Version control with RCS

Johannes Franken
<jfranken@jfranken.de>

On this page I show the use of a version control system, instancing the GNU Revision Control System.

Contents

Version control

The usage of a version control system enables the following operations:

- **Undo:** restore past states of a file's content.
- **Teamwork:** Editing the same set of files with several users. If they happen to change the same file by mistake, the last saving author would overwrite all other user's changes, their work would be lost. A version control system would analyze each user's changed lines and put them together as if they had worked on that file sequentially.
- **Changelogs:** output any changes: per user, time or file/directory.
- **Branches:** If you break down your development to different branches, you can first test each bugfix or configuration change in a development tree, and on success apply it to "at the push of a button" to the official tree.

It's obvious, that using these capabilities has a positive impact on both the efficiency and quality of team works.

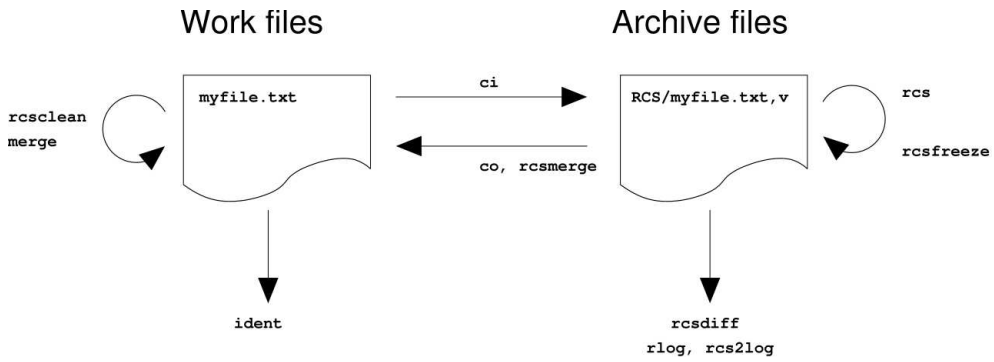
The GNU Revision Control System (RCS)

Overview

RCS was developed from 1992 to 1995 as free software by Walter F. Tichy and Paul Eggert. The following documentation refers to GNU RCS version 5.7, which was released in 1995, but is still the current version today (2003).

Theoretical basics: see Tichy-RCS-Paper (21 pages PDF).

RCS creates one archive file for each work file, in which it stores the evolution as deltas. The transfer to and from the archive files is done by the programs **ci** (for "check-in") und **co** (for "check-out"). In order to protect against mutual overwriting, users can only check into those branches, which they had locked before. And finally, there are some tools for administration and reporting.



Installing and configuring GNU RCS

You ideally install GNU RCS from a package, on Debian 3.0 for example using the following command:

```
$ apt-get install rcs
```

RCS does not require any configuration, but you can preset some of your commonly used options with the **RCSINIT** variable:

Option	Meaning
-q	quiet
-V4 or **-V3**	to check-out from archives, which were created with RCS version 3 or 4
-xSUFFIX	Suffix and directory of the archive files
-zLT	Print the local time and timezone instead of GMT for keywords

Example:

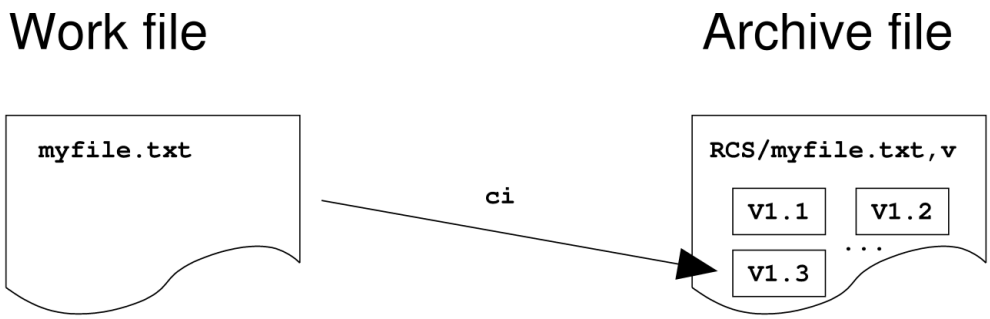
```
$ echo 'export RCSINIT="-zLT -q"' >> ~/.bashrc
```

If you want to separate your work files from the archive files, you can create a **RCS**-directory underneath each directory that holds working files. Archive files will then be put and looked for in that directory, unless you supply a differing location with the **-x** parameter.

