CSCI 2132: Software Development

Lab 3: Version Control with Subversion (SVN)

Synopsis

In this lab, you will:

- Learn about the Subversion (SVN) version control (VC) system
- Checkout projects from SVN
- Add a subproject to SVN
- Add and delete files to and from 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 made to your code
- Resolve conflicts between your changes and other people's changes

Contents

Overview

In this lab, you will familiarize yourself in more detail with the Subversion (SVN) version control system. While many people interact with SVN using a GUI interface (often as part of their IDE), it is useful to know how to interact with SVN using the command line. In particular, the command line interface to version control systems such as Subversion or Git provides a reference of the full feature set of the system. GUI front ends often offer only a (small) subset of these features. Knowing which features are available allows you to evaluate whether a given GUI front end meets your needs and being able to mentally map various GUI commands to the corresponding command line operations tends to increase your confidence in using a GUI system because you understand exactly what each operation does.

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

All software companies use some form of version control, so becoming familiar with the basic concepts of version control is useful. You will not be a master of version control after this lab, but you will have an understanding of the basic concepts and commands.

Subversion is relatively popular. Git is its strongest competitor and has a different philosophy of how different users share their work with each other. While a core feature of SVN is a linear history of revisions and interaction via a single central repository, Git allows highly non-linear revision histories and is a *distributed* version control system often without any centralized repository; instead, every developer has their own Git repository and developers share code with each other by asking other developers to "pull" their changes into their repository. We will discuss Git later in this course.

Subversion has grown out of a sequence of predecessors starting with SCCS, followed by RCS, and then CVS. (I am old enough to have used all of them :)).

Step 1: Log in

You should be used to this first step by now. Log into `bluenose` using `PuTTY` or `ssh`. Change your working directory to `~/csci2132/svn/CSID` and run `svn update`. Recall from Lab 2 that this ensures that your working copy is an up-to-date reflection of the repository contents.

`svn update` will print information about any updates that are applied to your working copy. You may want to inspect the changed files. This is a good idea in general if some of the changed files are relevant to your work as part of a software development team. In the context of this course, changes will usually be feedback from TAs that is valuable for you to review.

Now create a `lab3` directory and add it to SVN:

```
$ svn mkdir lab3
$ svn commit -m"Directory lab3 created"
$ cd lab3
```

You should understand by now what these commands do, even though we have used a slightly different workflow. Instead of creating a directory outside SVN using `mkdir` and then adding it to SVN using `svn add`, SVN offers a single command that combines these two steps: `svn mkdir`.

With the `lab3` directory prepared, we are ready to proceed to the main part of this lab. In this lab, you will use two windows to work with the SVN repository. We will call the window you just used to log in the **primary window** and the working copy under `~/csci2132/svn` the **primary working copy**. In the next step, you will create a second working copy to simulate two different users interacting with the SVN repository.

Step 2: Create a second working copy

Open another window, that is, start a second PuTTY session or open a new terminal window on macOS or Linux and log into `bluenose` using `ssh` a second time. We will refer to this window as your **secondary window**. You will use this window to interact with you **secondary working copy**, which you create in this step.

Change your working directory in the secondary window to `~/csci2132`. Create a subdirectory called `tmp` (a common name for a directory holding temporary files on Unix; there exists a system-wide `/tmp` directory, for example). Switch into this directory using `cd tmp`. `pwd` should display `/users/cs/CSID/csci2132/tmp` in your secondary window now.

Now check out your `lab3` directory in the secondary window:

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

After this command, the `tmp` directory contains only your `lab3` directory, not the `lab1` and `lab2` directories that you also submitted to SVN. This is one of the very few advantages that SVN has over Git; Git does not allow you to check out only part of a repository.

A few notes and reminders:

- Throughout this lab, we will continue to refer to your CSID as `CSID`. Remember to replace this with your actual CSID for every single command in this lab.
- **Important security notice:** Each time you enter an SVN command, you will be asked for your `bluenose` password. Enter it. **When asked whether you want to store the password, answer "no" each time.** While it does require that you frequently enter your password, the password is currently not stored safely, so we do not want you to store it. Tech support is working on resolving this issue.

