Faculty of Computer Science, Dalhousie University *20-Sep-2018*

CSCI 2132 — Software Development

Lab 3: SVN Source Version Control Tutorial

Location: Teaching Labs Instructor: Vlado Keselj
Time: Thursday

Lab 3: SVN Source Version Control Tutorial

Lab Overview

– Learning about SVN Source Version Control system
– Checkout projects in SVN
– Add a subproject to SVN
– Add and delete files to SVN
– Change files in SVN
– Retrieve previous versions of your code
– Identify what has changed between versions of your code
– Identify what changes other people have done to your code

Slide notes:

Source Control with SVN

– *Subversion* or *SVN*) source control system
– SVN can be accessed using command line interface on bluenose (Linux system); similarly on Mac
– GUI interfaces exist (including Windows)
– Development companies use some form of source control
– SVN is relatively popular, open-source system
– git is another newer open-source system (e.g., used for Linux kernel)
– Some commercial software: Perforce and ClearCase
– Predecessors: SCCS, RCS, CVS

In this lab you will familiarize yourself in more details with the *Subversion* (or *SVN*) source control system. While many people interact with SVN using a GUI interface (often part of their development IDE), we can access SVN with command line parameters on bluenose.

The SVN repository that we use for this course is setup in such way that the course instructor and TAs have access to your repository and SVN is used for submission of assignments in the course. You may see a similar SVN tutorial in several CS courses that use SVN for assignment submission. The tutorial was originally developed by Michael McAllister for CSCI 3171. Some adaptations are done for this course.

All development companies use some form of source control, so becoming familiar with the basic notions of source control is useful. You will not be a master of source control after this lab, but you will have at least seen some of the basic concepts and commands. Subversion is relatively popular. A more recent source control system is called "git" while Perforce and ClearCase are two popular commercial offerings. Older source control systems include SCCS, RCS, and CVS (in case you hear about them).

As usual, we will start by logging in into the server bluenose.

Step 1: Logging in. The first step is similar to the previous labs:

1-a) Login to server bluenose.cs.dal.ca via SSH from a CS Teaching Lab computer or from your own computer. If you use Windows, you can use PuTTY.

1-b) Change your current working directory to your main SVN working copy directory. It should be the following directory: `~/csci2132/svn/`*CSID* where *CSID* is your CSID.

1-c) Run the SVN update command as follows:

```
svn update
```

The command "`svn update`" will bring any possible changes in the SVN repository to your local working copy. If some files are changed, you should know why it has happened or you may want to check if the files are added or changed by the instructor or the TAs.

1-d) You can create the `lab3` directory in the SVN and save it in the SVN repository using the commands:

```
svn mkdir lab3
svn commit -m'Directory lab3 created'
cd lab3
```

You should understand by now what these commands mean.
Using these steps we prepared the `lab3` directory. In this lab, we will use two windows to work with the SVN repository. We will call this window the **primary** window, and this working copy as you **primary** working copy. You can always recognize it by the fact that this is the working copy of your whole SVN repository created earlier.
We will also create a secondary copy in the next step.

Step 2: Secondary working copy. Now, you need to open another window in which you will have another working copy that we will call the **secondary working copy.**

2-a) Login to server bluenose.cs.dal.ca via SSH in another window. If you use PuTTY, you will run another PuTTY windows.

2-b) Change your current working directory to `~/csci2132/`. Create a subdirectory named `tmp` inside directory `csci2132`, and change your current directory to this directory '`tmp`'.
If you run the '`pwd`' command, it should show that your current path is::

```
~/csci2132/tmp
```

where '~' is your home directory.

2-c) Check out only your `lab3` directory.
Your secondary copy will be in your `~/csci2132/tmp` directory and you can create it using the command:

```
svn co https://svn.cs.dal.ca/csci2132/CSID/lab3
```

where *CSID* is replaced with your CSID.
As you can see, your secondary working copy will contain only the `lab3` subdirectory in your SVN repository. SVN allows this: you can check out only a subdirectory in your project. For example, this is not allowed in git.
Remember that we will call this window the **secondary** window, and this working copy the **secondary working copy.** You will be able to recognize it by the fact that it is inside the directory called '`tmp`'.