Working with archives

ci

ci ("check-in") adds a new version of a work file to its archive file. At the first invocation for each work file it will automatically create the archive file.



Example:

```
$ ls -l
-rw-r--r--    1 jfranken users          15 10. Jan 14:25 myfile.txt
$ cat myfile.txt
RCS-Test
Nr. 1
$ ci -l myfile.txt
myfile.txt,v  <--  myfile.txt
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> test check-in
>> .
initial revision: 1.1
done
$ ls -l
-rw-r--r--    1 jfranken users          15 10. Jan 14:25 myfile.txt
-r--r--r--    1 jfranken users         223 10. Jan 14:25 myfile.txt,v
$ echo + Nr. 2 >> myfile.txt
$ ci -l myfile.txt
myfile.txt,v  <--  myfile.txt
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> addition Nr. 2
>> .
done
$ ls -l
-rw-r--r--    1 jfranken users          23 10. Jan 14:33 myfile.txt
-r--r--r--    1 jfranken users         351 10. Jan 14:34 myfile.txt,v
$
```

You can pass an arbitrary status text after the **-s** parameter (e.g **stable**), to make that version easier to find.

Locking:
Only the user owning the lock (write access) to a branch can check-in a new version. The **-l** parameter makes him keeping the lock, whereby he stays authorized to check-in further changes. The remainder of the users will then see the following message, when they try to check-in:

```
$ ci -l myfile.txt
myfile.txt,v  <--  myfile.txt
ci: myfile.txt,v: no lock set by bfranken
```

Before they can check-in their changes, they need to either

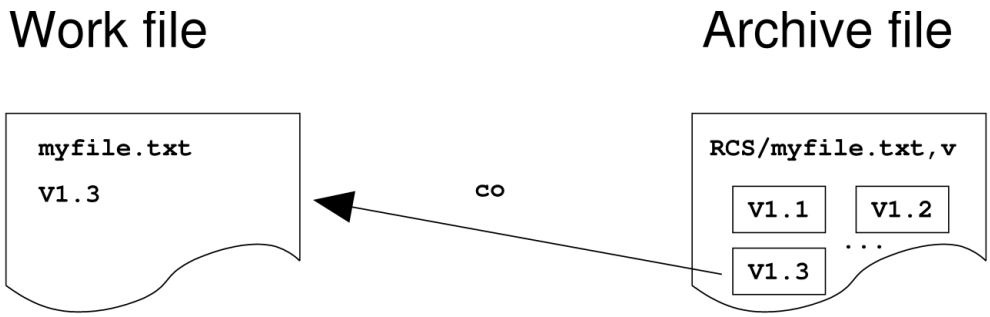
- wait for the lock to be removed (by the owner running **ci** without **-l** or with **-u**) and then adopt the predecessor's changes.
- break the lock administratively (see rcs), or
- opt for another branch (see branches).

The **-u**-parameter prevents **ci** from deleting the working file after the check-in.

More about:
More options: see ci(1) manpage

co

co ("check-out") retrieves a work file from its archive file.



If you append the parameter **-p**, **co** will print the version to STDOUT.

Selecting a version to check-out:
Normally **co** will check-out the most up to date version.

With the **-r** parameter you can specify a different version to check-out:

```
$ ls -l
-r--r--r--    1 jfranken users         337 10. Jan 15:48 myfile.txt,v
$ co -r1.1 myfile.txt,v
myfile.txt,v  -->  myfile.txt
revision 1.1
done
$ ls -l
-r--r--r--    1 jfranken users          15 10. Jan 15:49 myfile.txt
-r--r--r--    1 jfranken users         337 10. Jan 15:48 myfile.txt,v
$
```

With the **-d**-parameter you can request the last version checked-in before a given date; with **-w**, **co** will consider updates by one author only, and with **-s** those matching some status text.

