Rock Linux: Not for woozies!

Jos Visser 14-03-2000

Introduction

And thus it came to pass that my buddy Pjotr Prins pointed me to the existence of Rock Linux. In his unsurpassable style he explained to me that Rock Linux would probably become *the* Linux distribution for experienced Linux administrators and programmers who were (or would get) tired by the all-singing-all-dancing Linux distributions being put out by companies such as Red Hat, SuSE and Mandrake. A couple of weeks later, Pjotr manned a little booth at the NLUUG 1999 autumn conference, from which he sold stuff like Hacker Keyboards, plush penguins *and* Rock Linux CDROMs. When asked by someone what the particular advantage of this distribution was he pointedly replied: "It is more difficult to install". Obviously, after such a sales pitch, I could not resist checking out this new distribution.

Don't get me wrong, I think the "traditional" (ha ha) Linux distributors are doing a great job bringing Linux to the masses. Their additions to the "raw" software include such niceties of life as graphical installation programs, GUI driven control panels and other "user friendly" extensions. For a very experienced Unix hacker like myself, all this is becoming a bit too much; for instance I really hate it when I change a configuration file like /etc/resolv.conf, only for that change to get reversed at the next run of the idiot-proof vendor-supplied configuration utility (such as SuSEConfig). Another absolute no-no for me is a default alias rm='rm -i' which I spotted in some distributions. These kind of things instantly turn my spine in a blubbering piece of jelly and I need to write at least three sendmail rewrite rules to recuperate and regain my senses.

And so I decided to spent some of my valuable spare time investigating whether Rock Linux (my nickname: The Rock) might become my distribution of choice.

Intro to The Rock

Before I delve into a description of The Rock, it might be prudent to first read Clifford Wolf's (Rock Linux's project leader) [eup] E-zine article on Rock Linux Philosophy.

A "normal" Linux distribution consists of a number of components:

- A Linux kernel
- A C library (nowadays usually GNU LibC (glibc))
- The GNU C compiler and utilities
- The XFree86 X Window System
- A gazillion number of tools, utilities, add-ons, window managers and other stuff that we seemingly can not live without.

A Linux distributor's job is to bring such a system to you in one easy to install and handy to use package. They get all the sources, check the dependencies, apply patches, compile all packages, create default configuration files, write an installation program and burn the result on a CDROM. In order to distinguish themselves from other distributions, most big distributors try to add value by packaging a manual and non-free or commercial software such as ApplixWare, StarOffice or the

OpenSound system with their distribution. The result typically sells for 50 Euro and up.

The big advantage of the typical "big" Linux distribution is that it gets novice users flying in almost no time. Installation is usually a breeze, and through the integration work of the distributor and the default configurations supplied by them most stuff works immediately: you start KDE and it is completely pre-configured with menus, icons, the lot. However, the problems start when you start tinkering with the system: you can not upgrade software easily, because then the built-in software package manager start complaining; when you change a configuration file, the distribution's own configuration manager starts reversing your changes; libraries and packages turn out to have been built with compilation flags other than your preferences.

So, you want control, and that is where The Rock comes into play. Clifford Wolf created a Linux distribution unlike any other: A distribution for experienced administrators where *you* are in the driving seat.

Rock Linux is only a two megabyte download. "Only 2MB?" I hear you think? Correct! And that is because these 2MB only contain the programs necessary to download The Rock's components, configure and compile them, and create an installation CDROM. The Rock does not contain fancy integration or default configurations; when you do a Rock Linux build, you download all necessary Linux components from their primary FTP sites, unpack and compile them all, and then integrate the result on a CDROM which you can subsequently install. But, let's not delve into that into too much detail right now, just let me tell you of my experiences with Rock Linux.

