CSCI 2132
Software Development

Lab 3:

SVN Source Version Control Tutorial

Instructor: Vlado Keselj

Faculty of Computer Science

Dalhousie University

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

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

Step 1: Logging in

1-a) Login

1-b) Change directory

1-c) SVN Update

1-d) Prepare the `lab3` directory

(the primary copy)

Step 2: Secondary Working Copy

2-a) Login to bluenose in another window

2-b) mkdir and cd to `~/csci2132/tmp`

2-c) Check out only your `lab3` directory

A Couple Notes (Reminders)

- Replace *CSID* with your CS user id (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, since the password is not stored safely currently.
- If you are sure that you know what you are doing, you may temporarily enter 'yes' to make SVN use easier, but then make sure that only you have access permissions to the the directory `~/.subversion`, and it is a good idea to delete this directory after each session.

Repository and Working Copy

- SVN uses remote directory system called *repository* to store files and history of their changes
- You can review changes and retrieve older versions
- Files can be shared among users and SVN can be used for collaborative work
- SVN uses notion of *working copy* of the files
- We send (or *commit*) files to the repository when we are happy with their state
- For example, we generally do not commit program unless it can be compiled

- Review commands that you know in the notes

- `checkout` or `co`
- `add`
- `commit -m`*log_message*

Step 3: Adding Files.

3-a) Prepare file `hello.c` compile and run

- Submit `hello.c` to SVN

3-b) Adding more files

- Make two copies:

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

- Use SVN `add command`

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

- Run command: `svn status`
- Output:

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

- Run commit command (one line):

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

- Run again: `svn status`

Step 4: Deleting Files

- Delete a file: `svn delete sample2.c`
- Output:

```
D           sample2.c
```

- Re-issue the SVN status command, output:

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

- Commit the change to repository:

```
svn commit -m "File sample2.c ..."
```

Step 4: Deleting Files (continued)

- Enter: `svn status`
- Output:

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

- Enter in one line:

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

- The file is deleted, but it can be retrieved from the repository

Step 5: Updating secondary copy

- In the **secondary** window get the working copy up to date:

  ```
  svn update
  ```

- List the contents: `ls -la`

Step 6: Changing Files

- In your primary window edit `sample.c`:
  - Change for-loop to iterate 20 times instead of 10
- Run: `svn status`
- You should get the following output:

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

- Run: `svn commit -mtest`
- In the secondary window run: `svn update`
- Notice that a new file is there

Step 7: svn diff command

- In primary window: change `sample.c` for the for-loop to iterate 15 times, and save.
- Run: `svn diff sample.c`
- Output will show differences between working version and repository version
- Commit the change:

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

- Try again `svn diff sample.c`
- No output. Why?

Step 7(continued)

- In the *secondary* window look at `sample.c`
- Run: `svn update`
- Output:

```
Updating '.':
U    sample.c
Updated to revision 36.
```

- `U` means that there is a more recent copy in the repository
- Check again file sample.c

Step 8: Accidental Delete

- Let us try this in the secondary window
- Let us remove a file: `rm sample.c`
- Check that it does not exist: `ls`
- Run: `svn status`
- Run the update command

Step 9: Parallel Changes.

- Let us change `sample.c` in both windows:
- In *primary* window add a print statement before the loop:

```
print "I am about to start the loop\n";
```

- In *secondary* window add a print statement after the loop:

```
print "I have finished the loop\n";
```

- In each windows use `svn status` to verify that the working file is different from the repository copy (indicator `M`)

Step 9-a) Merging Changes

- In *primary* window commit changes, e.g.:
  `svn commit -mprimary`
- Try to commit changes in *secondary* window, e.g.:
  `svn commit -msecondary`
- Commit fails since the repository copy changed since our last update
- Run: `svn update`
- Output:

```
G    sample.c
Updated to revision 37.
```

- `G` indicates that changes have been merged

Step 9-a) Merging Changes (cont.)

- (Still in *secondary* window)
- Edit file `sample.c` to make sure you are happy with merged changes
- Run: `svn commit -m"merged changes"`
- This will submit merged changes to repository
- In both windows run: `svn update`
- Now both windows contain the latest copy in repository

Step 10: Conflicting Changes

- In primary window, edit `sample.c` and set loop to iterate 30 times
- Commit changes
- In secondary window, edit `sample.c` and set loop to iterate 5 times
- Try to commit changes (commit will fail as before)
- Run: `svn update`
- However, changes are not merged this time
- We get the following message . . .

- We get 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:
```

- `df`—see differences, or different options of editing or fixing the file
- Let us choose to postpone, enter: `p`
- We can see now several files in the *secondary* window by running: `ls`
- The files are (likely with different version numbers):
  - sample.c — the sample.c file with the "diff" information embedded
  - sample.c.mine — your modified code
  - sample.c.r38 — the original copy of the code that you modified in this working directory (revision 38)
  - sample.c.r39 — the most recent version of the code in the repository (revision 39)

Examining Conflict

- (Still in *secondary* window) Run status command
- Output will include line:
  ```
  C       sample.c
  ```
- `C` indicates conflict that needs to be resolved
- Examine the log using log command: `svn log sample.c`
- The output shows log messages (which demonstrate why it is useful to write descriptive log messages)
- Additional information can be obtained using diff command:
  ```
  svn diff sample.c
  ```
- A part of the output is:
  ```
  +<<<<<<< .mine
  +  for (i = 0; i < 5; i++) {
  +=======
  +  for (i = 0; i < 30; i++) {
  +>>>>>>> .r39
  ```

Resolving Conflict

- Fix `sample.c` file. 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`
- We can simply choose: `cp sample.c.mine sample.c`
- To indicate that the conflict has been resolved, and sample.c is the version that we want submitted to the repository, we issue svn resolved command:

  ```
  svn resolved sample.c
  ```

- Other files (sample.c.mine and sample.c.rX) are removed
- Finally, we commit the change:
  `svn commit -m"conflict resolved"`

Step 11: Retrieving copies of previous work

- Sometimes we make changes to a program and want to undo them
- It is useful to make commit each time we reach a stable point
- Aside: If collaborating with someone, it is useful to do frequent updates and commits
- In *secondary* window: Introduce error in file sample.c; e.g., change word `printf` to `print`
- Save the file but do not commit
- Suppose that we want to undo the change to the last "saved" copy in the repository. We already know three options:
  - make checkout elsewhere and get the fresh copy from there
  - delete sample.c and run update
  - use diff to identify changes and manually reverse them
- All these involve unnecessary work; much easier:

```
svn revert sample.c
```

Retrieving Older Versions

- How to retrieve older versions of a file?
- We can use `-r` option with the update command:

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

- Do not use number 34, instead we can use log command (`svn log sample.c`) to find "interesting" revision numbers
- Check the contents of sample.c
- If you change this file and try to commit, it will not work as a newer version is in the repository. You would need to make an update, and likely resolve any conflicts.
- Instead, simply run: `svn update`
- This will get the latest version from the repository

Retrieving by Timestamps

- Sometimes we do not know exact revision number, but we know time when the directory was last stable.
- We can specify time to retrieve the latest version that existed at this time
- For example:

```
svn update -r "{2013-10-20 13:00:00}"
```

Step 12: End of lab