We will use two windows and two directories in this lab to show how source code is shared between two copies. These copies are typically copies of two different users, but they may also be two code copies of the same user.

A Couple Notes (Reminders)

Note that in the commands for this lab, the userid *CSID* will often be used. Make sure to always replace it with your own CS userid; i.e., your userid on bluenose (CSID).

Important Security Note: Each time that you enter an SVN command in this lab, you will be asked for your bluenose password. Enter it. When asked if you want to store the password, answer "no" each time. While it does require that you frequently enter your password, the password is not stored safely for now so we do not want you to store it. Tech support is working to fix this problem.

If you would like to make your work easier and not have to enter the password each time, and you also confident that you will follow the following instructions, you can do the following: You can say “yes” to SVN to save your password, but then make sure that only you have access permissions to the the directory `~./subversion`, and it is a good idea to delete the whole directory after each session. If you decide to do this, be aware that your password is saved in a plain text in a file somewhere in the directory `~/.subversion`.

Concepts of Repository and Working Copy

SVN helps you manage a remote directory system where you can store files. That directory system is called a *repository*. The repository keeps a history of all the changes that are made to its files so that you can review how the files changed and retrieve previous versions of your files. Repositories are often shared among people as a way for all the people to get access to the same set of files.

SVN uses the notion of a working copy of the repository. So, when you put files into the SVN repository, you do not edit the files directly there. Instead, you retrieve a *working copy* of the files, modify your working copy, and then send (or *commit*) the files back to the SVN repository when you are satisfied with their state. So, you generally just commit files from your working copy to the SVN repository once the files are in some kind of stable (though possibly incomplete) state. For example, when using a repository to store programs, you generally do not commit files until they compile, at the least. In most cases, you will just commit the program files after they pass your unit tests.

We already saw in this and the previous labs how to checkout a repository, add files and directories, commit them to the repository, and do periodic updates from the repository.

SVN Review

The Faculty of Computer Science maintains an SVN server for use by faculty and students accessible by the host domain name `svn.cs.dal.ca`. The server already contains an SVN directory for each student in the class. You already used the command ‘`svn co`’ (or checkout) to retriev a copy of your SVN directory. You actually have not two copies, your primary and secondary working copy.

If you go to your primary working copy in the main working directory and list the files using the command ‘`ls -a`’ you should notice the directory called ‘`.svn`’. The option `-a` of the `ls` command lists hidden files. The directory ‘`.svn`’ contains the SVN state for the directory, including the location of your repository. Based on the information in the ‘`.svn`’ directory SVN knowns the exact address of your repository and some other information. You are not supposed to directly do any changes on the ‘`.svn`’ directory. Otherwise, you can “break” SVN and your working copy would not be in a consistent sate any more.

Step 3: Adding Files. First, we will prepare the file `hello.c`.

3-a) Make sure that you are in the primary window in the directory ‘`~/csci2132/svn/`*CSID*`/lab3`’. Remember that you can check your current directory with the pwd command, that ‘`~`’ is the label of your home directory, and *CSID* is your CSID.

Using emacs prepare the file `hello.c` with the following contents:

```
#include <stdio.h>

int main() {
  printf("hello world\n");
  return 0;
}
```

This is a simple C program. It has a lot of similarity with Java, but there are some differences. You can notice the line '`#include`' which is similar to the Java 'import' lines. There is no class, and we just have one function 'main', which is similar to the Java 'main' method. We use the function '`printf`' to print a string.

Save the program and compile it using the command:

```
  gcc hello.c
```

If there are errors, you need to check the file, and make sure that you copied the contents as specified. If there are no errors reported, then using the command '`ls`' you can verify that there is a new file called '`a.out`'. This is the executable program made from `hello.c`. You can run it using the command:

```
  ./a.out
```

and you can see the '`hello world`' output. Add the file '`hello.c`' to SVN and commit it.

