Glibc Installation HOWTO

Kai Schlachter

`<linux (at) murphyslantech.de>`

2004−03−19

Revision History		
Revision 1.01	2004−03−19	Revised by: KC
Initial Release, reviewed by LDP		

Table of Contents

Chapter 1. Preface

1.1. Copyright

1.2. Revision History

- v1.01: Fixed some missplaced quotation marks

1.3. Thanks

I especially want to thank Nico Schmoigl for helping me get my crashed system back up after I ran **make install** for glibc while my system was still fully running. This was the main reason I wrote this mini−HOWTO.

Thanks also go to all the other people who helped me along the way; getting the HOWTO in correct format, finding all the typos, etc.

Chapter 2. Introduction

In this HOWTO I will explain how you install a new version of glibc on your system.

I wrote this manual because I want to save others from the problems I encountered.

This HOWTO is a kind of guideline, made up from settings and ways of doing things that worked out for me. No liability for the contents of this document can be accepted. Use the concepts, examples and other content at your own risk. You are strongly recommended to make a backup of your system before major installations and backups at regular intervals.

If you have any suggestions, found another bug in some distribution and how to fix it, please let me know by writing me an e–mail: <linux_at_murphyslantech.de>.

2.1. Why?

Yes, that's a good question. Why would you need to install a new glibc? Well, there are several reasons:

- you are a developer and you need the new functions of the library
- you wanted to compile a new program that needs the new library
- you like the thrill of errors and bugs in new versions ;–)

2.2. What?

If you do not know what glibc is by now, don't worry! When I first had problems with a new program I wanted to compile, I only knew that my version of glibc was not sufficient for the compile. Now that I know more, I will try to explain in a very simple way what glibc does.

The glibc package contains a library written in the programming language of C. Libraries are a very useful thing in programming; instead of inventing the wheel from scratch for operations such as computing the square–root of a number, such common functions are stored in separate files the so–called libraries. When a new version of a library is published it often contains some new functions, uses more efficient algorithms for already implemented functions, and so on.

This is the reason why some programs complain about an older version of glibc: the current version just does not contain all the functions the program needs in order to run.

I know this is not technically correct in all detail, but it gives you a basic understanding of the architecture behind.

Chapter 3. Preparations

As an installation of glibc is not an easy task. You need to do a lot of things in advance, especially in the event that something goes wrong. (That is what happened to me on my first glibc install, and I did not make any preparations.)

3.1. Stuff you will need

Basically, you need two different things: software that is already running on your machine (eg., pre−installed by your distribution) and packages of source−code for different programs.

3.1.1. Software you will need

- a running gcc
- an older version of glibc ;−)
- GNU−binutils
- GNU−make
- the GNU−core−utils
- GNU−tar
- bash or any shell you prefer
- very useful but not a must: Midnight Commander
- an editor you like (vi, jed, etc.)

3.1.2. Sources you will need

- bashor the shell you like
- GNU−tar
- GNU−core−utils
- GNU−make
- of course:glibc
- possibly: gcc

3.2. Special things you need to do

Since you are going to substitute the basic library many programs rely on, you can imagine the problems that may occur.

For me, it so happened that everything went fine until I typed in **make install**. At about halfway through the installation process I got an error telling me that **rm** was not able to run, and I found out that even all the common commands like **cp**, **ls**, **mv**, **ln**, **tar**, etc., did not work; all told me that they were not able to find parts of the library they needed.

But there is help available. You can force the compilation of programs with the libraries compiled into them, so the programs do not need to look them up from the library.

For that reason, in this chapter, we will compile all the utilities we need for the install into a static version.

3.2.1. Things you will definitely need

3.2.1.1. The GNU−Binutils

1. Get the newest version from: ftp.gnu.org/gnu/binutils; at the time of writing, this was version 2.14
2. Open the package:
   ```
   tar xIvf binutils−2.14.tar.bz2
   ```
3. Change to the directory:
   ```
   cd binutils−2.14
   ```
4. Configure the Makefiles:
   ```
   ./configure
   ```
5. Compile the sources:
   ```
   make
   ```
6. Install them with:
   ```
   make install
   ```

If you run into trouble with the compilation of the binutils, referring to problems with gettext (indicated by errors like: "undeclared reference to lib_intl" or similar) please install the newest version, available from ftp.gnu.org/gnu/gettext.

