
Chapter 1. Bootstrapping The System

Table of Contents

1.1. What Is Bootstrapping?	1
1.2. Installing A Development System	1
1.3. Installing Basic System Packages	1
1.4. Booting The New System	2
1.4.1. Booting From Diskette	2
1.4.2. Exploring Functionality	3
1.5. Installing Basic Development Tools	4
1.6. Building A Kernel On The New System	5
1.7. Building Additional Development Tools	6
1.8. Making The System More Friendly	7
1.9. Taking Care Of Package Dependencies	9
1.10. Installing The Remaining Packages	9
1.11. Finalizing System Configuration	10

1.1. What Is Bootstrapping?

1.2. Installing A Development System

1. Find a PC that can be dedicated to the purpose of bootstrapping a system. Do not try to do this on a dual-boot system or any other production system, because it *will* get broken. If only one PC is available consider using a removable hard drive tray system to fully remove any valuable data before proceeding. (See Vantec's site [<http://www.vantecusa.com/product-storage.html>] for an example of what is available.)
2. Select a modern GNU/Linux distribution and install it onto a blank hard disk. Leave about 2G of unpartitioned disk space to be used for the bootstrap system.

1.3. Installing Basic System Packages

1. Use architect to build the following binary packages:

- bash-static
- coreutils
- e2fsprogs
- ed
- glibc

- gzip
- mdadm (optional for software RAID)
- modutils
- procps
- sysvinit
- tar
- util-linux

2. Create a second extended filesystem of about 2G in size and mount it on `/mnt`.

The following example show this task being done using `/dev/hdb2` as the partition for the bootstrap system. Make changes as needed when using a partition that is different than the example.

```
dev-system# mke2fs /dev/hdb2
dev-system# mount /dev/hdb2 /mnt
```

3. Create an FHS compliant directory structure rooted in `/mnt`. (See `architect/extra/install-tools/mkfhs`)
4. Extract all of the recently created binary packages onto the new filesystem mounted on `/mnt`.
5. Create device nodes in `/mnt/dev`. (See `architect/extra/system-scripts/dev/MAKEDEV`)

Not all of the packages above are absolutely essential for system operation. In fact, most of the work is done by `bash-static`, `coreutils` and `util-linux` with `glibc` providing the libraries. The other packages are mainly for convenience. For example, packages like `mdadm` and `modutils`, although not essential, are installed to aid those who would like to use kernel features like software RAID and loadable modules. Other packages, like `ed`, `gzip`, `tar` and `sysvinit` will be used in upcoming stages but are installed now in order to cut down the number of steps in the bootstrap installation.

1.4. Booting The New System

1.4.1. Booting From Diskette

1. Use `architect` to build `grub` (or `grub-serial0`)
2. Construct a GRUB bootdisk.
3. Boot the system manually using the `grub` command-line.

For a serial console system the commands would be similar to the following example:

```
grub> kernel (fd0)/boot/vmlinuz console=ttyS0,9600 root=/dev/hdb2 init=/bin/sh
grub> boot
```

Be sure to change the `root=` parameter to fit your system setup and leave out the `console=ttyS0,9600` for non-serial console systems.

1.4.2. Exploring Functionality

1. Verify that the system comes up to a # prompt.
2. Set a proper PATH variable.

Some commands will only be accessible using a fully qualified path unless the environmental variable PATH is set to a reasonable value. Since the bootstrap system is in superuser mode it is good to include all of the various `sbin` directories as well as the `bin` directories as shown in the example below.

```
sh# export PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin
```

3. Test basic system commands.
4. Manually remount the root filesystem.

The example below shows how to do this for a root filesystem that is on `/dev/hdb2`. Adjust accordingly if the root filesystem is on a different device.

```
# mount -n -o remount,rw /dev/hdb2 /
# mount -f -t ext2 /dev/hdb2 /
```

Disregard any warning messages about missing `fstab` or `mtab` files.

5. Create an `/etc/fstab` file for next time.

Assuming the root filesystem is on `/dev/hdb2` as in previous examples a simple `fstab` may be created using the `cat` command.

```
# cat <<EOF >/etc/fstab
> /dev/hdb2      /          ext2      defaults    1          1
> proc          /proc      proc      defaults    0          0
> EOF
```

Test by mounting `proc` using only the mount-point as an argument to the `mount` command (i.e. `mount /proc`).

6. Unmount filesystems using the `local_fs` script.

Navigate to the `/etc/init.d` directory and run `local_fs` with the `stop` command-line argument as shown below.

```
# cd /etc/init.d
# ./local_fs stop
```

Note

If the shell complains about "permission denied" make sure the scripts have the execute permission set. `chmod 750` will do the trick.

7. Reboot the PC and go back the development system.

For non-serial console systems this can be done with the **CTRL-ALT-DELETE** sequence. Systems using a serial console will need a hard reset since the three finger salute will reset the system running the terminal emulator rather than the intended host. Remember to remove the boot diskette.