3-b) Adding more files Copy the following program into a file called `sample.c`:

```
#include <stdio.h>

#define UNIX_ALL_OK 0

int main() {
  int status = UNIX_ALL_OK;
  int i;

  for (i = 0; i < 10; i++) {
    printf ("*");
  }
  printf ("\n");

  return status;
}
```

This is another small C program, which prints 10 asterisk characters. Create two more copies of this file named `sample2.c` and `tmp.c`:

```
cp sample.c sample2.c
cp sample.c tmp.c
```

We will now use the SVN `add` command to add two files into the repository:

```
svn add sample.c sample2.c
```

with output

```
A          sample.c
A          sample2.c
```

Notice that we can add any number of files at the same time to the repository. Are these files in the repository yet? No, since we have not issued a `commit` command to send them from your working copy to your repository.

We can compare the state of our working directory with the SVN repository with the SVN `status` command. In your primary window, enter

```
 svn status
```

Your output should look like this:

```
A       sample.c
A       sample2.c
?       tmp.c
```

The 'A' for the two sample files indicates that these files are waiting to be added to the repository. The '?' at `tmp.c` indicates that SVN knows nothing about the `tmp.c` file and will do nothing with the file.

Send the two files to the SVN repository with the SVN `commit` command:

```
svn commit -m "Adding the first version of my sample files."
```

If you re-issue the SVN `status` command, you should now get as output

```
?       tmp.c
```

SVN does not report on `sample.c` and `sample2.c` since the copy in your working directory is now identical to what is in the repository.

Step 4: Deleting Files. Suppose now that we did not mean to add `sample2.c` to the repository and want to remove it. We use the SVN `delete` command:

```
 svn delete sample2.c
```

that outputs

```
D       sample2.c
```

The leading "D" indicates that the file is now set for deletion. Re-issue the SVN status command to get the output

```
?       tmp.c
D       sample2.c
```

again, showing that SVN doesn't know about `tmp.c` and is ready to delete `sample2.c`. The change doesn't happen until you commit it to the repository (one line):

```
 svn commit -m "File sample2.c was added by mistake earlier.  It was just
       a backup copy of sample.c"
```

While `sample2.c` is no longer active in the SVN repository, we can still retrieve past copies of it later if we conclude that we should not have deleted the file. Notice that the `sample2.c` file is no longer in our working directory.

Step 5: Updating secondary copy. We will now start to work with two copies of your code. You might do this in real life if you are trying out two different solutions or if two different people were working on the same code.

In your **secondary** window, run the command:

```
  svn update
```

to bring the working copy up to date. Your `lab3` directory should now have the `sample.c` file in it, and also the '`hello.c`' file. Notice that there is no `sample2.c` file in this directory. We did the SVN `delete` command on the file, so we do not retrieve the file when we checkout fresh copies from the repository.

By default, SVN returns the most recent copy of the code that is in the repository.

Step 6: Changing files. Let us now experiment with changing a source code file. In your *primary* window, edit your copy of `sample.c`, so that the for-loop iterates 20 times instead of 10 times, and save it. Run the command:
`svn status`
You should get the following output:

```
M       sample.c
?       tmp.c
```

`M` indicates that the file has some modifications which are not submitted to the repository yet. Now, run:
`svn commit -mtest`
Now, go to the *secondary* window and run:
`svn update`
You can verify that the updated file is there.

Step 7: svn diff command. Go back again to your primary window and change the file `sample.c` to have the for-loop iterate 15 times now, and save the file. You can view the modifications with the SVN `diff` command:

```
 svn diff sample.c
```

The command will show you the lines where your version and the repository version of the file are different along with a few lines before and after the difference. Let us send this modification to the repository with the SVN `commit` command again

```
 svn commit -m "Use a 15-iteration loop"
```

You should get a message to show that the change was sent to the repository. Re-issue the SVN `diff` command for the `sample.c` file. Does it report any differences? Why?

In your *secondary* window, look at the contents of the `sample.c` file. It does not include the change that you just sent to the repository. This is because changes to the repository are not automatically sent to your *working* copy of the code so that this working copy remains stable for you.