Repository and working copy

An *SVN repository* is a remote directory system that you manipulate using the various `svn` commands. In many instances, this is indeed a *remote* directory system in that it is hosted on a different machine than the one you are working on, `svn.cs.dal.ca` in the case of the repository you work with in this course. The repository may also be in a different location on the same computer as the one you are working on. The important thing to remember is that it is different from your working copy and that you cannot meaningfully interact with it in any other way than using the `svn` commands. (The directories and files are stored in the repository in a Subversion-specific database format.) The SVN repository also does not only store the current version of each file but stores the full history of all changes made to directories and files in the repository.

In order to make changes to the files and directories in the SVN repository, you need to check out a working copy, modify this copy, and then send (commit) your changes back to the SVN repository. You generally commit files from your working copy to the SVN repository once they are in a stable (though possibly incomplete) state. For example, when using a repository to store programs, you generally do not commit files until they compile or pass a set of required tests.

We already saw in this and the previous labs how to check out a repository, add files and directories, commit them to the repository, and perform periodic updates of the working copy from the repository.

The innards of your working copy

If you navigate to your primary working copy at `~/csci2132/svn/CSID` and enter `ls -a`, you should notice a directory called `.svn`. The option `-a` makes `ls` display hidden files and directories, which are defined to be all files or directories whose names start with a period. These files are normally not displayed by `ls`, but the `-a` option ensures they are displayed. Do not directly modify the contents of this `.svn` directory or delete it. This is where SVN stores all the administrative information about your working copy such as the latest checked-out revision and snapshots of all files in this revision.

Step 3: Add a file

In this step and the next, you will practice adding new files to SVN. First make sure you are in the primary window in the directory `~/csci2132/svn/CSID/lab3`. Remember that you can check your current directory using `pwd`.

Use `emacs` to create the following file:

hello.c

```
#include <stdio.h>

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

This is a simple C program and I suspect you can guess what it does. There are some differences to the Java version you prepared in an earlier lab. The `#include` statement is similar to a Java `import` statement, even though it technically works a lot differently under the hood. C does not have classes, so the `main` function is not wrapped in a class definition. The `main` function in C has to have an integer return type, which is the exit code it returns to the shell. Finally, `printf` is C's most commonly used function to print strings to `stdout`.

Save the program and compile it:

```
$ gcc -o hello hello.c
```

If there are errors, you need to check the file and make sure that its contents are exactly as specified above. If there are no errors reported, you should run `ls` and see a file `hello` in your current directory. This is an executable produced by the C compiler `gcc` from your source code. Run it:

```
$ ./hello
Hello, world!
```

Add the file `hello.c` (but not the executable) to SVN and commit it.

Step 4: Add more files

Now create another file:

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 C program, which prints 10 asterisks. Create two more copies of this file named `sample2.c` and `tmp.c`:

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

Add the two `sample*.c` files to SVN:

```
$ svn add sample.c sample2.c
A           sample.c
A           sample2.c
```

Notice that you can add any number of files using a single `svn add` command. Are these files in the repository yet? No. We still need to use `svn commit` to commit our changes to the repository.

You can check the state of your working copy using

```
$ svn status
A           sample.c
A           sample2.c
?           tmp.c
```

The `A` indicates that the two files have been added but not committed yet. The `?` indicates that SVN knows nothing about this file, so `svn commit` will do nothing with this file.

Send the two files to the SVN repository:

```
$ svn commit -m"Adding the first version of my sample files"
$ svn status
?           tmp.c
```

`svn status` does not report on the `sample*.c` files because they have been successfully committed and are therefore identical to their state in the repository. To see all files, even the ones consistent with the repository content, use `svn status -v`.

The default behaviour of `svn`, without the `-v` option, is designed so that you get no output if there are no "issues" you may need to be aware of, that is, if all files below the current directory are under SVN control and match the version in the repository. Any differences between your working copy and the repository version (added files, deleted files, modified files, unknown files such as `tmp.c` above) are listed so you can take appropriate action to incorporate these files or the changes you made to them into the repository.

Step 5: Delete files

Suppose now that we did not mean to add `sample2.c` to the repository. We can undo the addition:

```
$ svn delete sample2.c
D           sample2.c
```

The leading `D` indicates that the file is now set for deletion but has not been deleted from the repository yet. Check the status of your working copy again:

```
$ svn status
D           sample2.c
?           tmp.c
```

Now commit your changes and check the status again:

```
$ svn commit -m "File sample2.c was added by mistake. Deleting it."
$ svn status
?           tmp.c
```

All changes have been committed successfully but SVN still does not know anything about `tmp.c`. If you run `ls`, you will notice that the file `sample2.c` has been removed from your working copy. (This happened already when you ran `svn delete sample2.c`, not only when your committed your changes.)

Step 6: Get changes made by others

The changes you just made in your primary working copy and committed to the repository are not present in your secondary working copy yet; the secondary working copy is still in the state of the repository before these changes. You can pull these changes into your secondary working copy by running

```
$ svn update
```

in your secondary window. The `lab3` directory in your secondary working copy should now contain the files `hello.c` and `sample.c`. Note that there is no `sample2.c` file because we removed it from the most recent revision of the repository; by default, `svn update` updates your working copy to the most recent revision in the repository.

Step 7: Change some 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

```
$ svn status
M           sample.c
?           tmp.c
```

`M` indicates that the file has some modifications that have not been committed to the repository yet. Run

```
$ svn commit -mtest
```

in your **primary window** and

```
$ svn update
```

in your **secondary window**. You can see that the changes in `sample.c` have been copied from your primary working copy to your secondary working copy by running

```
$ cat sample.c
#include <stdio.h>

#define UNIX_ALL_OK 0

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

  for (i = 0; i < 20; ++i) {
    printf("*");
  }
  printf("\n");

  return status;
}
```

Step 8: `svn diff`

Switch back to your **primary window**, change the file `sample.c` to have the `for`-loop run for 15 iterations, and save the file. You can view the changes you made using

```
$ svn diff sample.c
Index: sample.c
===================================================================
--- sample.c (revision 8)
+++ sample.c (working copy)
@@ -6,7 +6,7 @@
   int status = UNIX_ALL_OK;
   int i;

-  for (i = 0; i < 20; ++i) {
+  for (i = 0; i < 15; ++i) {
     printf("*");
   }
   printf("\n");
```

This correctly tells us that, compared to the current revision (revision 4), the working copy contains one change where the line

```
-    for (i = 0; i < 20; ++i) {
```

has been replaced with the line

```
+    for (i = 0; i < 15; ++i) {
```

The output is the same as would be produced using the command line tool `diff` for comparing two files (if you run it with the `-u` command line option). The two header lines state which files are being compared. The line enclosed in `@@` indicates the locations in the source files where the changes are located (in this case, line 6 in both files). By default, 3 lines of context before and after each change are displayed because changes are easier to interpret if we see a bit of the code around them. Finally, every deletion is marked with a `-` and every addition is marked with a `+`. Since `diff` and `svn diff` work on a line-by-line basis, the only concept they understand is the replacement of one group of lines with another group of lines, that is, they treat all edits as combinations of line deletions and line additions. Here, you see that `sample.c` in the working copy can be obtained from `sample.c` in revision 8 by deleting line 6 and replacing it with the line marked with a `+`.

Commit this modification using

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

You should get a message that indicates that the changes were committeed to the repository. Re-issue `svn diff` for the `sample.c` file. Does it report any differences? Why?

`svn diff` is a tremendously useful command. Git's counterpart, `git diff`, has become an integral part of my workflow; almost before every commit, I use `git diff` to review the changes I am about to commit to the repository to make sure they do not include any obvious mistake. You should also try to develop this habit, along with making it second nature to use version control to manage pretty much *any* non-trivial project you work on, whether it is a piece of software, a term paper or anything else you work on. It gives you the ability to recover from mistakes, experiment with confidence and then discard your changes if the experiment didn't work out or easily collaborate with others through a common repository.

Step 9: Retrieve the latest changes

In your **secondary window**, look at the contents of the `sample.c` file. It does not include the changes that you just committed to the repository because committing changes to the repository does not automatically send updates to all working copies of this repository. This is because it is generally technically impossible; it would require tracking all working copies that exist, which cannot be done in a robust manner and it would require some push mechanism that then effectively allows anybody who checks out this repository to mess with your files—not a good idea. It would also be terribly disruptive: imagine that you are working on implementing a new feature by editing a given file and, while you are working on this file, it gets changed as a result of somebody else's commit.

To retrieve all updates committed to the repository, run

