NFS Root

Introduction

This HOWTO is about running your Slackware Linux system without any hard disk - or perhaps with a very small hard disk - accessing the network to retrieve all files except the kernel. If you want to go the whole hog you can PXE-boot the kernel too, however this HOWTO expects you to have somewhere local to store the kernel. We're going to use VirtualBox virtual machines to **simulate** a diskless client. We'll still use a full Linux install along with our trusty LILO to prepare and then boot that kernel. This may seem a bit pointless, as there is no use-case for exactly what I'm doing here, but it should at least educate as to how you go about getting NFS root, and some of the pitfalls and workarounds.

Running an NFS server

Of course you need an NFS server to serve the root files from. I'm going to call this **slack-nfs-server**. The IP address is **172.17.0.80**. There is a a great article on setting up an NFS server. Following that, create your server using Virtual Box (with bridged networking), and then come back here for the next step.

The /etc/exports I'm using looks like this:

```
/nfs_share 172.17.0.1/24(rw,sync,no_root_squash,no_subtree_check)
```

That's insecure but good enough for the setup, we can refine it later. In addition to that guide it's worth doing:

```
chmod 777 /nfs_share
```

In order that we can be sure we can write to the root of our NFS share once it's mounted.

Creating the rootfs

Now create another virtual machine to do the install (bridged networking again). Mount the Slackware install disk on this virtual machine. If you want a 32-bit rootfs, obviously use the 32-bit install DVD, if you want a 64-bit rootfs, use Slackware64. If you do want 32-bit (quite likely for a diskless client) you may need to enable PAE in VirtualBox under System → Processor to get Slack32 to boot.

Assuming you now have the Slackware installer booted using the default huge kernel you can now start the install. Login as root, and get your IP address:

```
# dhcpcd eth0
```

Mount the nfs share you previously created on /mnt:

```
# mount -o rw,nolock slack-nfs-server:/nfs_server /mnt
```

If you don't have DNS setup that's fine, just substitute the IP address of your server for slack-nfs-server, i.e. 172.17.0.80.

Now we need to fudge something for the Slackware installer. Using fdisk, create a single partition on your hard disk (/dev/sda1). Unfortunately a valid partition on a drive connected to the system is a requirement for the installer to run. Don't worry, we'll just delete this entire virtual machine when we're done so it's no biggie.

Now run 'setup'.

- Remap keyboard as you want
- Don't configure any swap.
- At the screen 'Select Linux installation partition' don't select /dev/sda1 just down-arrow and select '(done adding partitions, continue with setup)'.
- Install from the slackware cd/dvd
- Select your packages as normal
- Don't create a bootdisk
- Don't install LILO.

When done you should now find a root fs 'installed' to the server directory. If you plan to use it with more than one thin client, then now would be a good time to take a backup copy before the first boot into it.

Creating the kernel

The full huge kernel that comes with Slackware 14.2 is close to providing everything we need, but we still need to recompile it. I'd recommend doing the compilation on a 32-bit virtual machine if you are targeting a 32-bit thin client, or 64-bit if your thin client is 64-bit. There are ways to avoid this and cross-compile kernels 32→64 bit and visa versa but virtual machines are cheap and life is short:

```
# cd /usr/src/linux
# zcat /proc/config.gz > .config
# make menuconfig
```

Configuration order is important, as selecting certain options makes others available. First off we will need a network driver compiled into the kernel for the NIC we're going to use. For VirtualBox the default NIC is PCnet32, an lspci will probably tell you yours:

```
Device Drivers -> Network Device Support -> Ethernet driver support -> AMD
PCnet32 PCI support <*>
```

Make sure this is compiled into the kernel (e.g. hitting 'y').

[OPTIONAL] We also need to tell the kernel which IP address to use, which can be set statically, but DHCP is much easier, so generally you will want to include these options:

```
Networking support -> Networking options -> IP: kernel level
autoconfiguration [*]
    IP: DHCP support [*]
```

Finally, we absolutely need the support for Root FS on NFS:

```
File Systems -> Network File systems -> Root file system on NFS [*]
```

[OPTIONAL] It's pretty useful to append a local version to this kernel release. I'd advise doing this to differentiate it from your standard Slackware huge kernel and avoid clobbering the modules from that by mistake. We can just add '-nfsroot':

```
(-nfsroot) General Setup -> Local version - append to kernel release
```

Save the configuration and then do a:

```
# make bzImage
```

While that build is running, it's time to configure LILO.

