Root Raid in Ram How To

Mehdi Bozzo-Rey[a], Michel Barrette[b], Benoît des Ligneris[c], Francis Giraldeau[d]
Centre de Calcul Scientifique, Université de Sherbrooke, Québec, Canada

[a]mbozzore@physique.usherb.ca

[b]mike@ccs.usherbrooke.ca

[c]benoit@des.ligneris.net

[d]giraldeau@hermes.usherb.ca

Sherbrooke University is involved in the development and support of diskless clusters. The approach use a regular Linux system with a minimal ramdisk as its root device. In order to support different kind of kernel without fancy compilation options we used the software RAID capacity of the kernel to mount an array of RAID-0 RAM disks, before the actual loading of the operating system. Once the real root file system is mounted, the loading of the operating system can then be done in numerous way (TFTP, wget, NFS, ssh, ...) and the diskless node can operate even when the network is down or the so called master-server is unavailable. We will present in this article all the necessary steps to build your own ramdisk based diskless or systemless node.

L'Université de Sherbrooke est impliquée dans le développement et le support de grappes dont les noeuds de calcul ne possèdent pas de disque rigide (grappe sans disques). Ce type de grappe utilise un système d'exploitation Linux standard avec un système de fichiers minimal résidant en mémoire vive. Nous telechargeons ce système via différents protocoles de communication (Trivial File Transfer Protocol (TFTP), Network File System (NFS), ssh, ...), afin d'assurer à chaque noeud de calcul une plus grande autonomie en cas de panne réseau ou d'inaccessibilité du serveur maître. Afin d'assurer la compatibilité avec différents noyaux Linux standards, nous utilisons la capacité du noyau Linux à gérer les réseaux redondants de disques indépendants (RAID, Redundant Array of Independant Disks) pour monter un réseau RAID-0 de disques virtuels en mémoire vive, afin d'y installer le système d'opération.Nous présentons dans cet article les différentes étapes nécessaires à l'élaboration d'un noeud de calcul sans disque. Nous présentons dans cet article les différentes étapes nécessaires à l'élaboration d'un noeud de calcul sans disque.

Introduction

By using a minimal Linux system loaded in RAM, we can achieve a much better scaling than « traditional »root-NFS based clusters [1]. With this minimal system in RAM, nodes do not depend on the network for their general operations. NFS is still used for general applications so that they are served on demand from a read-only partition which reduce the NFS traffic.

We will examine in details the different steps necessary for the support of a root file system in RAM. We will also specify some interesting points relevant to the support for RAM disk in the kernel, how to create from scratch the initial ramdisk, propose some way of obtaining the ramdisk we created from a disk (for testing) and then from the network. Finally, we will show how to support generic kernel (with small 4Mb ramdisk) by mounting a software RAID-0 array of ramdisk.

1 Kernel recompilation

It is not the goal of this document to explain in great details how to compile a kernel. Some very pertinent information already exists [2]. Please refer to it if something goes wrong.

1.1 Kernel compilation 101

- download : `http://www.kernel.org/`
- configuration interface : **make xconfig**
- **make dep** ; **make bzImage**, and if you enabled the loadable module support **make modules**, **make modules_install**

1.2 Necessary options

If you want to support modules you can do it. We will see how to support modules in our initial ramdisk (NIC driver, RAID, file-system, ...) but it is easier to start with a monolithic kernel.

It is necessary to select the initial ramdisk support. We will then be able to specify it size (4Mb by default). Support for the loopback device is also necessary.

1.3 Essential documentation

If the kernel sources are symlinked (or directly installed) in `/usr/src` then some files are very interesting and you should refer to them for up to date documentation (especially if you use another kernel version : kernel evolution can be very fast and some adjustments to the instruction given in this article can be necessary).

- `/usr/src/linux/Documentation/devices.txt` This is the complete list of defined devices with corresponding minor and major numbers (in particular, `/dev/ram` and Myrinet devices).
- `/usr/src/linux/Documentation/initrd.txt` Complete description for the initial ramdisk support.

– `/usr/src/linux/Documentation/ramdisk.txt`
How ramdisk support is working.