To retrieve this newer copy, issue the SVN `update` command to update the working copy

```
 svn update
```

with output

```
U    sample.c
Updated to revision 16.
```

The leading ”U” means that there is a more recent copy of the code in the repository than what is in the working copy of the code. What happened to the number of iterations in `sample.c` in the working copy?

Step 8: Accidental Delete. What happens if we delete a file in a working copy? In the *secondary* window, delete the `sample.c` file:

```
 rm sample.c
```

Confirm that the file is gone with the Unix `ls` command. Executing the SVN `status` command will show you that the file `sample.c` is missing by the ! symbol in its output:

```
!    sample.c
```

Issue the SVN `update` command one more time. What happens to the contents of the working directory?

Step 9: Parallel Changes. Let us now introduce changes in our two copies of `sample.c`. In `sample.c` of your *primary* window, add a print statement before the for loop:

```
printf("I am about to start the loop\n");
```

In `sample.c` of your *secondary* window, add a print statement after the loop:

```
printf("I have finished the loop\n");
```

Now each copy of your file has a different change to `sample.c`. In each of the directories, use the SVN `status` command to confirm that the `sample.c` file is different from what is in the repository (look for the leading "M" tag beside the file names).

Step 9-a) Merging Changes. In your *primary* window, issue an SVN `commit` command to send the change to the repository. Do not forget to include a `-m "xx"` log message in the command. For example, you can use:

```
svn commit -mprimary
```

In your *secondary* window, try to commit the change to `sample.c`, for example, using command:

```
svn commit -msecondary
```

The command will fail since someone (you) committed a change to `sample.c` in the repository after the working copy in the secondary window was retrieved. To do the commit, you must first ensure that you have the most up-to-date version of the code from the repository. Use the SVN `update` command to get this copy:

```
 svn update
```

with output

```
G       sample.c
Updated to revision 17.
```

The "M" tag on `sample.c` from before has now changed to a "G". The "G" indicates that your changes and those from the repository have been automatically merged in the file. SVN was able to merge the changes since the changes in the primary and secondary windows did not overlap.

Edit the `sample.c` file to confirm that it now has **both** changes. We are satisfied with this change, so re-issue the SVN `commit` command in the secondary window to send the update to the repository. Again, do not forget to include a comment for the `commit` command, for example:

```
 svn commit -m"merged changes"
```

Bring both directories up to date. In both the *primary* and *secondary* windows, issue the SVN `update` command

```
 svn update
```

Step 10: Conflicting Changes. We will now see what happens if the separate changes conflict with one another. In your *primary* window, edit `sample.c`, change the for loop to iterate 30 times, and commit the change to SVN. In your *secondary* window, edit `sample.c`, change the for loop to iterate 5 times, and try to commit the change to SVN.

Like the last time, the commit in your *secondary* window failed because your secondary window did not have the most recent copy of the code. Again, like before, use the SVN `update` command to try and get the most recent version of code from the repository. Unlike last time, SVN is not able to merge the two changes since they are on the same line. Instead, the update returns with the following message:

```
Conflict discovered in 'sample.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options:
```

SVN is giving you the opportunity to fix see the differences (df) or fix the problem (edit the file). If you choose to edit the file, SVN will show you the file in your default editor. Since we have not defined that editor, let us select the postpone option. Enter `p` and let SVN continue with the update (if there were other files).
In the *secondary* directory, SVN has now left several files. In addition to your sample.c, it has the files sample.c.mine, and two sample.c files with an ending of .rX where the "X" is the version number of each file.

– sample.c — the sample.c file with the "diff" information embedded for you to resolve
– sample.c.mine — your modified code
– sample.c.r9 — the original copy of the code that you modified in this working directory (revision 9)
– sample.c.r11 — the most recent version of the code in the repository (revision 11)

At this point, you can either change `sample.c` to merge both changes (still having the other versions around as a reference) or copy one of the other versions to overwrite sample.c with what you want the final copy to be.