If this does not help, try disabling the native−language support by using:

```
./configure −−no−nls
```

You don't need to build a static version of the binutils, though it would not hurt, but I encountered many systems running with very old versions and ran into errors almost every time, so I think it is a good idea to mention them here.

3.2.1.2. GNU make

The **make** command is responsible for the compiling of the sources, calling gcc and all the other programs needed for a compile. Since you may need to compile something if a problem occurs with the new glibc, it is a good idea to have it static, otherwise it might not work after an error appears.

1. Download the source from ftp.gnu.org/gnu/make/; at the time of writing the current version was 3.80
2. Unpack the source, eg.:
   ```
   tar xIvf make−3.80.tar.bz2
   ```
3. Change to the created directory:
   ```
   cd make−3.80
   ```
4. Take care that the binaries are built static:
   ```
   export CFLAGS="−static −O2 −g"
   ```
5. Run the configure script:
   ```
   ./configure
   ```
6. Compile the stuff:
   ```
   make
   ```
7. Install the binaries:
   ```
   make install
   ```
8. Make a check:
   ```
   make −v
   ```

You should now see the new version installed. If not, check for older binary files and replace them by smlinks to the new version.

Congratulations! You have compiled another static−linked program.

3.2.1.3. the GNU core−utils

The core−utils are commands like: **cp**, **rm**, **ln**, **mv**, etc. In case of an error in the installation, these are an absolute requirement to help bring your system up again, so static binaries are really necessary here.

1. Again, download the source tarball from: ftp.gnu.org/gnu/coreutils/; at the time of writing, version 5.0 was current.
2. Unpack it:
   ```
   tar xIvf coreutils−5.0.tar.bz2
   ```
3. Change to the directory:
   ```
   cd coreutils−5.0
   ```
4. Take care that the binaries are built static:
   ```
   export CFLAGS="−static −O2 −g"
   ```
5. Configure the package:
   ```
   ./configure
   ```
6. Compile the binaries:
   ```
   make
   ```
7. And install them:
   ```
   make install
   ```
8. Verify that the right core−utils are used:
   ```
   cp −−version
   ```
   . You should see the correct version, otherwise remove any old binaries and replace them with symlinks to the new version.

Now that the binaries of these very elementary tools are static, you can be sure they will work every time you need them.

3.2.1.4. GNU tar

You have already used GNU tar to unpack all the programs compiled and installed so far. But maybe you need to compile another program which is needed by glibc after you had a crash, and in this situation (I experienced this myself!) it is very useful to have a working **tar** ready to unpack the missing programs. With tar, we also need to take care of the bz2 compression algorithm, which is not included in the normal source distribution of tar.

1. Get the source of GNU tar from ftp.gnu.org/gnu/tar; at the time of writing, version 1.13 was up−to−date.
2. As many source tarballs are compressed with bzip2, we would like to have the support built in, rather than working with pipes, so get the patch from: ftp://infogroep.be/pub/linux/lfs/lfs−packages/4.1/tar−1.13.patch.
3. Unpack the source by invoking:
   ```
   tar xzvf tar−1.13.tar.gz
   ```
4. Copy the patch to the source directory of tar:
   ```
   cp tar−1.13.patch tar−1.13/
   ```
5. Apply the patch:

```
patch -Np1 -i tar-1.13.patch
```

6. Set the compiler flags to make a static binary:
   ```
   export CFLAGS="-static -O2 -g"
   ```
7. Now we are ready to configure:
   ```
   ./configure
   ```
8. Compile with:
   ```
   make
   ```
9. And as the next step, install the package:
   ```
   make install
   ```
10. Do a quick check to ensure the new version is being used from now on:
    ```
    tar --version
    ```
    The version you just installed should display, otherwise check for old binaries and replace them with symlinks to the new location.