1.5. Installing Basic Development Tools

1. Boot into the development system.
2. Use architect to build the following binary packages:

- binutils
- bison
- diffutils
- findutils
- flex
- gawk
- gcc
- grep
- m4
- make
- patch
- sed

3. Copy the binary packages to the bootstrap system.

Mount the bootstrap filesystem on `/mnt`, create a directory called `/mnt/var/spool/packages/` and copy the new packages into it.

4. Reboot into the bootstrap system.
5. Extract all of the recently created binary packages onto the bootstrap system.

6. Test the compiler and linker.

Write a simple hello world program like the one shown below. This can be accomplished using the **ed** editor or the old cat <<EOF trick. Save the file as `hello.c`.

```
#include <stdio.h>

int
main () {
    printf("Hello World!\n");
}
```

Build a binary using `gcc` and run it. Check for any compilation or runtime errors. The example below demonstrates a successful test.

```
sh# gcc -o hello hello.c
sh# ./hello
Hello World!
```

1.6. Building A Kernel On The New System

1. Download the latest tar.gz kernel source package onto the development system.

This first step is only necessary because the bootstrap system cannot download files from the Internet. Once the source is downloaded to the development system it is easy to copy it over to the bootstrap system.

2. Start up the bootstrap system.

Use the GRUB boot diskette with the same procedure as when the system was booted for the first time. When a shell prompt appears remount the root filesystem as read-write and mount the `proc` filesystem.

3. Check the time and timezone setting on the bootstrap system.

The following example shows how to set the proper time zone for someone living in Chicago, IL USA (central time zone).

```
bash# cd /usr/share/zoneinfo/US
bash# cp Central /etc/localtime
```

Once the time zone is set check the time with the `date` command. If the PC's hardware clock is set to local time, rather than Greenwich Mean the time will be wrong. In this case it is necessary to update the kernel clock to match the hardware clock as show in the following example.

```
bash# date
<wrong time shown>
bash# /sbin/hwclock --hctosys
bash# date
<correct time shown>
```

If the time is not fixed, there is a chance that the `make` commands we use later could get confused since `make` uses a file's timestamp to see what is up-to-date and what needs to be built.

4. Copy the kernel source tarball to the bootstrap system's `/usr/src` directory.

This can be accomplished in a few different ways, but the most straightforward may be to mount the development system's root filesystem on `/mnt` and copy the source tarball from there. It is also possible to put the source on a CD-ROM or Zip disk. Those looking for an extra challenge may want to try using `tfcp` to fetch the

source from another PC on the local network. This requires a tftp server on your network, as well as building the net-tools and netkit-tftp packages and setting up `/etc/protocols` and `/etc/services` on the bootstrap system.

5. Unpack the code and create a symbolic link called `linux` that points to the kernel source code directory.

The following example demonstrates what the process looks like on the bootstrap system.

```
bash# cd /usr/src
bash# ls
linux-2.4.27.tar.gz
bash# tar -zxf linux-2.4.27.tar.gz
bash# ls
linux-2.4.27/  linux-2.4.27.tar.gz
bash# ln -s linux-2.4.27 linux
bash# ls
linux@  linux-2.4.27/  linux-2.4.27.tar.gz
```

6. Change directory to `/usr/src/linux` and configure the kernel source.

Since no ncurses library is loaded on the bootstrap system the only option for configuring the kernel source is to use **make config**. Anyone who has ever tried this knows that the **make config** method of kernel configuration can be slow and very unforgiving of mistakes. But, if things go well it only has to be done this way once.

7. Run **make dep** on the kernel source.

Many source packages depend upon parts of the kernel source in order to build correctly. A lot of this has to do with the symbolic links `/usr/include/asm` and `/usr/include/linux` pointing to directories within the Linux source code. So even if a new kernel is not needed it is still necessary to do **make dep** on the kernel source.

8. Run **make bzImage** to create a new kernel.

Those who are happy with the kernel on the boot disk may be tempted to skip this step. However building a kernel is a good way to test the development tools on the bootstrap system and shake out any bugs. If the kernel build is successful there is a good chance that compiling other packages will be successful as well. If there are problems with the kernel build the solution should be investigated and resolved before moving on to build other packages.

1.7. Building Additional Development Tools

There are several packages that need to be built because other packages depend upon them. For example, BASH cannot be built without having the ncurses library installed first and postfix will not work without db.

1. Fetch the source code for the db, gettext, ncurses, texinfo and zlib packages.

This can be done in several ways just like the kernel source code. Options include mounting the development system's hard drive on `/mnt` of the bootstrap system, burning a CD-ROM, copying to Zip disk or using tftp.