The Rock 1.3.4

At the time I started to look into Rock Linux, Clifford had just put out Rock Linux 1.3.4. Like Linux kernels, even numbered releases (1.2.0) are considered stable, and odd numbered ones (1.3.x) are potentially unstable development versions. Since I like living dangerously (not to mention I was/am in research mode) I decided to go with the "unstable" 1.3.4. Obviously, "unstable" is somewhat of a misnomer. Most so-called "unstable" Linux versions are more stable that some operating systems I could mention....

Anyways, I downloaded the Rock Linux 1.3.4 distribution, unpacked it, and read the installation notes provided by Clifford. To build The Rock you need a running Linux system with about 2GB free disk space and which has support things such as ramdisks and NFS. My primary system is an AMD K6 laptop with 64MB ram and a 6GB hard disk, running SuSE Linux 6.1 (kernel 2.2.13). However, to free up 2GB disk space would mean removing my MP3 collection which I obviously would not contemplate in any situation :-).

Fortunately my laptop's hard disk is easily replaced through a slot on the right hand side next to the PCMCIA bay, and I decided to use my spare 6GB hard disk to build The Rock. Using an NLUUG promotion CDROM I installed SuSE Linux 6.2 on the spare hard disk.

The first thing one should do when building Rock Linux is downloading the packages that make up the distribution. The Rock Linux distribution (you remember, the 2MB) contains configuration files outlining which versions of which packages are needed, and where these can be downloaded. You download the packages by executing "make-misc download", which uses GNU wget to retrieve the packages from the Internet. We're talking >200MB download here, so a fast Internet connection comes in quite handy. Fortunately, my home office is hooked up to the Internet through a cable modem, so a 200MB download is not really a problem. However, notwithstanding the cable modem, I estimated that downloading would take considerable time, so I decided to execute this step on one of my desktop systems (which also ran SuSE Linux 6.2 at that time). I thought that I could easily transfer the entire archive directory to my laptop once every package was on-board. I started the "make-misc download", and ..... nothing happened!

Hmmmm... Time to scratch my head a bit. I could see that the download program tried to contact the FTP server to download the first package, but then it seemed to hang. It dawned on me pretty quickly

that the program had tried an ordinary FTP get, whereas my firewall only allows passive FTP's (for security and other reasons). As a quick fix, I edited the make-misc script to issue wget with the --passive-ftp option. I restarted the download, and everything seemed to go swell this time. I then retreated for the night. (Later, Clifford pointed me to the /etc/wgetrc and the $HOME/.wgetrc files in which you can make passive FTP the default mode for FTP downloads).

The following morning I immediately checked the status of the download, and to my dismay not all packages had been downloaded. Fortunately the download step, as are most of The Rock's build procedures, is completely restartable, so I just restarted the download, saving the screen output in a file for later analysis. This second download had no more success than the first, certain packages eluded download. Analysis of the download's output revealed the two sources of the problem: First of all, in the dynamic environment of open source development, some packages had already been replaced by a higher version or release, and the package's maintainers had removed the previous version from their sites. This was for instance the case with ETerm (0.8.10 available, 0.8.9 referenced in The Rock) and the binutils (2.9.5.0.19 instead of 2.9.5.0.16). The second problem source was that some sites were simply down or otherwise unavailable.

To solve the version problem, you can find out which version *is* currently available, and change Rock Linux's configuration files to reflect the new package version. Doing this is quite simple. For each package included in Rock Linux you can find a subdirectory containing version information, name of the package file, the download site(s) for the package and instructions on how to configure and build the package. Changing this information is quite easy, and after making the changes you only need to run "make-misc puzzle" to put the various pieces of the puzzle together into a new set of download and build configuration files. However, changing package versions is not without "risks". Chances are that the new package version has changed so much that it will break the build (i.e., needs different building instructions or requires changes in other packages). So obviously, this should not be done lightly. I changed the version number on ETerm and binutils to get the process going, but this strategy would not be very handy for most of the missing packages.