Examining Conflict. In the *secondary* window, re-issue the SVN `status` command. Rather than seeing that `sample.c` is modified, you will now get the output

```
C       sample.c
```

where the leading "C" indicates that `sample.c` has a conflict that you need to resolve before it is sent to the repository.

Before fixing the conflict, we should find out what the changes were. A first step is often to look at the comments from the commit to find out why someone else changed the repository. Use the SVN `log` command in the *secondary* window to get the log of messages for a file:

```
 svn log sample.c
```

You will get output to show each of the versions of the file along with the commit messages. If the message on the last `commit` for `sample.c` is not giving you enough information, you can use the SVN `diff` command again to see where your file differs from what is in the repository:

```
 svn diff sample.c
```

A part of the output is something like:

```
+<<<<<<< .mine
+  for (i = 0; i < 5; i++) {
+=======
+  for (i = 0; i < 30; i++) {
+>>>>>>> .r39
```

Resolving Conflict. Given this information, choose a fix to `sample.c` (for the lab, it does not matter what the "fix" is). Usually, you can either:

- Edit `sample.c` directly (delete extra lines, etc.)
- Choose our version: `cp sample.c.mine sample.c`
- Choose repository version: `cp sample.c.r39 sample.c`
- Choose original last updated version: `cp sample.c.r38 sample.c`

In this case, we can simply choose:
`cp sample.c.mine sample.c`

If you try to commit the file, SVN will still refuse since it does not know yet that you have resolved the conflict. To do so, issue SVN's `resolved` command on the file:

```
svn resolved sample.c
```

After issuing this command, the `sample.c.mine`, and `sample.c.rX` files will be removed from your working directory. You can now issue the SVN `commit` command to send the changes to the repository.

Step 11: Retrieving copies of previous work. You sometimes make changes to a program that you later regret and want to undo. Many of you probably make a copy of a file once it is working and this "undo" step means copying that old file to overwrite your current file. We want to use a repository to automate that process better.

Each time you reach a stable point in your development, it is worthwhile to do an SVN `commit` to put that stable point into your repository. Aside: If you are making changes in parallel as someone else is, it is useful to make frequent updates and commits.

In your *secondary* window, edit `sample.c` and introduce a compile error (like changing the reserved word `printf` to `print`). Save the file to disk (but not to the repository).

Suppose that you now want to undo that change and return to the last "saved" copy in the repository. Three options that you already know how to do are:

- use SVN `checkout` elsewhere to get a fresh copy of the repository and copy the sample.c file from there
- delete `sample.c` in the working directory and then use the SVN `update` command to have SVN notice that there is a new version of `sample.c` in the repository and make a copy in your working directory
- use SVN `diff` to identify the changes that you made and manually change `sample.c` to return it to its previous state

Each of these involves a bit more work than is necessary. If you want the last version of the file, SVN has a command called `revert` to retrieve that copy:

```
svn revert sample.c
```

After issuing the command, look at sample.c again to notice that you now have the version of the file that precedes your latest editing.

Retrieving Older Versions. It is possible that you might want to retrieve a much older version of your file. You can use the `-r` option the `update` command. The `-r` lets you specify which revision of the file that you want. Try something like

```
svn update -r 34 sample.c
```

but do not use the number 34. Instead, you can use '`svn log sample.c`' to find "interesting" revision numbers. What is the content of this sample.c file?

If you changed this earlier version of the file, SVN will not let you commit the change directly to the repository since it is based on an out-of-date version of the file. You would need to do an SVN `update` command to return to latest copy (inducing a conflict with the file) where you then need to resolve any conflicts with the latest version.

Retrieving by Timestamps. Sometimes you don't know the exact revision number that you want to retrieve, but you know when the directory was last stable. You can specify a time, rather than a revision number, in the SVN `update` command.

Suppose that I knew that the file was stable on October 20th, 2013 at 1pm. I could issue the following command to retrieve the version of the file that existed in the repository at that time:

```
svn update -r "{2013-01-08 13:00:00}"
```

Step 12: End of lab. By now, you have finished the required work of this lab. You can work on your assignment, or some of the practice programming questions.