If you experience problems with the execution of **make**, try to turn off native–language support (nls). You may achieve this by invoking configure with the option:

```
--disable-nls
```

Note: In this new version of tar, you must use the `-j` switch to decompress .bzip2 files, so instead of

```
tar xIvf anyfile.tar.bz2
```

you now have to use

```
tar xjvf anyfile.tar.bz2
```

I do not know why this was changed, but it works fine.

3.2.1.5. The Bash shell

I prefer Bash as my shell; if you use a different one, please be sure you have installed a static version of it before you install glibc.

1. Get Bash from: ftp.gnu.org/gnu/bash/. Download the newest version you can find; at the time of writing this was version 2.05b.
2. Unpack the source tree:
   ```
   tar xzvf bash-2.05b.tar.gz
   ```
   which will create a directory called `bash-2.05b` with all the unpacked sources in it.
3. Go to the directory:
   ```
   cd bash-2.05a
   ```
4. Set everything up for building a static version:
   ```
   export CFLAGS="-static -O2 -g"
   ```
5. Configure the makefiles:
   ```
   ./configure
   ```
   If you would like something special in your Bash, see
   ```
   ./configure --help
   ```
   for a list of options.
6. Compile everything:
   ```
   make
   ```
7. Install the compiled binaries:

```
make install
```

This will install the binaries to `/usr/local/bin/`.

8. Make sure there is not another version laying around (like in my Suse−Linux: `/bin/`), by copying the file:

   ```
   cp /usr/local/bin/bash /bin/
   ```

   We don't use a symlink here because both at boot−time and when starting Bash there might be trouble with symlinks.

You now have installed a static version of Bash. For that reason, the binary is much bigger than usual, but it will run under all circumstances.

If you prefer to use another shell, you are free to do so, but make sure it is a statically−linked version. Feel free to email me a method to build the shell of your choice in a static version, and chances are good that it will be implemented in the next revision of this document.

3.2.2. Software that may come in handy

3.2.2.1. Midnight Commander

Midnight Commander is a very useful file manager, supporting many nice features like transparent decompression of packed files, built−in copy, move and other common commands, as well as an integrated editor.

To compile this piece of software, you will need to have glib installed; in some distributions this is already the case. If you get an error in the **make** command saying that ld could not find glib, you will need to install this library first. You can get the sources from: ftp.gnome.org/pub/gnome/sources/glib/2.2/, and the installation is straight−forward.

Here are the steps to build Midnight Commander:

1. Get the source from http://www.ibiblio.org/pub/Linux/utils/file/managers/mc/"; at the time of writing, the newest version was 4.6.0.
2. Unpack the sources:

   ```
   tar xzvf mc−4.6.0.tar.gz
   ```
3. Change to the directory you just created:

   ```
   cd mc−4.6.0
   ```
4. Set up the configuration−files:

   ```
   ./configure
   ```
5. Start compiling:

   ```
   make
   ```
6. Install everything:

   ```
   make install
   ```

Chapter 4. The installation of glibc itself

Now we come to the most important thing: the glibc install.

4.1. Obtaining and compiling the source

There are several versions of glibc available, but not in all cases are new versions really better than the old ones. The best thing you can do to find out which works and which ones you should not use is to check out the different forums on the Internet. If you have someone to ask, talk to him about the topic. Maybe he already has installed the new version and might be able to tell you that version x.y.z is a PITA, but version a.b.c works really well!

I decided to install glibc−2.2.4, as I was told it works well, but it is your decision which one to choose.