2. Unpack the source code tarballs in `/usr/src`.
3. Install the latest GNU/Linux Architect Toolkit.

The latest Architect Toolkit is available from SourceForge. Architect should be unpacked into the `/usr/local` hierarchy and a symbolic link made for `/usr/local/bin/architect`. The example below shows how to do this via tftp.

```
sh# mkdir /usr/local
sh# cd /usr/local
sh# tftp
tftp> connect 192.168.1.1
tftp> get architect-0.10a.tar.gz
tftp> quit
sh# tar -zxf architect-0.10a.tar.gz
sh# mkdir /usr/local/bin
sh# cd /usr/local/bin
sh# ln -s ../architect/bin/architect architect
```

4. Use architect to build the gettext, ncurses and zlib library packages.

The order in which these packages are built is not important. However, there is one thing to watch out for. Do not attempt to build the ncurses-with-fallbacks package until the regular ncurses package has been built and installed. The reason is that using the `--with-fallbacks` option in the configure script of ncurses-with-fallbacks requires that the terminfo database be present. This creates a paradox since there will not be a terminfo database until ncurses is installed. To use fallbacks build and install regular ncurses first then build and install ncurses-with-fallbacks.

5. Install the new packages on the bootstrap system.

Assuming the bootstrap system is currently booted and active the following example shows how to quickly install the packages built in the previous step.

```
sh# cd /var/tmp/staging
sh# ls *.i386.tar.gz
gettext-0.14.1.i386.tar.gz  ncurses-5.4.i386.tar.gz  zlib-1.2.1.i386.tar.gz
sh# for PKG in *.i386.tar.gz; do echo "Installing $PKG"; tar -C / -zxf $PKG; done
Installing gettext-0.14.1
Installing ncurses-5.4.i386.tar.gz
Installing zlib-1.2.1.i386.tar.gz
```

1.8. Making The System More Friendly

Most people will probably want to install BASH to take advantage of features, like command completion and command-line history, that are not present in bash-mini. The vi editor from the Elvis package might be high on the list of cool tools as well. This will be no problem since ncurses was installed in the previous bootstrap phase. And finally it would be great to automate many of the system startup and shutdown tasks.

1. Fetch the source code for the bash and elvis packages.

The example below shows source code being downloaded from a tftp server on the local network. Fetching from disk or CD-ROM is even easier.

```
sh# cd /usr/src
sh# tftp
tftp> connect 192.168.1.1
tftp> get bash-3.0.tar.gz
Received 2436918 bytes in 1.6 seconds
tftp> get elvis-2.2_0.tar.gz
Received 1450266 bytes in 1.1 seconds
tftp> quit
sh# tar -zxf bash-3.0.tar.gz
sh# tar -zxf elvis-2.2_0.tar.gz
```

2. Build and install both bash and elvis on the bootstrap system.

Typical installation is shown in the following example.

```
sh# cd /var/tmp/staging
sh# tar -C / -zxf bash-3.0.i386.tar.gz
sh# tar -C / -zxf elvis-2.2_0.i386.tar.gz
```

3. Use and enjoy BASH.

Start up bash by typing its name on the command-line. If the PATH variable is not set correctly it may be necessary to give the full path to the bash binary as shown in the example below.

```
sh# bash
sh: bash: command not found
sh# /usr/bin/bash
bash-3.00#
```

4. Create an /etc/profile login script using vi.

A simple /etc/profile script will take care of mundane tasks like remembering to set the PATH variable each time. A simple example is shown below.

```
# /etc/profile - system login script

if [ $UID -eq 0 ]; then
    PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin
else
    PATH=/usr/bin:/bin:/usr/local/bin
fi

case $0 in
    -sh|sh)
        if [ $UID -eq 0 ]; then
            PS1='# '
        else
            PS1='$ '
        fi
        ;;
    -bash|bash)
        PS1='\h:\w\$ '
        alias ls='ls -F'
        ;;
esac
```

This script also found in the /usr/local/architect/extra/config-files/etc/ directory.

5. Restart BASH to make /etc/profile take effect.

Using the --login option when invoking bash will cause it to read and execute the contents of /etc/profile as the following example demonstrates.

```
bash-3.00# exit
sh# /usr/bin/bash --login
(none):/#
```

6. Create /etc/passwd and /etc/group.
7. Create additional init.d scripts to automate startup tasks.
8. Write an /etc/inittab file and make rc.d symlinks.

There is no networking support yet so only runlevels S, 0, 1, 2 and 6 are of any concern.

1.9. Taking Care Of Package Dependencies

Some packages depend upon other packages being present in order to build correctly. For example the at daemon will send an email when an at job finishes so it needs to have sendmail (postfix) installed in order to work.

- Fetch the source code for the following packages.
 - groff
 - less
 - postfix

1.10. Installing The Remaining Packages

1. Build the packages listed below.

The remaining packages required for a full-featured system are listed below. These can be built in any order.

- bc
- cpio
- dcron
- file
- iptables
- LPRng
- man
- netkit-base
- netkit-ftp
- netkit-telnet
- netkit-tftp
- net-tools

- psmisc
- rsync
- shadow
- sysklogd
- time

Some of the packages in the list contain commands that are required by the Linux Standard Base (LSB) specification.

2. Install the packages onto the running bootstrap system.
3. Test the new packages.

1.11. Finalizing System Configuration

1. Add support for runlevel 3.
Create directory, symlinks
2. Create configuration files for basic network connectivity.
IP address, route, firewall
3. Configure network services.
Enable telnet, rsync