```
$ svn update
Updating '.':
U    sample.c
Updated to revision 9.
```

The leading `U` means that there was a more recent copy of this file in the repository and that the file was updated to this version. What happened to the number of iterations in `sample.c` in your secondary working copy?

Step 10: Accidental delete

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

```
$ rm sample.c
```

Use `ls` to confirm that the file is indeed gone. Using `svn status`, we obtain the following output:

```
$ svn status
!       sample.c
```

The `!` indicates that the file is in the repository but not in the working copy. SVN assumes that the file was deleted by accident (because you didn't inform SVN about your intention to delete the file by running `svn delete` instead) and informs you of this grave (`!`) mistake. You can restore the file using `svn update`:

```
$ svn update
Updating '.':
Restored 'sample.c'
At revision 9.
$ svn status
```

Step 11: Merge parallel changes

Next we discuss the main issue that distinguishes a good version control system from a bad one: if different team members make changes to the same file in their respective working copies, how do we combine these changes made in parallel into one consistent version of the file in the repository?

In your **primary window**, add a print statement before the `for`-loop:

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

In your **secondary window**, add a print statement *after* the `for`-loop:

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

Now the two copies of `sample.c` in the two working copies have different changes in them. In both windows, use `svn status` to verify that the files are indeed different from the repository version. (Look for the leading `M` before `sample.c` in the outputs of `svn status` in both windows.

Now we merge the two changes into a single repository version. Commit the changes made in your **primary window**:

```
$ svn commit -mprimary
```

Try to do the same in your secondary window:

```
$ svn commit -msecondary
```

This fails because someone (you, in your primary window) made a change to the repository version of `sample.c` that has not been integrated into your secondary working copy yet. Before committing your changes in the secondary working copy, you need to *merge* the repository changes into your working copy. The merge is performed, once again, using `svn update`:

```
$ svn update
G    sample.c
Updated to revision 10.
```

The `G` tag indicates that changes from the repository have been merGed successfully into your working copy of `sample.c`. You can check this by viewing the file `sample.c`, which should now have *both* print statements, before and after the loop, in it.

Now you can commit your changes to the repository:

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

Ensure that both working copies are in the same state by running

```
$ svn update
```

in your **primary window**.

Step 12: When changes conflict

The previous merge of parallel changes was unproblematic because they affected different parts of the same file. When two users make changes to the same region of a file, `svn update` cannot determine how to meaningfully combine these changes. If two users each add a line between the same two lines in the repository version of a file, how should these two additions be treated? Which of the two lines should be first and which should be second? Did both users happen to make the same change and only one of the two additions should be kept? Which version of two parallel edits *of the same line* should be kept? There is no way SVN can guess our intent, so it needs our help to figure out how to deal with such conflicts.

To learn how to do this, let us create two conflicting edits. First open `sample.c` in your **primary window** and change the loop to run for 30 iterations. Then open it in your **secondary window** and change the loop to run for 5 iterations. Clearly, these two changes are in conflict and SVN cannot guess how to resolve this conflict.

Again, commit your change in the **primary window** and then run `svn update` in your **secondary window**. (You already know that trying to commit your secondary working copy will fail without running `svn update` first.) This time, the update does not run as smoothly:

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

You already know what this conflict means. `svn update` gives you a number of choices how you want to resolve the conflict. You can get descriptions of the different available options by choosing `s`. Choose `p` here, which gives you the most flexibility to deal with conflicts. This leaves 4 files in your working copy:

`sample.c:`	the `sample.c` file with the "diff" information embedded for you to resolve
`sample.c.mine:`	the version of `sample.c` in your working copy before running `svn update`
`sample.c.r11:`	the version (revision 11) of `sample.c` that your working copy version was based on. (The last revision you checked out before editing was revision 11.)
`sample.c.r12:`	the version (revision 12) you just tried to check out and which contains conflicts with your working copy.

Step 13: Examine the conflict

If you run `svn status` in your secondary window now, you will see the following output:

