**Faculty of Computer Science, Dalhousie University** *14-Sep-2018*

**CSCI 2132 — Software Development**

# Lecture 5: File Permissions

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

**Previous Lecture**

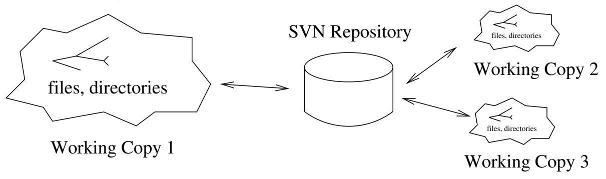
- **Files and Directories**
- Pathnames
- Commands for managing and navigating directory structure
- Commands: cat, logout, exit, ls, dirname, basename, pwd, cd, mkdir, rmdir, mv, rm, tree
- File manipulation commands
- File permissions:
  - users, groups
  - checking permissions

## Note About SVN

*Slide notes:*

**A Note About SVN**

- SVN (Subversion) — Software Versioning and Revision Control System
- A simplified view:
  - Backups — creating backups in a repository
  - Historical — “time machine”, labeled versions
  - Collaborative — different users can contribute and merge changes



SVN is a Software Versioning and Revision Control System, or express more briefly, a tool for software version control. If you are not familiar with such tools, a simple way to describe it is by presenting its three main general functionalities:

1. backup creation,
2. preserving historical backups, and
3. collaborative functionality.

**Backup Creation:** The SVN tool can simply be used to preserve a backup of our work that can be accessed from different computers. This is a functionality similar to how people use DropBox, Google Drive, of Microsoft One Drive these days.

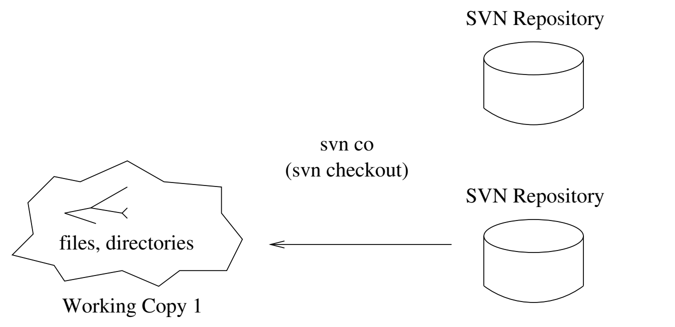
**Historical Backups:** The backups are kept historically, which means that new backups do not delete old ones, but we can retrieve the last backup from any moment in time in the past. SVN in this way behaves like a “time machine”. If at a significant moment we release a software version, for example version 1.17, we can label that backup copy and later if we need this exact version, we can retrieve it. The whole directory structure and all files will be restored as in time when the verion 1.17 was saved in the SVN repository. SVN is smart about saving only differences so disk space is not wasted by keeping a lot of copies of the same file if the file does not change from version to version. Even for files that do change, only differences are saved.

**Collaborative Functionality:** The third and a more advanced functionality is the collaborative functionality. Several people can make their own copy of the software and they can work independently on it. SVN provides a way to merge their work into the project. SVN will try, and it usually succeeds, in merging any changes done independently, even when done on the same file. The users will sometimes make edits that are hard to merge, such as edits made on the same line, and it that case SVN will generate something called a “conflict”, and the users will need to resolve this manually.

*Slide notes:*

**SVN Checkout (‘svn co’ or ‘svn checkout’)**

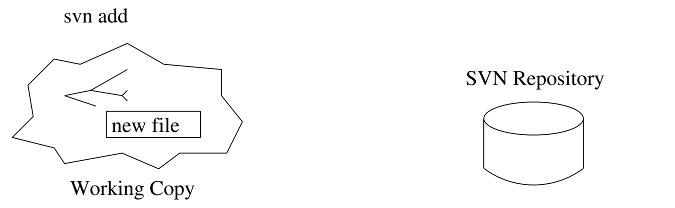
- SVN checkout command is used to create an initial working copy



