CS 1XC3 Lab 6: Environment Variables and Libraries

Mar. 6^{th}-10^{th}, 2023

Contents

Introduction:

In this exercise, we are going to explore the way Unix-like operating systems manage their programs and libraries through command-line shells.

Once again, your deliverables will need to be submitted through your CS1XC3 repository, in a new "L06" directory. No templates to download this time, however!

Redirecting STDOUT to a File

At various points in this lab, you will be asked to create a file whose contents is the result of the command you just entered. There are two ways to do this, the bad and smelly Windows way and the good and prosperous Linux way.

1. The Windows way is to use a GUI to copy and paste the text from your terminal into a file. Obviously this doesn't work if your terminal isn't embedded in a GUI (as with the pascal server for instance).

2. The Linux way is to redirect the stdout stream to a specified file.

```
$ [Some command] > [Some File]
```

Everything that the command outputs will be stored in the file *instead* of being displayed in the terminal. We can always view the file afterwards with `cat` or `less`.

If you want to append an existing file, use the following.

```
$ [Some command] >> [Some File]
```

1 Activity: Introduction to Environment Variables [5 points]

Environment variables are used by your Bash shell environment to keep track of information vital to the correct execution of your commands. Let's go through a few of them and describe what they do.

1. Display the environment variable `HOME`, and write the output to a new file, "part1.txt".
   - `HOME` gives the path to the user's home directory.
2. Display the environment variable `PWD`, and append the output to "part1.txt".
   - `PWD` gives the path to the current working directory. This can also be displayed using the `pwd` command.
3. Display the environment variable `USER`, and append the output to "part1.txt".
   - `USER` gives the path to the user's home directory.
4. Display the environment variable `PATH`, and append the output to "part1.txt".
   - `PATH` lists all the paths Bash will look in for a command requested by the user. If a command is not on `PATH`, you'll get an error!

   Errors are not stored in `stdout`, but in `stderr`. To redirect `stderr` to a file, you either need to use the following, which captures just `stderr`:

   ```
   [Some command] 2> [Some File]
   # Overwrite mode
   [Some command] 2>> [Some File]
   # Append mode
   ```

   Or the following, which captures both `stdout` and `stderr`.

   ```
   [Some command] &> [Some File]
   # Overwrite mode
   [Some command] &>> [Some File]
   # Append mode
   ```

5. Use an invalid command of your choice, and append the error message to "part1.txt"
6. All the environment variables currently active in your session can be viewed using the `env` command. Append the output of `env` to "part1.txt".

To complete this activity, commit your "part1.txt" file to the "L06" directory in your CS1XC3 repo.

2 Activity: Bash Startup Scripts [5 points]

Now that we know a thing or two about bash scripting, let's put that knowledge to use!

1. Take a look at `/etc/profile`. This file is executed by Bash before the user profile script. This global profile is run for all users, so any changes here will effect everyone using the system.
   - If you're on the pascal server, that means all those other people on Pascal!
   - You can view the other students on Pascal with the `w` command.

   Write the contents of `/etc/profile` into a new text file, "part2.txt".
   - You may get a "permission denied" error here. Figure out a way to solve this problem without invoking super user!
     - For one thing, people on the server won't even have super user privileges.
     - For another, it's a terrible idea to leave random text files lying around your operating system folders. Cleanliness is next to Godliness!
2. Next, append six hyphen characters to the file "part2.txt".
3. Now, append the results of the `w` command to "part2.txt".
4. Now let's worry about our own user profile. In general, there are two files that are run on Bash startup that control user configurations. Both these are hidden files within your user home directory.

- `.bashrc` runs when the user opens **interactive non-login shells**
- `.bash_profile`, `.bash_login` or `.profile` are run when the user opens an **interactive login shell**.
- `.bash_logout` runs when the user logs out of a login shell (such as the Pascal server).

You can tell which of these two you are using by whether you had to log in to open up the terminal!

- Incidentally, the correct way to quit a login shell is using the `logout` command...