Locking:

co creates the work files write protected. If you want to modify and check them in, you have set the **-l** parameter to apply for a lock.

```
$ co -l myfile.txt
myfile.txt,v  -->  myfile.txt
revision 1.2 (locked)
done
```

This will only work as long as no other user has already locked this branch. If one has, **co** will come up with the following error message:

```
$ co -l myfile.txt
myfile.txt,v  -->  myfile.txt
co: myfile.txt,v: Revision 1.2 is already locked by jfranken.
```

In this case you should ask the owner of the lock to check-in his changes using **ci** (without **-l**), or (if it's an emergency) break the lock administratively with **rcs -u**.

More about:
More options: see co(1) manpage

rcs

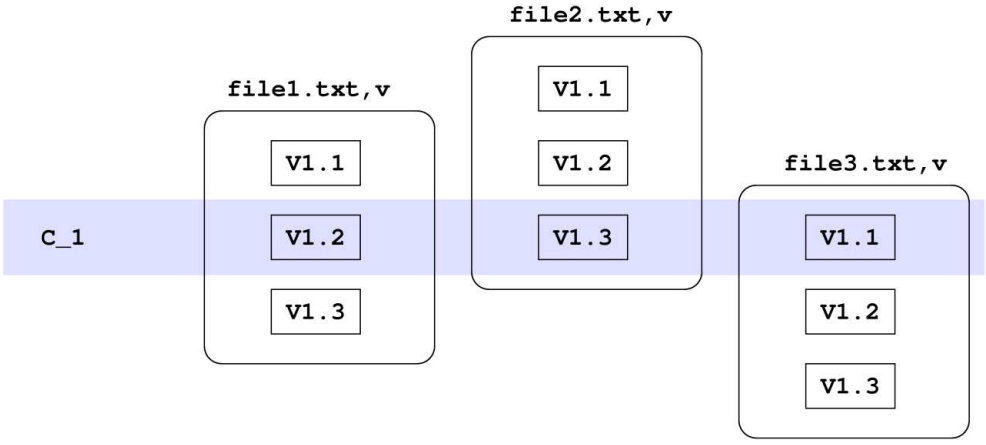
rcs changes archive files. You can use it to

- assign (**-l**) or revoke (**-u**) the lock to/from some user,
- subsequently modify a version's changelog (**-m**) or state (**-s**),
- delete a version (**-o**),
- modify the description (**-t**) or
- grant (**-a**) or deny (**-e**) check-in access to particular users.

More about:
More options: see rcs(1) manpage

rcsfreeze

rcsfreeze is a shell script, which assigns a common version name to the latest version of each archive file.



In this example, **rcsfreeze** was called, when **file1.txt** was at version 1.2 and **file3.txt** at 1.1. **file2.txt** has not been changed since then. **rcsfreeze** labeled this combination **C_1**.

So, what's all that in aid of, you ask? - You can alternatively use a version name in place of a date on check-out:

```
$ for a in RCS/*,v; do co -rC_1 $a; done
```

More about:
More options: see rcsfreeze(1) manpage

rcsclean

rcsclean removes any unlocked (i.e. checked out without **-l**) work files from the current directory.

```
$ ls -l
-r--r--r--    1 jfranken users          23 20. Jan 22:05 myfile.txt
-r--r--r--    1 jfranken users         346 20. Jan 22:04 myfile.txt,v
$ rcsclean
rm -f myfile.txt
$ ls -l
-r--r--r--    1 jfranken users         346 20. Jan 22:04 myfile.txt,v
$
```

If you set the **-u** parameter, **rcsclean** will also remove any locked files, that have not been changed, and release their locks.

More about:
More options: see rcsclean(1) manpage

merge

merge combines the changes, that were made by two authors on copies of a textfile.