If you experience problems downloading packages, The Rock's build notes suggest you download the missing packages manually from other FTP sites. This is of course completely out of the question. I do not posess all these programming experience and skills to start doing these kind of things manually. So, instead, I wrote a Perl script called "getpkg.pl". This script feeds on an input file naming packages to be downloaded (e.g.: ETerm-0.8.9.tar.gz). "getpkg.pl" first contacts http://ftpsearch.lycos.com to search the first 500 download sites for these packages; it extracts these sites from the HTML reply, sorts them according to top level domain and then proceeds to try and download the package from the sites. The preference order for domains can be configured through a variable, in my case I first try to download from sites in the .nl domain, then the .se domain, then the .uk domain and so forth. From the failed download's output I extracted a list of packages that failed to download (vi rules!) and I put getpkg.pl to work. My little domestic effort paid off quite nicely, getpkg.pl retrieved the missing packages in a breeze from various sites across the Internet. (By the way, the only headache I've had so far with getpkg.pl is that certain sites contain damaged archives for some of the packages that I needed).

After the download succeeded (which you by the way can test with "make-misc testdown"), I moved the entire Rock Linux 1.3.4 build tree from my desktop system to my laptop (now running from the spare hard disk), configured the build environment (setting up NFS, making directories, creating symbolic links, editing the Rock configuration file) and started the Big Build.

The Rock Linux build proceeds through a number of stages. The ultimate goal is to build a completely functional Linux system (from the sources) in a subdirectory of the Rock Linux build tree. In the first stage the build process primes this subdirectory with an absolute minimal Linux system containing the GNU C compiler, the C libraries and other essential features. In the subsequent stages, the build process does a chroot() to this subdirectory, and builds the rest of the system. The sources of the packages are retrieved using NFS (a clever way to reach outside of the the chroot()-ed environment). A Rock Linux build can take >20 hours, even on a relatively fast machine, so there is no real need to stare at the screen waiting for it to finish.

After a couple of hours my attention was drawn towards the faithfully humming laptop by a series of beeps. It turned out that the compilation of the "shadow" package failed because of a dependency. After some investigation it turned out that make tried to re-create the aclocal.m4 because the configure.in had recently changed. I digged some more, and it turned out that The Rock's build had patched configure.in to work around some bug or another. Because configure.in was now newer than the shipped aclocal.m4, make tried to rebuild aclocal.m4 with the aclocal program, and this failed due to some undefined m4 macro. Now, I must confess that I am not well versed in the inner workings of autoconf and aclocal, so I decided that a quick hack would have to suffice. I patched The Rock's configuration files to touch aclocal.m4 after applying the patch, rebuild the puzzle, and restarted the build.

Again, some hours later, another series of beeps disturbed me. This time, compiling the procinfo package failed because it couldn't find the termcap library libtermcap.a (-ltermcap). After some digging it turned out that SuSE Linux 6.2 (my build system), for one strange reason or another, hides the termcap libraries in the /usr/lib/termcap directory, instead of directly in /usr/lib. This means that a link step that includes libtermcap.a must include /usr/lib/termcap in some way or another. Being thoroughly in hacking mode, I simply copied libtermcap.a to /usr/lib, and restarted the build.

Building The Rock is such a lenghty process that I finally got round to spending some quality time with my family. My laptop was still busy building Rock Linux when I went to bed. The following morning when I got up, I immediately checked the status of the build. To my dismay, I saw that the build had barfed out again (but now somewhere in the second stage). Compiling the ncurses package had failed with an "illegal instruction" when executing the TermInfo compiler "tic". The ncurses package executes tic during its build to create the compiled TermInfo directory from a set of terminal descriptions delivered with the package. That tic exploded in mid-flight was somewhat disturbing and I spent quite some time investigating this problem. When executing tic on my build system (SuSE Linux 6.2), it worked flawlessly. When executing the tic built by The Rock Big Build build tree, it worked flawlessly as well. Huh? Then it dawned on me that the second stage build executes in a chroot()-ed environment. I did a chroot to the virtual Rock Linux environment, and executed tic from there. Again, it barfed out with an illegal instruction. I then spent some time with the GNU debugger to find out where it actually went wrong, and it turned out to be somewhere in a shared library that is used by tic. Some more deep thinking later I remembered that when initally configuring the Big Build I had selected options suitable for an Intel 686 processor, under the impression that a K6 was a 686 compatibled. I rebuild the shared library in question with i386 compatible flags and the problem disappeared. I reconfigured the build for an i386 and started it from scratch (the latest Rock Linux versions support the K6 as a specific processor that the build can be configured for).