We're going to make some modifications to `.bashrc` that will tweak our bash interface. Specifically, we're going to remove some redundant information from the bash prompt to shorten it. By default, Bash displays the full path of the current working directory within the prompt itself.

```
username@hostmachine:/sometimes/your/workflow/requires/a/large/number/of/nested/
    directories $
```

If you have deeply nested directory structures, this can mean that the Bash prompt doesn't leave enough room on the line for your commands! Bash tends not to support multi-line commands very well, so let's shorten it!

```
username:directories $
```

You may not have had this problem yet yourselves, but your professor certainly has.

The environment variable `PS1` controls the appearance of the bash prompt itself. This variable will be set in various places depending on your distribution, so we're going to simply overwrite however it's been set up.

Select the profile file appropriate to your situation as indicated above, and open it with your favourite editor (emacs of course).

Add a line to then end of the file setting `PS1` to a custom string value, using the following shortcuts:

- `\u` will be substituted for the username of the current user.
- `\h` will be substituted for the hostname of the current machine.
- `\w` (lower case) will be substituted for the *full* path to the current working directory.
- `\W` (upper case) will be substituted for only the name of the current working directory (i.e., just the last directory in `PWD`).

Using the above information, set a value for `PS1` which displays the shortened bash prompt shown above. To see the changes, you will need to logout or exit your terminal and log back in again.

5. Also, change the iconic bash $ to any other single character of your choice (but be sure to leave the space afterwards!). Have some fun with it, but make sure it renders: `https://en.wikipedia.org/wiki/List_of_Unicode_characters`
6. In addition to setting `PS1`, add a short shell script to perform the following actions:
   - Enter the directory your CS1XC3 git repo has been cloned to.
   - Pull the latest changes.
   - Return to the home directory.
7. Write a script in `.bash_logout` which performs the following actions in the following order:
   (a) Enter the directory your CS1XC3 git repo has been cloned to.
   (b) Stage all changes in all subdirectories for commit.
   (c) Commit the staged changes with the commit message "Shell Logout Commit".
   (d) Push the committed changes.
8. Copy the two files you modified into your L06 repo directory and commit them.

NOTE: If you find the above operations useful, feel free to keep them. Otherwise, feel free to revert these scripts to their previous state. This is meant as an illustration of how powerfully a bash environment can be automated.

3 Activity: Manual Installation via Adding to Path [5 points]

In this part, you will manually install a C program, so that it may be run from the command line in any directory, like any other Bash command.

1. Download "lab6.tar.xz" from the course content on Avenue and unpack it into a newly created subdirectory `.../CS1XC3/L06/part3/`. For more detailed instructions, please see the setup section of Lab 5.

2. Compile "solitaire.c", and produce an executable binary file named "solitaire".

3. Modify the appropriate Bash startup script for your shell environment to extend `PATH` to include this new directory. The line you add to your startup script should look something like this:

   ```
   PATH="$PATH:/path/to/executable/"
   ```

   - NOTE: it is very important that the previous PATH directories are still there. *If you delete the information in PATH instead of appending it, your computer won't be able to find its own programs, like* `cd`, `ls`, *etc.* The colon character separates different paths within `PATH`, and the current value of `PATH` must be directly invoked using the `$` character.
   - If you mess this up in a destructive way, you likely won't realize it until the next time you log in. If you log in and none of your commands work, you have one of two options:
     - Try to access the commands you need to fix your startup script by directly addressing the commands themselves in the operating system.
     - Email Sebastian (`mailto:rabiejs@mcmaster.ca`), John (`mailto:nakamura@mcmaster.ca`), *or* Derek (`mailto:lipiec@mcmaster.ca`). Tell them you broke your user account on the Pascal server "through nothing but my own foolishness", and you need them to delete it. Express to them how very sorry you are for the inconvenience, and that you will never ever do it again. When you log in again, a fresh account will be created for you. You will need to set your git repo up again, but that's child's play at this point, right?

4. Once your startup script has been modified, log out of your shell and back in again for the changes to take effect.

5. Try executing solitaire from the command line from a directory other than the one you just added to `PATH`:

   ```
   $ solitaire
   ```

   Notice you no longer need to include `./`

6. For your deliverable in this part, copy your startup script into your "L06" directory, and rename it "part3.txt". Make sure it does not have executable permissions enabled.