Example: Let us assume, a file originally contains the following lines:

```
$ cat orig
Q1:      Why do ducks have big flat feet?

Q2:      Why do elefants have big flat feet?
```

Now, let two users copy and modify that file. Maybe one works out the top part:

```
$ cat changed1
Q1:      Why do ducks have big flat feet?
A1:      To stamp out forrest fires.

Q2:      Why do elefants have big flat feet?
```

and the other the bottom part:

```
$ cat changed2
Q1:      Why do ducks have big flat feet?

Q2:      Why do elefants have big flat feet?
A2:      To stamp out flaming ducks.
```

Then you can have **merge** combine these changes:

```
$ merge -p changed1 orig changed2
Q1:      Why do ducks have big flat feet?
A1:      To stamp out forrest fires.

Q2:      Why do elefants have big flat feet?
A2:      To stamp out flaming ducks.
```

If the users blunderingly changed the same lines, **merge** will mark them as "conflicting":

```
$ echo 'A2:      No idea :-(' >> changed1
$ merge -p changed1 orig changed2
Q1:      Why do ducks have big flat feet?
A1:      To stamp out forrest fires.

Q2:      Why do elefants have big flat feet?
<<<<<<< changed1
A2:      No idea :-(
=======
A2:      To stamp out flaming ducks.
>>>>>>> changed2
merge: warning: conflicts during merge
```

More about:
More options: see merge(1) manpage

Working with keywords

Keywords

If you use the following keywords in your work files, RCS will enrich them them with appropriate values:

Keyword	Value
$Author$	The Unix username of the user who checked in this file
$Date$	Date and Time of check-in
$Header$	Filename (incl. full path), date, author, state
Id	Filename (without path), date, author, state
Log	The Changelog
$Name$	The symbolic name used to check-out this version
$Source$	The filename of the archive file (with path)
$RCSFile$	Filename of the archive file (without path)
$Revision$	The version number
$State$	The status text (defaults to **Exp**)

ident

ident prints any keywords and values contained in the files provided.

Example: I use the wml-compiler to build my webpages (***.html**) from one ***.wml**- and several ***.inc**-files. The **ident**-tool tells me any source versions used for generating each webpage:

```
$ ident -q ssh2.wml ssh2.de.html
ssh2.wml:
     $Id: rcs.wml,v 1.16 2006/01/06 19:54:46 jfranken Exp $

ssh2.de.html:
     $Id: rcs.wml,v 1.16 2006/01/06 19:54:46 jfranken Exp $
     $Id: template.inc,v 1.29 2003/01/02 13:58:47 jfranken Exp
```

As a result you can see, that the target **ssh2.de.html** was derived from **ssh2.wml** in its present form plus **template.inc v1.29**.

More about:
More options: see ident(1) manpage

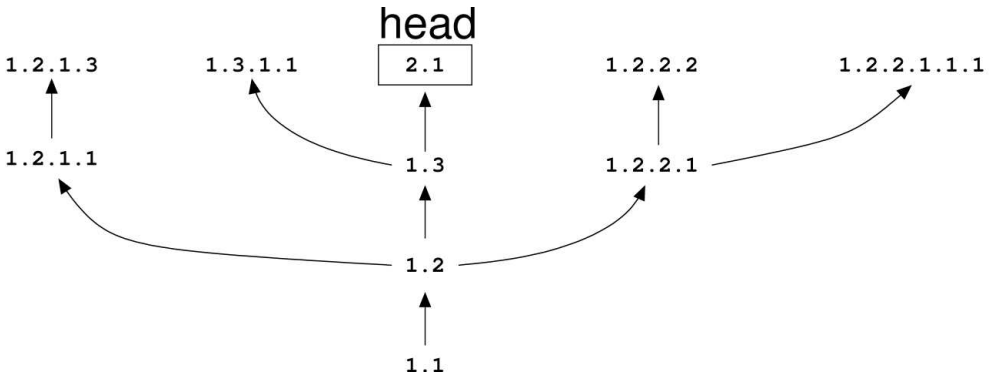
Working with branches

Overview

Branches allow you to test your set of special changes first, and check them in on success, combining them with any other changes that may have appeared in the meantime. This way you have a chance to spot and fix incompatibleness between several bugfixes apart from the official release.

For complex software projects it's advised to permanently practice multiple development trees, like "unstable", "testing" and "stable".

The version number determines the branch a version belongs to: The first branch deriving from **`x.y`** contains the version numbers **`x.y.1.1`** to **`x.y.1.n`**, the second branch **`x.y.2.1`** to **`x.y.2.n`** and so on.