*Slide notes:*

**‘svn add’ command**

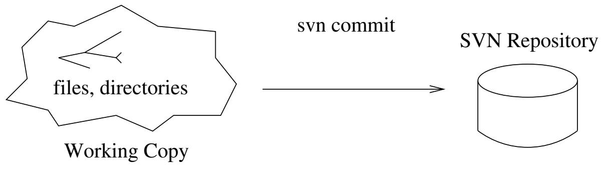
- We can create new files in the working copy, but they are ignored by SVN
- With ‘svn add’ we add files and directories to an SVN internal list, i.e., we “mark” them not to be ignored
- The SVN repository does not know that we added files yet



*Slide notes:*

**‘svn commit’ command**

- ‘svn commit’ will save changes to the repository

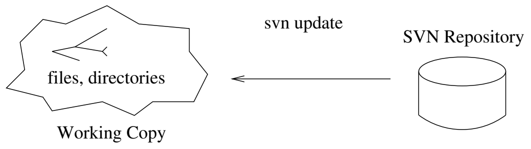


- Changes are saved to the SVN repository
- Local working copy stays (you can delete it if you want, SVN repository does not need to know)
- Remember that you must provide a log message: ‘svn commit -mmessage’

*Slide notes:*

**‘svn update’ command**

- It is possible that someone, or yourself, made new changes in the repository and your working copy has old versions of the files
- ‘svn update’ will update your local copy according to the changes in the repository



- It is a good idea to run ‘svn update’ if you did not modify the working copy in a long time

The ‘svn update’ command gets the newest changes from the repository to our working copy. If we are the only person working on the project at the moment, this will usually make no changes, but if someone modified the project, or even ourselves but from a different working copy, this command will bring our working copy up to date. If we made changes to a file that are not save in the repository yet, they will not be reversed. It may happen that we made a changes to a file and someone else modified the file as well and saved them in the repository. In that case, the ‘svn update’ command will try to merge the changes and create a file that contains both: changes made in the repository and our local changes. If this is not possible, SVN will report a conflict and prepare a few files for us to review and resolve the conflict. There is also an option where we can simply say that we want to ignore our changes and accepted changes in the repository or vice versa.

If we modify some files and try to run ‘svn commit’ in a situation where someone else saved a version that our working copy does not know about, SVN will refuse to commit and instruct us to run and ‘svn update’ first.

Similar situations happen when a new file is added by two different users. The first user to commit will save their version, and the second user will need to run an ‘svn update’ to make merge the files.

*Slide notes:*

**‘svn rm’ and ‘svn mv’**

- If we remove or rename an SVN-marked file using ‘rm’ or ‘mv’, SVN will complain about it and will not remove or rename the file in the respository
- Use ‘svn rm’ to remove a file and remove it form the SVN internal list of marked files
- Use ‘svn mv’ to rename or move a file
- Changes will take affect at the next commit

*Slide notes:*

**SVN Troubleshooting**

- Do not interrupt an SVN operation (unless it takes very long time)
- Helpful commands: 'svn info', 'svn status -v', 'svn log -v'
- A working copy can be recognized by the hidden `.svn` directory
- A way to resolve a problem is to move or remove working copy, and make a new checked out working copy
- If you allow SVN to save your password, you can remove the record with:

```
rm  ~/.subversion/auth/svn.simple/*
```

*Slide notes:*

**SVN and Git**

- There are many Version Control Systems
- Git and SVN are probably the most popular
- Both are open-source, with a lot of similarities and some differences
- 'svn co' is similar to 'git clone'
- 'svn add' is similar to 'git add'
- 'svn commit' is similar to 'git commit' + 'git push'
- 'svn update' is simlar to 'git pull'
- We will cover git in more details later

## A Short Note about 'wc'

*Slide notes:*

**A Short Note about 'wc'**

- You used '`wc`' command in the lab
- `wc` stands for "word count"
- It prints the number of characters, words, and lines
- Options '`-c`', '`-w`', and '`-l`' can be used to print only one of those numbers
- Example: `wc -c file1`
- Concepts: command, arguments, options or flags

## A Short Note about Pipelines

*Slide notes:*

**A Short Note about Pipelines**

- You were asked in the Lab to create a pipeline
- The concept of the pipeline, which belongs to the **pipe-filter software architectural pattern**
- Use pipe symbol '`|`' to connect commands to create pipeline in the Unix command-line interface
- If filename can be specified as the input file, use it only with the first command

## Back to File Permissions

*Slide notes:*

**Back to Permissions. . .**

- We will now continue with the topic of file permissions

### Octal Representation of Permissions

- Permissions can be represented with 9 bits:

$$\overbrace{\texttt{rwx}}^{\texttt{user}}\ \overbrace{\texttt{rwx}}^{\texttt{group}}\ \overbrace{\texttt{rwx}}^{\texttt{other}}$$

- For practical reasons octal system is used
- For example, what permissions are represented by octal number 750?