4 Activity: Static Libraries using GCC [5 points]

Now that we know something of program installation, let's explore the `gcc` library system. But first! Let's recall what an **object file** is.

4.1 Object Files

Object files are the output of the assembly phase of compilation (recall Topic 3, slide). This means an object file has all its functions compiled into machine code, but the file has not been linked to other source files. Instead, object files contain references to other functions in other files which will be used during linking to produce the final executable. These references are resolved during the **linking** phase of compilation.

Like many compilers, `gcc` won't normally bother writing object files to your filesystem, as most of the time the linker will be invoked immediately afterwards anyways. We can ask `gcc` not to link the files by using the `-c` flag.

4.2 Static Libraries

A static library (also known as an **archive**) is a collection of object files which can be loaded as a unit into `gcc`. This allows us to design libraries with an arbitrarily large number of files, which can be very beneficial if you're the type of person who uses their file system as a system of organization. Static libraries have some advantages over dynamic libraries:

- They are a bit faster, because they already have everything they need to run.
- Once produced, the executable will aways use the version of the library you compiled with it, so you don't have to worry about a new version breaking your code.

1. Inside the "lab6.tar.gz" compressed archive you downloaded earlier, you will find four files: "top.c", "sums.c", "products.c" and "C_Shanties.c". Extract these (if you haven't already), and move them to a new folder within your CS1XC3 repo: "L06/part4"
2. You may notice that we're making a library, but we're missing a header file! Create a header file, "Lab6Part4.h", which contains function prototypes for all of the functions in "sums.c", "products.c" and "C_Shanties.c". Do not have this header file include any other files. Modify "top.c" so that it includes your new header file.
3. Compile "sums.c", "products.c" and "C_Shanties.c" to object files using `gcc`.
4. We can use object files directly during compilation without creating a static library archive. To do so, simply type:

   ```
   $ gcc top.c sums.o products.o C_Shanties.o -o top
   ```

   Using this method, we have to specify each object file in the command line separately! This is fine for beginners, but doesn't scale well.
5. Let's make an archive! The GNU archive program `ar` is used to produce `*.a` archive files, which gcc knows to look for.

   ```
   $ ar -rc libLab6Part4.a *.o
   ```

   - The `-r` flag means "if the archive already exists, replace it!"
   - The `-c` flag means "if the archive doesn't already exist, create it!"
6. Delete all your object files. In general, it's considered good practice to delete object files once you're finished with them. (This is part of the "cleanup" phase).
7. Now, let's compile our program using this new library archive!

   ```
   $ gcc top.c -L. -lLab6Part4 -o top
   ```

   - The`-L` flag means "Look for libraries here!"
   - The `.` after `-L` means "current working directory," in the same way that `..` means "up one directory." You can see both when you type `ls -a`.
   - The `-l` flag means "Link with this library"
   - You may notice that the name of the library we are linking is included immediately afterward, except that the `lib` prefix, and `.a` suffix have been left out.
8. If all goes well, your executable will be produced without any errors or warnings. Give it a try!
9. For this section, commit your archive file as your deliverable.

5 Activity: Dynamic Libraries using GCC [5 points]

Now, let's learn how to create a **shared object library**. Shared libraries are different from static libraries because they are linked when you run the program, not when you compile it. This means a program using dynamic libraries can incorporate updates to called functions (in the form of updates to the library) without needing to recompile the program. This is of particular interest if you are distributing a software product, since it's not normally a good idea to have the end user compile their own programs from source (particularly if they're Windows users). It gets a bit trickier than static libraries though, because the compiler has to know where to look for the shared library.

1. Create a new folder, "dynamic" within your L06 folder, and copy all the C source files from part 4 into it.
2. Next, we need to create a special version of our object files from part 4, *position independent code*.
   - Making code position independent just means that we modify the machine code so that any jumps with it are relative to the position of the jump in the source code, rather than absolute on the position of the code in the file.
   - If jumps are absolute, we can't mix and match code post-compilation!