```
$ svn status
C       sample.c
```

The leading `C` indicates that `sample.c` has a conflict that needs to be resolved to produce a merged version that can be committed to the repository.

Before fixing the conflict you should find out what the changes are. A first step is to look at the commit messages to find out why someone else changed the repository. Use

```
$ svn log sample.c
```

to inspect all log messages that pertain to the `sample.c` file.

If the message on the last commit for `sample.c` does not give you enough information, you can use

```
$ svn diff sample.c
```

to see where your working copy version of `sample.c` differs from the one in the repository.

Part of the output looks something like this:

```
+<<<<<<< .mine
+  for (i = 0; i < 5; ++i) {
+=======
   for (i = 0; i < 30; ++i) {
+>>>>>>> .r12
```

Step 14: Resolve the conflict

You now have multiple options to resolve the conflict. First, you could replace `sample.c` with one of the files `sample.c.mine` or `sample.c.r11` to choose one of these versions as the "merged" version. The "mine-full" (`mf`) or "theirs-full" (`tf`) options shown when you enter `s` in response to the prompt presented by `svn update` produce the same result.

Here, you take the most fine-grained route of inspecting the changes one by one and choosing how to reconcile them by editing `sample.c`. The file `sample.c` should look something like this at this point:

sample.c

```
#include <stdio.h>

#define UNIX_ALL_OK 0

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

<<<<<<< .mine
    for (i = 0; i < 5; ++i) {
=======
    for (i = 0; i < 30; ++i) {
>>>>>>> .r12
        printf("*");
    }
    printf("\n");

    return status;
}
```

Every conflict is enclosed in a <<<<<<<, >>>>>>> pair. The region is further subdivided using a marker =======.[1] This gives you two subregions of the conflict region. The first subregion is the version of this region found in your working copy before the merge. The second subregion is the version of this region you just unsuccessfully tried to merge into your working copy. You should edit this region to pick a version that reflects the state you expect the file to be in. You can make additional changes and add or remove lines in the process. Do not forget to remove the various markers (<<<<<<<, >>>>>>>, and =======) from the file because otherwise, they become part of the source code.

[1] On newer versions of SVN, there is a third marker ||||||| that creates a third subregion that shows the repository version the working copy is based on. Since this is a predecessor version of both the working copy and the most recent repository version, this can be useful for resolving conflicts.

Step 15: Commit the merged version

After resolving the conflict, you have to commit the merged version into the repository. If you try to commit the file using `svn commit`, you will still get an error message. This is a safeguard SVN provides to prevent you from accidentally committing `sample.c` with all the diff markers in it. You need to indicate that you are done making changes and are ready to commit your changes by invoking

```
$ svn resolved sample.c
```

This indicates that the conflicts in `sample.c` have been resolved and also removes all the temporary `sample.c.mine` and `sample.c.r*` files created by the `postpone` option of `svn update`.

Now you are ready to commit your changes using