There are 9 on-off permissions for each file and these are stored as 9 bits in the operating system—bit 0 means off, in other words the permission is not given, and bit 1 means on, or that the permission is given. For example, if a file has permission read, write, and execute for the user, read and execute for the group, and read for others, then this could be represented as the string `rwxr-xr--` or if we use binary digits as `111 101 100`. Since these binary digits are grouped in groups of three it is convenient to represent them in octal representations, since translating a binary number to an octal number can be done by replacing consecutive groups of three digits from the right into octal digits 0, 1, 2, . . . 7. For example, 111 translates to 7, since the binary number 111 corresponds to the number $4 + 2 + 1 = 7$. This means that the above set of permissions `111 101 100` is translated into the octal number 754. Similarily, the octal number 750 would be translated into the binary 111 101 000, which corresponds to the file permissions `rwxr-x---`.

### Checking Permissions

- Command: `ls -l`
- Note: a few more useful ls options: -a -t -r
- Example:

```
$ echo test > tmpfile.txt
$ ls -l tmpfile.txt
-rw-r--r-- 1 vlado csfac 5 Sep 13 11:21 file.txt
```

If we want to check permissions of a file we can do it using the command `ls` with the option `-l` (lowercase letter ‘L’). The command `ls` for listing a directory has several useful options. The command by itself will list contents of a directory by printing all file names. The option `-l` is used to print the long format for the files, which includes file permissions, as well as other information. The option `-a` prints ‘hidden’ entries in the directory, which are all entries whose name starts with a dot (‘`.`’), including ‘`.`’ and ‘`..`’. The option `-t` is useful for sorting entries by the last modification time, and the `-r` can be used to reverse sort. The options can be combined with one leading dash (‘`-`’), so for example, the command ‘`ls -ltra`’ will print the contents of a directory in a long format, including hidden files, sorted by the last modification time in a reverse order; i.e., the host recently modified files will appear the the bottom of the listing.

As an example, if we run the following commands:

```
$ echo test > tmpfile.txt
$ ls -l tmpfile.txt
-rw-r--r-- 1 vlado csfac 5 Sep 13 11:21 file.txt
```

we can see that the file `file.txt` has read and write permissions for the user, the read permission for the group, and the read permission for others. In more details, the line above presents the following information:

| | |
|---|---|
| `-` | — the first character `-` denotes that the file is the regular file; |
| `rw-r--r--` | — denotes the permissions bits for the file 110100100; |
| `1` | — is the number of hard links to the file, that we will se later, or the number of entries in a directory if the file is a directory; |
| `vlado` | — is the user owner of the file; |
| `csfac` | — is the group owner of the file; |
| `5` | — is the size of the file; |
| `Sep 13 11:21` | — is the last modification time of the file; and |
| `file.txt` | — is the file name. |

**Changing Permissions**

- Command: `chmod` *mode files*
- chmod — changing file mode bits
- Some examples:
  - chmod 664 file.txt
  - chmod og-r file.txt
  - chmod u+x,og+r file.txt
  - chmod u=rw,og= file.txt
  - chmod a+r file.txt
  - chmod -R u+r+w+X dir1
- Note: `a` is used for 'all'

---

The command `chmod` is used to change file permissions, or in other words, the file mode bits. Its syntax is:
`chmod` *mode files*
where *mode* specifies the new mode-bit pattern for the files given in the argument *files*. One way is to give the pattern explicitly in the octal form, as in:
`chmod 664 file.txt`
where the number 664 represents in the octal base the new bit-mode pattern for the file `file.txt`. If we translate the octal number 664 in binary, it is 110 110 100, so the new permission for the file are `rw-rw-r--`, or read, write, and not execute for user, read, write, and not execute for group, and read, not write, and not execute for the others.