The new build proceeded quite nicely, and barfed out only once during the compilation stages due to a trivial problem when compiling the X11 screensaver package (the /opt/kde/bin directory did not exist in the virtual Rock Linux chroot()-ed environment). However, all was not yet Hunky Dory. When I next paid attention to the console of my laptop, it turned out that the build had stopped, claiming that it could not find "/dev/rd/0". I searched my memory and started remembering two things: firstly, The Rock build requires that ram disks are configured, and secondly, The Rock is geared heavily towards Richard Gooch's devfs (a kernel patch which makes /dev a virtual file system that automatically contains device files for the devices you have). My build system supported neither, and the build could therefore not complete. I researched the final build stages a bit, and it turned out that after the build has compiled all the packages it starts generating the install CD-ROM. It uses ram disks in this process to create boot floppies.

I downloaded the devfs, patched the kernel, reconfigured it to support ram disks, and rebooted the system. Everything came up OK, and I restarted the build. An enormous amount of errors related to glibc 2.1.2 flew over my screen. I interrupted the build and tried to analyse the problem. As far as I could see, the build could not easily be restarted straight into the final phase, and it tried to rebuild the packages from scratch instead. Since the chroot()-ed Rock Linux environment already existed, I got errors such as "Changelog.gz already exists". The restart had however already wrecked a number of things, and the only thing I could think up was to restart the entire build once more.

The Rock 1.3.5

However, the events described above had taken about one week of clock time, and when I browsed the Rock Linux web site (http://www.rocklinux.org) it turned out that Clifford had released another Rock Linux version: 1.3.5. Without much thought, I decided to go with 1.3.5. From the FTP site, I downloaded the 1.3.5 source, analysed the package differences (diff rules!), and used my getpkg.pl Perl script to download the new or changed packages in the new release. By now, I was becoming a Rock Linux build expert, so configuring and starting the build was a breeze.

The Rock 1.3.6-DEV-1999111912

Due to external schedule pressures, I was unable to continue working on building The Rock for a week or so. When I finally got round to it the next weekend I immediately ran into a small problem concerning the compilation of the hwclock program (something concering "outb()"). Since a week had passed, I first checked the Rock Linux web site, and, lo and behold, a development snapshot of 1.3.6 had magically appeared! Since staying on the bleeding edge suited me fine, I downloaded the new snapshot, fired up getpkg.pl to download the new and changed packages, and started the build again.

Again, I ran into some small problems which were easily fixed. For instance the build of the shadow package failed again (remember the aclocal.m4 fix I described earlier). By now I understood enough of aclocal and automake to solve this problem structurally by installing GNU gettext and GNU libtool on my build system; this solved the problem by allowing aclocal to regenerate aclocal.m4 from configure.in. With these little improvements, the build progressed nicely, and into the stage where the boot disks and the install CD-ROM is generated. There, another problem popped up.

What the build procedure does is creating a 1.44MB ram disk, and setting it up as a boot floppy: formatting it, mounting it, copying files to it, the lot. When it is completely laid out, the ram disk gets dd'ed to a file which eventually finds its way onto the install CD-ROM image. By the way, to create a bootable CD-ROM you need a 720KB, 1.44MB or 2.88MB boot floppy image and specify that as the El-Torito boot image when running mkisofs. (By the way, if you have enough disk space and partitions, you can install Rock Linux from one hard disk partition to another, skipping the CD-ROM step altogether).