Configuring LILO

Let's call the kernel /boot/vmlinuz-nfsroot. Add a section to the lilo.conf file:

```
image=/boot/vmlinuz-nfsroot
  label = nfs
  read-only
  append= "root=/dev/nfs ip=dhcp nfsroot=172.17.0.80:/nfs_share,v3 rw"
```

If you didn't want to use dhcp you'll now need to have a read of Documentation/filesystems/nfs/nfsroot.txt in the kernel sources to figure out the many options that you can include for ip= other than 'dhcp'.

Obviously keep your default linux kernel in another image= section so you can switch between booting the nfsroot and the normal kernel to play around with this stuff.

You cannot specify a normal root= entry in this section because LILO doesn't recognise /dev/nfs for root (the device doesn't actually exist to LILO). So instead just specify it in the append= line which LILO doesn't try to interpret, and LILO will include this extra nfsroot image without error.

The v3 seems to be really important in making anything at all happen on boot. If that isn't set, no communication seems to occur.

The 'rw' is also important. It prevents the fsck of the root fs. because root is NFS and can't be checked. Slackware won't boot properly if we give 'ro'. Instead of using 'rw' you could optionally hack fsck out of the Slackware startup scripts on your NFS root, however simply using 'rw' is quicker (albeit dirtier).

With the kernel compilation finished, copy the kernel into the /boot directory and rename it:

```
cp /usr/src/linux/arch/x86/boot/bzImage /boot/vmlinuz-nfsroot
```

It may be created elsewhere than arch/x86 depending on your architecture, e.g. x64, arm.

Don't forget to run LILO:

```
# lilo
```

First Boot

The above is enough to get you a booting Slackware system, or should be. There are some additional steps that you may wish to now do.

Modules

None of the modules have been installed, let's add them. Shutting down the nfsroot system and booting back into the Slackware kernel compilation virtual machine we can now compile the missing modules. First we will mount the rootfs, just as we did from the installer virtual machine:

```
mount -o rw,nolock slack-nfs-server:/nfs_share /mnt/tmp
```

Then we can compile and install the modules:

```
# cd /usr/src/linux
# make modules
# make modules_install INSTALL_MOD_PATH=/mnt/tmp
```

For the last command, avoid adding a trailing slash to /mnt/tmp, and try not to forget the INSTALL_MOD_PATH, otherwise you may have just clobbered your system modules. If you gave your kernel a local suffix (e.g. -nfsroot) you'd have been protected against that.

Swap on NFS

You can create a swap file on your NFS share somewhere like this:

```
# dd if=/dev/zero of=/nfs_share/swapfile bs=1024 count=64k
```

Then format it for swap:

```
# mkswap /nfs_share/swapfile
```

Then on the client you associate a loopback device with the file:

```
# losetup /dev/loop0 /swapfile
```

Then start using the loopback device for swap:

```
# swapon /dev/loop0
```

Obviously you need to add the last two commands to /etc/rc.d/rc.local or another startup script to run

on each boot.

Locking down /etc/exports

Assuming your thin client connects from a predictable address, now that we've installed the modules we can finally lock down access to only the thin client (/etc/exports on the server):

```
/nfs_share 172.17.0.81/32(rw,sync,no_root_squash,no_subtree_check)
```

And we presumably don't want all-and-sundry using our newly prepared rootfs directory, so drop it down a level and qualify it by IP address (on the server):

```
# cd /
# mv nfs_share 172.17.0.81
# mkdir nfs_share
# mv 172.17.0.81 nfs_share
```

Now over on the client machine, configure LILO so nfsroot requests the nfs share based on the client's IP address with '%s':

```
image=/boot/vmlinuz-nfsroot
  label = nfs
  read-only
  append= "root=/dev/nfs ip=dhcp nfsroot=172.17.0.80:/nfs_share/%s,v3 rw"
```

NFS Root is never going to be considered secure, but at least this makes cross-contamination of nfsroots less likely.

Note that I am using dhcp in the above example, but I've added an entry to /etc/dnsmasq.conf on my router mapping the thin client MAC address to the IP address 172.17.0.81 so the client always gets that address.

Sources

- Originally written by User bifferos

howtos, nfs, author bifferos