2 Initial ramdisk creation

The initial ramdisk is a minimal installation of Linux that has to be used for several reasons (one of them is the installation of modules necessary for the normal operation of a regular Linux box : NIC, file-system, RAID, ...). As such, it has to include a shell, some utilities as well as all the libraries used by the binaries you want to use in this minimal system.

We will not detail how to make a suitable system for embedded systems [3] and how to use some minimal library [4] to reduce the memory footprint. In modern computers, RAM is not so expensive and while we will use generic techniques to obtain a small ramdisk, we wanted to be fully compatible with all existing applications (Open Source or commercial ones).

2.1 Which files to include ?

One of the great virtue of open source software is the availability of others work. We will first examine a regular `initrd` from your preferred Linux distribution. We made our experimentation on Mandrake 8.2 but it will work on any distributions that support ramdisk (it has to be selected explicitly for Debian and you will certainly have a warning if you never did that before).

The generic location of the initial ramdisk is `/boot`. In this directory you will find files with name starting with `initrd` and those are the actual initial ramdisks. For Mandrake 8.2 it is `initrd-2.4.18-6mdk.img` and we will use this name as reference for our example of operation.

The kernel can uncompress ramdisks on the fly. Ramdisks are generally a compressed image of a Linux filesystem so that transfer time is minimized. In order to actually browse through this image and then modify it, add .gz extension : **mv initrd-2.4.18-6mdk.img initrd-2.4.18-6mdk.img.gz**, uncompress it with a **gunzip initrd-2.4.18-6mdk.img.gz**, create a `/mnt/initrd1/` directory and then mount this image using the loopback device with the following command : **mount -t ext2 -o loop /tmp/initrd-2.4.18-6mdk.img /mnt/initrd1/**.

Then, if you go into the `/mnt/initrd1/` directory, you will see a similar structure than the one presented figure 1.

```
- bin/
- dev/
- etc/
- lib/
- loopfs/
- proc/
- safedev/
- sysroot/
- sbin/ → bin/
```

Figure 1. Structure of an initial ramdisk

This is almost a « complete »functional Linux system. The `sbin/` directory point to the `bin/` one because at this early stage of the boot process system commands and user commands are the same : the initial ramdisk at this stage is in single user mode.

2.2 Which binary to install and how to find the linked libraries ?

As mentioned previously, we won't compete to make the smallest linux distribution but only examine which tools are mandatory.

The number of binaries will be kept small so that we don't end up with a huge initial ramdisk. We will choose only low-level binaries and sometime some binaries that are useful for the debugging of ramdisk experimentation. Once you have a working system, you can remove those debugging tools.

Binaries are usually linked against a certain number of libraries. This is a great feature that allows binary programs to be small and only include the functionality they have to provide. Binary programs can be compiled statically in which case they include the libraries. This is not generally done by distributions because it produce huge executables, and you don't want to have several copies of the same library on your hard drive, you want to optimize your space (as well as upgrade only one file when there is a library [security] update !). An interesting fact is that we are working the opposite way for an operational cluster, in this case we don't want to install the mathematical libraries on every node, but just on the « compilation node », users compile statically their application, so it can run in a calculation node. In our ramdisk image we will use the tools provided by the distribution and include the necessary libraries too.

In order to find which library is used by each binary, we will use the **ldd** tool. **ldd** prints shared library dependencies of the program given in argument. For instance **ldd /bin/cat** will produce an output similar to the one presented figure 2.

```
libc.so.6 => /lib/libc.so.6 (0x40028000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Figure 2. Example of ldd usage

In order to save you some time, we provide a complete list of the binaries that are necessary as well as the libraries dependencies (all this was done on a Mandrake system). The list of binaries is presented figure 3 and the corresponding libraries figure 4.

2.3 The linuxrc file

This script is the first one executed at boot. Don't forget to make it executable. Also, it may need to be added explicitly into your boot loader like this :