Ok, now let's go to work:

1. Get the source from ftp.gnu.org/gnu/glibc/; as I said, I used version 2.2.4.
2. Unpack the source:
   ```
   tar −xzvf glibc−2.2.4.tar.gz
   ```
3. In addition, you will need a package called "linuxthreads," found in the `linuxthreads` directory on ftp.gnu.org. The file is called:
   ```
   glibc−linuxthreads−2.2.4.tar.gz
   ```
   Make sure you get the version that corresponds to your glibc source tree.
4. Copy the linuxthreads package to your glibc source directory:
   ```
   cp glibc−linuxthreads−2.2.4.tar.gz glibc−2.2.4
   ```
5. Change to the glibc directory:
   ```
   cd glibc−2.2.4
   ```
6. Unpack linuxthreads:
   ```
   tar xzvf linux−threads−2.2.4.tar.gz
   ```
7. Configure the package:
   ```
   ./configure −−enable−add−ons=linuxthreads
   ```
   This will configure the package in such a way that the linuxthreads are included in the compile; this is necessary for compliance with other Linux systems. For example, programs you compile will probably not run on another machine if you forget to include this package.
8. Afterwards, start the compilation of glibc:
   ```
   make
   ```
   This may take some time (about half an hour on my Duron XP, running with 1.5 GHz).

Now that the library is compiled, everything is ready for the installation, but things are not as easy this time.

4.2. The installation

To install glibc you need a system with nothing running on it, since many processes (for example sendmail) always try to use the library and therefore block the files from being replaced. Therefore we need a "naked" system, running nothing except the things we absolutely need. You can achieve this by passing the boot option

```
init=/bin/bash
```

to your kernel. Depending on your bootloader you may need to do different things. In the following I will explain this using the two most common bootloaders, LILO (LInux−LOader) and GNU grub, as examples.

4.2.1. LILO

To start the "only−basics" system, reboot your computer and at the LILO prompt enter the kernel image−name you like to load and append

```
init=/bin/bash
```

to it before pressing **Return**. If you are planing to replace your glibc more often, it might be a good idea to add a separate configuration to your `/etc/lilo.conf`. For details, refer to the man−page of LILO.

4.2.2. Grub

Grub is a newer bootloader, with enhanced support for different operating systems and and file system types (eg. it supports booting from reiserfs partitions). If you would like to know more go to: http://www.gnu.org/software/grub/, where you will find all the stuff you need.

If you already have Grub installed, you probably use the text−based front−end to select the kernel you prefer to boot. Grub has a nice feature instead of returning to doing everything by hand, you can simply select your entry and type **e**, which will bring up an option menu. In this menu you will see the commands Grub executes prior to booting the kernel. Simply select the line saying

```
kernel="/where/your-kernel-is and-options-are"
```

and press **e** again. Now you can edit this line. Here you just add

```
init=/bin/bash
```

and after pressing **Return** to make the changes take effect, press **b** to start booting.

4.3. After the kernel is booted...

...You will find yourself in an absolute minimal bash−environment.

You will not even be asked to enter your username or password! At this time you are the ultimate super−user; no−one can get around you because the system is in single−user mode, so be careful what you are doing. There are no file−rights set or anything else!

Your prompt will probably look like this:

```
init-x.y#
```

At this moment your root (/) has been mounted as read−only, thus you will not be able to write the new library to your hard drive. To make it `r/w`, enter the command:

```
mount -o remount,rw /
```

If your source is located on another partition you must also mount it, as it is not done for you (for me this means mounting my raid system):

```
mount -t reiserfs /dev/md0 /usr/src
```

As you see, I defined the file system type, which is needed because **mount** does not look anything up in `/etc/fstab`.

Now you can go to the directory containing the source and type in:

```
make install
```

If you like, now might be a good time to pray that everything works out fine... ;-)

If everything went through properly, you will return to your prompt after the installation without any error message. In all other cases, please see Chapter 5.

If everything goes fine, run:

```
ldconfig -v
```

to update your library cache.

Congratulations! The library is successfully installed. Now type in: **mount -o remount,ro /** to ensure that all the data is written to the hard drive.

Now start the reboot:

```
exit
```

This will cause an error message saying that you have caused a kernel-panic. If possible, restart the computer by using **CTRL-ALT-DEL**, otherwise use your hardware's reset switch.