```
$ svn commit -m"Conflicts resolved"
```

Note: *I left the documentation of this step unchanged from previous versions of this lab. As I worked through this, however, I did not get an error message when trying to run* `svn commit`. *Everything worked without errors, probably because SVN realized that my file had a newer modification time than the conflicting version. This may be a change in behaviour introduced by an upgrade of SVN on* `bluenose`.

Step 16: Restore the repository version of a file

One of the main reasons for using version control is to be able to inspect who made which changes and, according to the log messages, for what reason. The other reason is to be able to go back to earlier versions of a file in case some changes made after a certain point prove to be problematic (e.g., introduce a bug).

In order to have this ability to undo changes, it is important that you commit your changes every time your code is in a stable state. Compared to Git, merging changes and resolving conflicts is SVN's greatest weakness. Thus, to minimize the pain, it is also advisable to frequently commit changes and update your working copy to the repository version if you know others are working on the same file as you are.

The simplest scenario is when you want to undo changes you haven't committed yet and simply want to discard all changes you made since the last `svn update`.

```
$ svn revert sample.c
```

does exactly this. It discards all changes you have not committed yet.

Experiment with this by making some changes to `sample.c`, use `svn diff` to verify that your working copy version is indeed different from the version in the repository. Then run `svn revert` to undo the changes, check the status using `svn status` and inspect the file contents using `cat`.

Step 17: Retrieve a repository version before the most recent one

The slighly more complicated scenario is when you need to discard a whole series of commits. In this case, you will normally identify the revision that has the most recent correct copy of the file in question in it. You can do this by inspecting log messages using `svn log` or checking out various earlier versions until you find the one you are looking for. To check out an earlier version, use

```
$ svn update -r XXX
```

where `XXX` is the revision you want to check out. This checks out the specified version as your working copy.

If you now edit files in this older revision, SVN does not allow you to commit the changes. In keeping with SVN's notion of a linear history, only working copies based on the most recent revision in the repository can be committed. To commit the changes you just made, you need to run

```
$ svn update
```

which will likely cause some conflicts. You can now choose various conflict resolution strategies, possibly just choosing your current working copy version over the repository version since your goal was to discard the changes made by recent revisions.

After resolving the conflicts as appropriate, you can once again commit your changes using `svn commit` (and possibly `svn resolved`).

As an example, let us keep the current version of `sample.c` but let us remove the two `printf` statements before and after the loop that we introduced earlier. Find the last revision that did not have these statements in it. Run `svn update -r XXX` with different revision numbers `XXX` and inspect the checked-out version of `sample.c` until you find the most recent revision that does not have either of the two `printf` statements in it. For me, this was revision number 9.

In addition to not including the `printf` statements, this revision had 15 iterations of the `for`-loop. You still want only 5 iterations of the loop as in the most recent revision. So change this back to 5 iterations.

In order to commit these changes, you need to base them on the most recent revision in the repository:

```
$ svn update
```

As the conflict resolution option, choose "mine-full" (`mf`) because you want the version of the file you just created to be the new version of the file. Finally, commit your changes:

```
$ svn commit -m"Removed printf statements"
```

Step 18: Retrieve a revision by timestamp

If you know the date of the version to which you want to reset your working copy, you can also check out a version based on timestamp rather than revision number:

```
$ svn update -r "{2018-12-31 00:00:01}"
```

In other words, you use the same command line flag `-r` to specify the revision to check out, but instead of providing a number, you give a timestamp enclosed in curly braces.

As an example, use `svn log` to find the revision that removed `sample2.c`. Use a timestamp just before this revision to restore this revision. As I am working through this, I get:

```
$ svn log
...
------------------------------------------------------------------------
r7 | nzeh | 2018-12-30 18:04:08 -0400 (Sun, 30 Dec 2018) | 1 line

File sample2.c was added by mistake. Deleting it.
------------------------------------------------------------------------
...
```

So, by checking out the revision that was active at time 18:04 on December 30, 2018, I get the version with the sample2.c file present:

```
$ svn update -r "{2018-12-30 18:04}"
$ ls
hello.c  sample  sample2.c  sample.c
```

The important part is that the sample2.c file has been restored to my working copy because it was present before revision 7.