```
init=/linuxrc
```

- bash : interactive shell
- cat : useful to read /proc, /proc/mdstat for instance
- cp : to copy file (move files too with rm)
- gzip : compression / uncompression utility
- insmod : module loader
- ls : useful for debugging
- modprobe : automatic module loader
- mount : to mount a file system
- rm : to remove some files and directory
- tar : useful when retrieving any tape archive
- umount : unmount a file system
- pivot_root : change root

Figure 3. Typical binaries to include in the initial ramdisk, in the `bin/` directory

- libcom_err.so.2
- libc.so.6
- libdl.so.2
- libe2p.so.2
- libext2fs.so.2
- libpthread.so.0
- librt.so.1
- libtermcap.so.2
- libuuid.so.1

Figure 4. Library dependencies, typically in the `lib/` directory

This script has to be very simple because very few commands are available. One example is given figure 5.

The root is switched by the use of the **pivot_root** command. The old root (initrd-2.4.18-6mdk.img) is put into /initrd of the new root (tiny_system.img). As a consequence that the old root is contained into the new root.

This method is the simplest one. It is strongly recommended to use it to replace the old method that consist of setting a specific string into `/proc/sys/kernel/real-root-dev` because it may not be supported in future kernel release.

In the following section, we will describe more precisely how to create devices and where to found relevant documentation on this subject.

2.4 Building a tiny system image

The complete procedure is detailed in the kernel documentation [6]

This image will be run on the computer after the boot process : it will become the new root.

How to build it :

1. Creation of the root tree in a temporary directory
2. Create, format (ext2) and mount the initrd ramdisk
3. Creation of the necessary devices and copy of the necessary files

A script doing those operations is presented figure 6.

It is mandatory that the kernel support the « loopback block devices ». It is preferable to build first the structure of the ramdisk so that no space is lost when we create the file-system. Of course, if you use a compressed file system, this is not very important as the compression will reduce any unused space to a very small size because there is a repetitive structure of ”zeros” (don’t **dd /dev/urandom** in order to create your initial ramdisk or you will have some trouble !). The first initial ramdisk must fit into a 4Mb partition because, at this time, it is not possible to have more space unless you compiled a specific kernel.

```
#!/bin/bash

# Default Path
export PATH=/bin:/sbin

# Mount the /proc file-system
mount -t proc /proc /proc

# Mount the disk (read-only) to retrieve
mount -t ext3 -o ro -o nocheck /dev/hda1 /mnt/hda1

# Copy the model in ram, uncompress on the fly
cat /mnt/hda1/tiny_system.img.gz | gzip -dc > /dev/ram3

# Unmount the disk
umount /mnt/hda1

# Mount the new root
mount -t ext2 -o ro -o nocheck /dev/ram3 /mnt

# let's unmount /proc and change the root
# The old root is placed into /initrd of the new root
umount /proc && cd /mnt && pivot_root . initrd

# Get rid of the original console device file still
# in the initrd
exec </dev/console >/dev/console 2>/dev/console

# Pass ctrl to the normal init script of the new root
exec /sbin/init 3
```

Figure 5. `linuxrc` example.

```
# Make the {bin, etc, lib, dev, initrd, loopfs,
# proc, safedev, sysroot, sbin } dir
mkdir -p essai_initrd/bin [ ... ]

# Image creation and mount
dd if=/dev/zero of=image bs=1024k count=4
mke2fs -F -m0 image
mount -t ext2 -o loop image /mnt/toto
cd /mnt/toto/dev

# Make the devices
mknod [ ... ]
```

Figure 6. Script : How to make an image.

2.4.1 Creation of devices

Devices are special files with a definition given by the kernel. We will use the user-land tool available to create the devices : **mknod**. The command has numerous options, please read its **man** page for more information. Basically it has the following syntax : **mknod name type major minor** where the type can be **b** for block device, **c**, **u** for a character device and **p** for a FIFO relay.

For instance, in order to create the device corresponding to `ram0` we will use the following command : **mknod ram0 -m 660 b 1 0**. For more information about device type, major and minor numbers please consult the kernel documentation[7]. In most distributions, the **MAKEDEV** command is available. This script will create or delete devices in `/dev` so that you don't have to manually create each of them. Devices creation or deletion are grouped together (for instance std, ram, cpu, ...).

All those groups are defined in the `/etc/makedev.d` directory. Inside this directory, the `00macros` defines the permission for devices. All other files are definitions extracted from the kernel documentation that will create each group of devices with the correct permissions for a typical Linux system.

In our case, we are only interested in essential devices. The minimal list with a short description of each device is given figure 7.

- `console` : In order to have a display to the screen (5, char, 1).
- `tty#` : Virtual consoles. In order to have multiple screen on the same video output (4, char, #) and (5, char, 0) for the current TTY device.
- `md#` In order to use meta-disk (RAID) device (9, block, 0 or 1) for `md0` or `md1`.
- `null` The black-hole device.
- `ram#` Ramdisk. (1, block, #)

Figure 7. Devices that have to be present in the initial ramdisk (major number, type, minor number)

In order to have a functional system, some symbolic links have to be created : `ram`→`ram1`, `ramdisk`→`ram0`, `systty`→`tty0`.

In order to create all of those devices, we used a short script presented figure 8.