In this build, the build procedure ran out of space on the boot floppy ram disk while copying files into it. I wondered how that could happen. Surely, Clifford would have tried a complete build before putting out the release? On the other hand, I was building a development snapshot, so I guessed all bets were off. I did some more analysis of the build procedure and modified it so as to remove the lost+found directory on the boot floppy and to exclude the scsi kernel modules. Due to my better understanding of the build, I could now restart the final stage of the process manually. After a while (and plenty of toying around in what ended up on the boot floppies), the build finished. I was now the proud owner of a Rock Linux install CD-ROM image, which I subsequently burnt onto a blank CD-R disk.

Installing the Rock

The next challenge was to install The Rock from this CD-ROM. With shaking hands and my heart banging somewhere in my throat I slid the CD-ROM into the drive of one of my systems, and, ... the boot failed :-(For some reason or another, LILO failed to read the initial ramdisk (initrd) that contains the installation stuff. I looked around a bit, changed the image, burnt it on CD-ROM again, but all to no avail.

Allright, time for plan B. Using the install floppy images on the CD-ROM I created a set of boot diskettes and tried to boot from them. Aha, more success there: The Rock install shell came up and the installation process could begin. Now, if there's one thing I really like about Rock Linux it is its

install procedure. No fancy GUI, not even a fancy TUI, the Rock Linux install process dumps you in a shell, and leaves it up to you to use tools like fdisk, mke2fs and mount to create and mount the hard disk partitions where you want to install The Rock. For an experienced command line hacker like myself, this is the preferred way of working by far! After having created the necessary partitions, you execute the Rock install program which basically unloads a bunch of bzip2'ed tarballs onto the install partitions. Once finished, you can then manually generate a new kernel (the chroot command comes in handy here) and run LILO.

Booting the Rock

The install was a breeze, and in no-time I rebooted my new Rock Linux system for the first time. However, in my youthful enthousiasm, I made a small error in building my new system's kernel: I had inadvertently de-configured the IDE disk driver! As you'll understand, the Linux kernel finds it pretty difficult to operate /dev/hda when the IDE driver is absent. Fortunately, the Rock Linux install disk can act as a rescue disk as well. I used to boot floppies to boot a shell, mounted in the hard disk partitions containing Rock Linux, chroot-ed to the root file system on the hard disk, and generated a new kernel (not forgetting to run LILO afterwards :-).

The next boot went a bit more successful. Just a little bit, though. The kernel came up quite nicely, but could not mount the root file system on the hard disk. As I mentioned before, Rock Linux comes with the devfs integrated, and after some research it turned out that the device file name of the partition containing the root file system was incorrect. Seemingly, the devfs naming scheme for disks had recently changed, and the /etc/lilo.conf shipped with this Rock Linux development snapshot still contained the old name. After another rescue operation this problem was fixed as well, and Rock Linux came up like a charm!

Finally

So, my goal has been achieved. And, although I did not consider myself unknowing to start with, I learnt a lot about Linux along the way. Rock Linux is positioned for experienced Linux administrators, and rightly so! Downloading, building and installing Rock Linux (especially the snapshots) is not for the faint hearted. However, with unlimited control comes unlimited power. And if you want complete control over a new Linux server, and do not want to install unneeded packages, or want to change the options which with certain packages are built, then Rock Linux is for you!

In the time between the events described here and the writing of this article, Clifford and his team put out Rock Linux 1.3.7 and a bunch of 1.3.8 development snapshots. However, rest assured that you probably need a lot of Unix/Linux knowledge and experience to build and install these as well.

Rock Linux: Now you're playing with power!

See also:

- ROCK Linux Philosophy by Clifford Wolf.
- ROCK Linux Homepage.

Other articles by Jos Visser:

- Kernel based random number generation in HP-UX 11.00
- Comal has been set free
- Unix failover clusters - If you've seen one, you've seen 'm all
- Interessante ontwikkelingen in Linux

E-mail: Jos Visser

[Index]

For more information write e-zine@e-zine.nluug.nl