Try booting your normal kernel. If everything turns out fine, you are ready to use the new library.

Chapter 5. Troubleshooting if something goes wrong...

If you come to this section and you have followed all the instructions above, you probably encountered the problem of different Linux distributions, some of which do not keep stuff where it should normally be, but somewhere else. The Suse Linux distributions are most famous for such stupid quirks, but there may be others having similar problems. If you encounter such a thing and you find the cause for the trouble and hopefully a solution please let me know and I will add this to my HOWTO.

I think this section will never really be complete, but I will list some possible errors and solutions for these issues.

5.1. Errors with configure or make while trying to compile glibc

Sometimes you get configuration error telling you that, for example, a requirement is not met typical for software or other library−packages which are to old. I encountered this with a series of programs, especially during the compile of all the static software. Normally there should be no problem: get an up−to−date version of the software or libraries needed and then compile them according to the description found in the source tree (usually called `README`, `INSTALL` or something similar).

But there are sometimes cases where this just does not want to work. For example, I experienced problems compiling a new version of my binutils (one of the reasons why I mention them in the requirements), as I needed to compile glibc. In return, the configure script for the binutils told me, "Your glibc is to old!" So I thought, *Here the snake starts eating its tail*. Thankfully there is a solution for this problem: if you can not take one big step, try taking smaller ones, but more of them.

In my distribution, a glibc with the version number 2.1.1 was included. To get around the error I tried to compile version 2.1.3, which was no problem. After I installed this version of the library, I tried to compile the binutils once more, and this time all the requirements were met.

If you encounter such a "loop," try to find out what version of the software required is the minimum, then download that version (I think that is one of the reasons why so many old versions are still lying around on the FTP servers). After successfully compiling and installing, try to build the software that complained about the version again; in most cases you should now be able to compile. It may be necessary that you continue to use this method to compile missing or old software. This is what I call "the long tail of the rat" or "domino−effect." You only wanted to do one thing, but you needed to do many more before you could make the move you wanted to. It might be very nasty, but there is one good side to it: afterwards you can be quite sure that many of the programs which were really old will replaced by the time you finish your installation.

5.2. Something goes wrong during make install

The most common mistake is not to have a set of basic, static tools; in this case you can only use the command **cd**, but nothing else. This is the reason why in this HOWTO I have described in detail how to make those tools static.

The only tool not static is **mount** and, for good reason in my opinion, it is included in the package of linux−utils, which also contains **login**, **passwd**, etc. Since you are not able to use statically−linked versions in combination with PAM or other security−related software, it would be unwise to compile them statically under all circumstances. Of course you are free to do so if you are really sure about what you are doing.

5.2.1. Going back to a working configuration

The way back to a working configuration is quite simple if you have the static tools: go to the directory `/usr/local/lib/` and move all the newly installed files into another location (eg. `/usr/local/lib/storedaway`). You may identify them by looking at their version number, which should be the same as the one from your glibc installation (in my example all files fitted to the scheme lib*−2.2.4), and of course by the creation date and time. It is quite uncommon that two different libraries have the same version number at the same time I, myself, have never experienced such a thing but just to be sure you do not delete something important to your system, check the date and time of creation. A very useful tool in this case is the Midnight Commander, if you have it installed.

You could try to remove the files `ld−2.2.4.so` and `libc−2.2.4.so` and run **ldconfig −v** afterwards, before removing all the crashed files. This will enable you to use at least most of your programs and in every case you will be able to run the Midnight Commander.

Do not forget to do at least one **ldconfig −v** after you have removed all the files.

5.2.1.1. Removing the causes for the crashing of the installation

A common cause for problems is that your distribution has stored all the library files in a different location than the newly−installed routine will use, thus it often happens that there are two versions running simultaneously, disturbing each other. In my case, lots of trouble was caused by a second copy of `libc6.so` lying around in `/lib`, a symbolic link from this file to the corresponding one in `/usr/local/lib` fixes this problem.