packages, it may be unwieldy to ship a completely new distribution of the software every time you make a small change. If the number of changes is small compared to the total code size, it is much more efficient to send a bunch of patch files that store all the changes that were made and apply these patch files at the receiving end.

- `patch` is quite resilient in that it can apply patches even if the patch file does not exactly “match” the file to be patched. (It will warn you if it finds the pattern to be replaced in a location that is a few lines off.) This is the property that the patch management system `quilt` relies on. Whenever a new upstream version of a given software package is available, the person maintaining this package as part of a Linux distribution can often apply the existing patches to the new code base without problems. This is much more convenient than editing the new code base by hand to make all the changes that were applied to the previous version of the software to integrate it into the Linux distribution.

Step 10: Edit, compile, and fix with emacs

If you are used to using an IDE to support your programming workflow and are ready to complain about having to switch between emacs and the terminal to edit and compile your code, the hardcore Unix enthusiast will respond that your Unix desktop with editor, shell, and debugger *is* your IDE. This is of course mainly a matter of personal preference, but no matter whether you prefer powerful IDEs, it is useful to know how to program using editor and shell in case your favourite IDE is not available. There is nothing that prevents you from having one window open with an emacs session and another that you use to work directly at the command line. Still, it would be nice if you could at least compile your programs directly from the editor. Especially if you log in remotely as we do here, it is a bit tedious to start two PuTTY sessions or run ssh twice.

In this step, you will learn how to switch between emacs and your shell in the same window without having to restart emacs again and again. The answer is, of course, your shell's job control. In the next step, we explore how to compile your program directly from emacs, without interrupting your emacs session at all.

Open the `hello0.c` file and introduce an error by editing the `return 1;` line so the program returns 0 again, but you forget to add the semicolon at the end:

hello0.c
```
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0
}
```

Save this file. To compile it, **do not** exit emacs. Instead, suspend it using `C-z`. This gets you back to your shell prompt.

You can check that emacs is still running:

```
$ jobs
[1]+  Stopped                 emacs hello0.c
```

Now try to compile your program:

```
$ gcc -o hello0 hello0.c
hello0.c: In function ‘main’:
hello0.c:6:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

To fix your program, enter emacs again. You **do not** want to start a new emacs session. You want to resume the one you suspended using `C-z` before:

```
$ fg
```

Now add the missing semicolon, suspend emacs again, and compile your program at the shell prompt. Once it all works, resume `emacs` using `fg` again and then make sure you actually exit emacs using `C-x` `C-c`.

Practice this edit, compile, fix workflow by making a few more changes to your `hello0.c` file, compiling it, and fixing any errors the compiler complains about.

Step 11: Compile programs from within emacs

emacs allows you to run any shell program from within emacs. Open your `hello1.c` file in emacs. Then change the `return` statement to `return 0`. After all, there is nothing wrong with this program, so the exit code should be 0.

Now, **do not** exit emacs. **Do not** suspend it either. Instead, enter emacs's shell prompt using `M-!` (or `Esc` `!`). Then enter `gcc -o hello1 hello1.c` and press `Enter`.

emacs executes the command you enter at the prompt after pressing `M-!` in a shell that runs inside emacs and displays the output. Once you press any key, the output goes away. In this case, the output you should see is

```
(Shell command succeeded with no output)
```

unless you made a mistake and your file contains errors.

A more convenient way to compile programs from within emacs is by pressing `M-x` or `Esc` `x` to enter emacs's command prompt. Then enter the command `compile` and press `Enter`. emacs will now display the command it intends to use to compile your program (most likely "`make -k`"). This can once again be any shell command. You can edit this, to `gcc -o hello1 hello1.c` in this case, and then press `Enter` to execute the command.

This opens a second emacs window with the output of the compilation displayed. This is useful in case there are errors. If you run gcc using `M-!`, then the error messages go away the moment you start typing. When fixing errors in your code, it is useful to have the error messages on hand *while* making the required changes, so having a separate window with all error messages in it is useful.

For now, close the message buffer using `C-x` `1`.

There is a second reason why compiling your program using `M-x` `compile` instead of `M-!` is more convenient: you do not have to re-enter the command the next time you run it; emacs remembers your last choice. Thus, if you compile more than once, `M-x` `compile` `Enter` `Enter` is likely faster than re-entering the shell command every single time.

Finally, you can run multiple commands in a row, as at the shell prompt. Set up your compilation process so it also runs the program, provided the compilation was successful:

`M-x` `compile` `Enter` `gcc -o hello1 hello1.c && ./hello1` `Enter`

Exit emacs using `C-x` `C-c`.

Step 12: Learn about C library functions using `man`

The C standard library is fairly extensive, not as extensive as those of other programming languages, but still substantial. Thus, you will most likely not remember the arguments to functions you do not use often.

Just as with Unix commands, the manpages are the information source you always have at your fingertips.

Try to find out about `printf()` using the `man` command:

```
$ man printf
```

The output should be fairly confusing because it shows you information about a command line utility with that name.

To see the information about the C function `printf()`, you need to use

```
$ man 3 printf
```

This tells `man` to look in Section 3. The manpages are organized into numerous sections, Section 1 being reserved for commands, Section 2 for system calls, Section 3 for C library functions, Section 5 for configuration files, and so on.

Now look up information about `scanf()`:

```
$ man scanf
```

This *does* display the desired information. The reason is that Section 3 is the first section that has a `scanf` entry whereas an entry with name `printf` is found already in Section 1.

Step 13: Practice using `printf` and `scanf`

This final step is Project 4 on page 50 of the C textbook. Your task is to write a C program that asks the user ot enter a phone number in the format `(xxx)xxx-xxxx`. It prints the phone number back to `stdout` in the format `xxx.xxx.xxxx`.

An example run should look like this:

```
$ ./phoneconvert
Enter a telephone number [(xxx)xxx-xxxx]: (902)494-9999
The number is 902.494.9999
```

Use what you learned in class about the `printf` and `scanf` functions, information about `printf` and `scanf` found in the manpages, and the skills you have acquired in this lab in working with `emacs` and the command line to edit and compile C files to implement such a program in a file `phoneconvert.c`.

The behaviour of the program given a valid phone number should be self-explanatory given the example above. If the user enters something that is not a valid phone number (or at least not in the expected format), your program should output

```
ERROR: Not a valid phone number
```

Explore how to use `scanf`'s return value to decide whether the entered phone number is valid.

Step 14: Commit your work

Commit your work to SVN. The files you should commit are

```
lab6
├─ bashrc.old
├─ bashrc.new
├─ profile.old
├─ profile.new
├─ hello0.c
├─ hello1.c
├─ hello2.c
├─ hello0.patch
└─ phoneconvert.c
```