   Compile the three library source files to object code with the `-fpic` flag to enable position independent encoding.
3. Next, we need to create the library file itself. Last time we used the archiving program `ar` to do this, but this time we use `gcc` itself:
   ```
   $ gcc -shared -o libLab6Part5.so .o
   ```
   Remember that the C convention for libarary names is to include the prefix `lib` and, in the case of shared libraries, an `.so` suffix. Don't forget to clean up those object files!
4. Now let's compile our program!
   ```
   $ gcc -Wall -o top top.c -lLab6Part5
   ```
   Uh-oh! That doesn't seem to have worked! The linker doesn't know where to find our library! Capture the output of this command, and write it to the text file "part5.txt".
5. Again, we have to use the `-L` flag to specify the location in the file system where `gcc` can find the library, since it's not in any of the places `gcc` knows to look for it by default. The path to the library must be provided immediately after the `-L` flag with no spaces:
   ```
   $ gcc -L/path/to/library/file -Wall -o top top.c -lLab6Part5
   ```
   This should run with no errors. Test the program out by running it.
6. When you run the test, you should get another error message. Append this error messgage to your "part5.txt" file.
7. The problem is that the loader doesn't know where to find the shared object file we just created (the loader of course being different from the compiler). In general, there are three ways to get around this problem.
   - By setting the environment variable `LD_LIBRARY_PATH`.
   - By embedding the path in the executable itself using `rpath`.
   - By "properly" installing our library in one of the default directories `gcc` uses by default. This requires super user privileges, so we will discuss this method, but you won't be required to use it in this activity. (You are encouraged to do so, however, if you happen to be a super user).

5.1 Using `LD_LIBRARY_PATH`

This is very similar to setting `PATH` in part 3, but with one exception. We can't assume that "top.c" is the only program using `LD_LIBRARY_PATH`, so we can't just set it up from a startup script and forget about it.

1. Write a shell script "runtop.sh" which appends the directory containing the shared library to `LD_LIBRARY_PATH` (and exports it), runs the program, and then resets the variable using `unset`.
2. Test your script to ensure that it works.
3. Commit your shell script as part of the deliverables for this section.

5.2 Using `rpath`

While programs compiled using the run path option (`rpath`) to embed the library path in the executable don't require the setting of an environment variable, they do require that the library never be moved from the specified location.

1. To do so, add the `-Wl,-rpath=/path/to/library/file` option.

```
$ gcc -L/path/to/library/file -Wl,-rpath=/path/to/library/file -Wall -o test main.c
    -lLab6Part5
```

   Now, execute the program to make sure that it works.

2. Notice the repetition of the file paths? This would be a good time to start using shell scripts! Write a shell script, "compiletop.sh" which performs the above compilation, with the path to the shared object library as the first argument. If the user hasn't provided any arguments to the script, display a message asking them to provide one, and what its purpose is.

3. Test this script to make sure it works, and then commit it as part of the deliverables for this section.

5.3 Doing it like a Super User

The above solutions are fine, but kind of a pain, since you're constantly having to tell `gcc` where to find things. If you just put the library where `gcc` expects it, everything will be fine!
Everything beyond this point requires sudo priveleges and will not be graded.

1. First, we need to copy our library into one of the standard library directories, and modify its permissions so that everyone can read it:

```
$ cp /path/to/library/file /user/lib
$ chmod 0755 /usr/lib/libLab6Part5.so
```

2. Now, we need to update the cache and register the library so that the loader knows to look for it:

```
$ ldconfig
```

   You can check that this process worked using:

```
$ ldconfig -p | grep Lab6Part5
```

   Congratulations! You have just installed a library!

3. Now, relink the executable:

```
$ gcc -Wall -o top top.c -lLab6Part5
```

   Notice we don't need to specify where the library is, although we do still need to specify which library we're using.

   We can confirm which shared libraries our executable is using with `ldd`.

```
$ ldd top
```

4. Now, test your program by running it.

5. There is no deliverable for this subsection.

5.4 Deliverables

For this part of the lab, you must commit "part5.txt", your console output file, "runtop.sh", which sets `LD_LIBRARY_PATH`, and "compiletop.sh", which uses `rpath` to embed the library path in the executable.