The octal form is a very direct and oldest form of chmod argument. There are other ways of specifying how permissions bits will be set, but on some older systems they may not be supported.

The other ways of specifying how permission bits will be set are called *symbolic* and they tend to be more intuitive. The following are some examples:

`chmod og-r file.txt --- The part '\verbog,'` stands for others and group, and the part '`-r`' states that the read permission will be turned off for group and others bits. The other permission bits are not changed.

`u+x,og+r file.txt` — We can connect serveral symbolic modifications using comma ('`,`'), so in this example, the execute permission for the user will be set, and the read permission for the group and others will be set. The other bits are not changed.

`chmod u=rw,og= file.txt` — The equal sign ('=') specifies that all three permission bits must be set as exactly given according to 'r', 'w', or 'x' letters included after. In this example, the user bits read and write will be set on **and** the execute bit will be set off. All permission bits for group and others will be set off because there is no 'r', 'w', nor 'x' characters after the second equal sign.

`chmod a+r file.txt` — The letter '`a`' is used to denote *all*, that is user, group and others. In this command the read bit will be set for al three bits, user read, group read, and others read.

`chmod -R u+r+w+X dir1` — The uppercase letter '`X`' for the permissions intuitively means that the execution bit should be set if it "makes sense." This is particularly useful when changing permissions of many files, since the execute permission makes sense for directories and executable files, but not for other types of files. Since it is not straightforward to decided whether a file is 'executable' (we will talk about it later), chmod uses a simpler criterion: it will set execute bit if the file is a directory or at least one of the execute bits (user, group or other) is set. The option '`-R`' in the above command means that chmod will be executed recursively on subdirectories and files of the directory `dir1`, and further on subdirectories of subdirectories, their files, and so on, to deeper levels of directory hierarchy.

You can read more in the text book to learn more details about the `chmod` command.

### Role of File Permissions in Web-sites

One situation that frequently comes up regarding file permissions is when working on web-sites. A web-site is frequently prepared as a set of HTML files in a directory structure. The files and directories usually belong to a user. When we access the web-site using a browser, the files are accessed via a Web server, which is a process owned typically by its own userid, such as 'httpd' or 'apache'. In order to access the site, this process needs read permission on the files and if these are not set properly, we may not be able to access the web-site. For example, to access a file, such as '`/home/joe/public_html/a/b.html`' the web-server will need execute permissions on all directories in the path, such as '`/home`', '`/home/joe`', and so on, and read permissions on the file. Since we frequently do not have a group shared with the web server, we solve the problem by making all directories executable to all, and the file readable to all. This does mean that all users on the system will be able to see the file. If we want to avoid this, there are other ways to approach this issue, but they more complicated.

Similarly, if we want to show the contents of a directory over the web server, we need to make the directory readable to all as well. The web server, such as the one accessing sites on bluenose server, may also need to read a special file called '`.htaccess`' if we create it, and this file needs to be all-readable as well.

### Changing Owner and Group of a File

- Examples:
  - `chown newuser file.txt`
  - `chown -R newuser files dirs`
  - `chgrp newgroup file.txt`
  - `chgrp -R newgroup files dirs`
- `-R` is used for directory recursive change

### Effective UserID and GroupID

- How does the system decide access permission for a process?
- Each process has an effective UserID and GroupID, as well as real UserID and GroupID
- Example: our shell has our UserID and a GroupID
- How are processes assigned effective userids and groupids?

### Changing Effective GroupID and UserID

- `newgrp newgroup`
  - changes into newgroup (logs into new group)
- `su newuser`
  - changes effective user
  - needs to be superuser (root user)
- Additional permission bits: setuid, setgid, and sticky bit bits

**Reading**

- Reading: UNIX book, Ch1 and Ch2 to page 51, so far
- The book contains tutorials on vi and emacs

## 5 Redirection and Pipes

We previously mentioned that each process that we start from shell in Unix is assigned by default three standard Unix communication channels: the standard input (stdin), the standard output (stdout), and the standard error output (stderr). These channels are by default connected to the terminal that a user is using, so the keyboard feeds into the standard input, and the standard output and the standard error output are displayed on the screen. We will learn here how to modify these channels withouth the process in question even detecting the change.