Creating a new branch

To create a new branch, you only need to supply a version number at check-in, which has one more digit than the one it's basing on:

```
$ ci -l -r1.2.1 myfile.txt
myfile.txt,v  <--  myfile.txt
new revision: 1.2.1.1; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> test branch
>> .
done
```

Retrieving a version from a branch

To retrieve a version from a branch, you only need to supply the desired version number at check-out:

```
$ co -l -r1.2.1.1 myfile.txt,v
myfile.txt,v  -->  myfile.txt
revision 1.2.1.1 (locked)
done
```

If you later want to check this version into the same branch, just check-in as usual:

```
$ echo goes to same branch >> myfile.txt
$ ci myfile.txt
myfile.txt,v  <--  myfile.txt
new revision: 1.2.1.2; previous revision: 1.2.1.1
enter log message, terminated with single '.' or end of file:
>> checking ci for branches
>> .
done
```

rcsmerge

rcsmerge applies the changes between some versions to a work file. This way you can combine the bugfixes checked into different branches, to a new, general work file.

More about:
More options: see rcsmerge(1) manpage

Evaluating archive files

rlog

With **rlog** you can peek into archive files:

```
$ rlog *,v
RCS file: myfile.txt,v
Working file: myfile.txt
head: 1.2
branch:
locks: strict
access list:
symbolic names:
        C_1: 1.2
keyword substitution: kv
total revisions: 4;     selected revisions: 4
description:
test checkin
----------------------------
revision 1.2
date: 2003/01/10 13:34:20;  author: jfranken;  state: Exp;  lines: +1 -0
branches:  1.2.1;
addition Nr. 2
----------------------------
revision 1.1
date: 2003/01/10 13:25:44;  author: jfranken;  state: Exp;
Initial revision
----------------------------
revision 1.2.1.2
date: 2003/01/27 06:22:19;  author: jfranken;  state: Exp;  lines: +1 -0
checking ci for branches
----------------------------
revision 1.2.1.1
date: 2003/01/24 14:56:54;  author: jfranken;  state: Exp;  lines: +1 -0
test branch
=============================================================================
```

More about:
More options: see rlog(1) manpage

rcs2log (CVS)

rcs2log is shipped with the CVS packet. It generates changelogs from your archive files. If you don't specify any archive files, it will look for them in all subdirectories.

```
$ rcs2log -v
2003-01-27  Johannes Franken  <jfranken@tp>

        * myfile.txt 1.2.1.2: checking ci for branches

2003-01-24  Johannes Franken  <jfranken@tp>

        * myfile.txt 1.2.1.1: test branch

2003-01-10  Johannes Franken  <jfranken@tp>

        * myfile.txt 1.2: addition Nr. 2

        * myfile.txt 1.1: New file.
```

More about:
More options: see rcs2log(1) manpage

rcsdiff

`rcsdiff` shows the differents between two versions of a file.

Examples:

- Comparing the work file against the archive file's most up to date version:

```
$ rcsdiff -q myfile.txt,v
$
```

  Result: No differences. The work file has not been changed since it was last checked in or out.
- Comparing the work file against an older version from the archive file:

```
$ echo 'and Nr. 3' >> myfile.txt
$ rcsdiff -q -r1.1 myfile.txt,v
2a3,4
> + Nr. 2
> and Nr. 3
$
```

  Result: In contrast to version 1.1, the work file has additional lines 3 to 4, which append after line 2 and contain some text a/m.
- Comparing two versions from the archive file:

```
$ rcsdiff -q -r1.1 -r1.2 myfile.txt,v
2a3
> + Nr. 2
$
```

  Result: The difference between versions 1.1 and 1.2 is, that version 1.2 has a third line appended after the second, containing **`+ Nr. 3`**.

More about:
More options: see rcsdiff(1) manpage

Outlook: CVS

The Concurrent Versions System (CVS) extends RCS by the following capabilities:

- Subdirectories
- Server-interface (pipes or TCP-ports)
- Automatic execution of scripts at check-in (automated build system, ABS)

More about:
More options: see cvs(1) manpage