The first modification is by having the standard input read from a file, and the standard output and the standard error output writing to a file; and the second modification is to use pipes to connect the standard output of one process to the standard input of another process.

*Slide notes:*

**Redirection and Pipes**

- The three standard channels: standard input, standard output, standard error output
- Modifying channels: redirection and pipes

### 5.1 Input and Output Redirection

**Output Redirection**

Let us start first with learning how to do output redirections. By default, the standard output of a program is directed to the terminal, so we can see its content on the screen. By redirecting the standard output, we typically store its content into a file, or redirect into the standard input of another process. The redirection syntax is:

```
$ command > filename
```

After this command, the shell will create a file with the name ‘filename’ if it does not exist, or delete the content of this file if it already exists, and redirect the output of command to be stored in the file.

*Slide notes:*

**Output Redirection**

- Remember what we learned about: stdin, stdout, stderr
- Redirecting the standard output of a program into a file: `command > filename`
- Creates a file (filename) if it does not exist
- Example: `ls lab1 > listing`
- Important: ‘>’ redirection deletes previous file contents
- To append a file with new content use ‘>>’
- Example: `ls lab1 >> listing`
- Creates a file (‘listing’) if it does not exist, as well

For example, suppose that the current working directory is the ‘`~/csci2132`’ directory. The symbol tilde (~) is a feature of the Bash shell—it is a short form of the pathname of our home directory. If we would like to list the files in the subdirecotry lab1 and store this listing into a text file named ‘`listing`’, we could enter the following command:

```
ls lab1 > listing
```

If we would like to append the standard output of a command to an existing file rather than overwriting it, we use the '>>' symbol as follows:

```
  command >> filename
```

If the file does not exist, the command will create it, but if the file exist the output will be appended to the existing contents of the file named 'filename'.

For example, if we would like to list the contents of the directory '`lab2`' and appende the output to the existing file named '`listing`' we would use the command:

```
  ls lab2 >> listing
```

**Input Redirection**

The input redirection redirects standard input to be read from a file. We use the symbol '<' to redirect the standard input. This symbol is mnemonic since it looks like an arrow leading input from a file to a program. If we know in advance the complete input that we would type into a process executed as a command '`command`', we can save all this input into a file named '`filename`' and then redirect the file into the process using the command:

```
  command < filename
```

The input redirection feature is particularly useful when testing program. Rather than having to type in the same input everytime we want to do a particular test on a program, we can save input into a file and redirect the file into the program.

Sometimes we can quickly do some tasks that require interaction with a program by redirecting the standard input from a file, or by "feeding" the file into the standard input as we can say. For example, the Unix command '`mail`' can be used to quickly send an email message. If we want to send a program named '`HelloWorld.java`' to a username CSID, we could use the command:

```
  mail CSID < HelloWorld.java
```

or we could use a full email address as follows:

```
  mail a-full-email-address < HelloWorld.java
```

*Slide notes:*

**Input Redirection**

- Redirects the standard input from a file into a process
- Useful in testing
- Syntax: `command < filename`
- Example: `sort < names.txt`
  - sorts names in a file `names.txt` and prints out
- Example 2: `sort < names.txt > names-sorted.txt`
- Example 3: `mail csusername < HelloWorld.java`
- Example 4: `mail full@email < HelloWorld.java`

**Error Redirection**

- Standard error output is not redirected by >
- Syntax (bash specific):
  `command 2> filename`
- Cannot be a space between '2' and '>'
- Example: `rm x 2> error`
- If file `x` does not exist, it will produce an error message
- >> can be used to append output:
  `command 2>> filename`

**More About Redirection**

– File descriptors of stdin, stdout, and stderr are 0, 1, and 2, respectively
– That is where 2 comes from in error redirection
– Similarly we can use 0 and 1 in input and output redirection:
```
command 0< filename
command 1> filename
```
– These are equivalent to previous redirections