```
liste="0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15"
for i in $liste; do mknod ram$i -m 660 b 1 $i; done
mknod null -m 660 c 1 3
mknod tty -m 666 c 5 0
liste1="0 1 2 3 4"
for i in $liste1; do mknod tty$i -m 666 c 4 $i; done
mknod console -m 666 c 5 1
for i in $liste1; do mknod md$i -m 660 b 9 $i; done
```

Figure 8. Script : How to make devices.

Once this is done, we've got a functional minimal system and we can try to boot our system with it.

3 How to load a complete system image from a disk

We will first review several ways to do an image for a complete system and then, in order to test our work, examine how to load the image from a regular disk. In the next section, we will examine how to download the image from the network.

3.1 How to make a [diskless] image ?

In our first cluster called Elix, we made the images ourselves from a minimal installation of Linux. Now, this step is completely automated, thanks to the System Installation Suite[8] and the OSCAR[9] integration of this tool. The thin-OSCAR work-group[10] has been created to automate completely the server configuration for diskless nodes, the image creation and the transfer of the diskless image to the node.

3.2 Proof of concept

In order to transfer the image, we will just use the `linuxrc` file given figure 5. In this file, we mount the regular hard disk and then copy this minimal system in the ramdisk after the archive extraction. Then we change the « official »kernel root by running the command **pivot_root**.

The compression of the image is not strictly speaking necessary. We can either uncompress the image on the fly or use an uncompressed image. The last solution is the more interesting while debugging your system because it allows you to easily update your mounted image without never being out of sync from your uncompressed image.

4 How to transfer an image from the network

For the network part, we have two ways to proceed, including or not the necessary drivers in the kernel. We choose the second case : building a modular kernel with his modules collection ; we don't want to compile a collection of monolithic kernels for an heterogeneous cluster and as we have to use modules.

Once this first step is done, there are several ways to get the image, here is a non exhaustive list :

- PXE This can be used with regular NIC supporting PXE (very common nowadays) by including lines like those one **DEFAULT bzImage_2.5.30_nfs3** and then **APPEND initrd=thinimage.img.gz devfs=nomount root=/dev/ram0**.
- archive (**tar**), compression (**gzip**) of the repertory containing the image → transfer via **wget**, **tftp**, **scp** followed by **gunzip**
- **rsync** (with or without ssh) of the repertory containing the image
- creation of an ext2 file containing the image → transfer via **wget**, **tftp**, **scp** and **dd** on the fly on the device in RAM

After having tested different ways to create, modify, and transfer the image, we came to the conclusion that the most reliable way is the combination of ext2 file and **dd** on the fly. This technique is presented in the complete `linuxrc` file in figure 10.

5 Boot from the network and usage of a RAID-ram device

In this section, we will add a refinement and build a RAID device with some ram disks.

5.1 Creation of RAID disks in RAM

In order to create a RAID ramdisk we will use the **raidtools** package [11]. The definition of the RAID device is defined in the `/etc/raidtab` file. Don't forget that `/dev/ram0` is already used by the initrd, so you can't use it to build your raid array in `/linuxrc`. In order to create a RAID device in RAM with 4 ramdisks in it, please see figure 9.

```
raiddev       /dev/md0
raid-level    0
chunk-size    64k
persistent-superblock 1

nr-raid-disks 4
device    /dev/ram1
raid-disk 0
device    /dev/ram2
raid-disk 1
device    /dev/ram3
raid-disk 2
device    /dev/ram4
raid-disk 3
```

Figure 9. `/etc/raidtab` example : 4 ramdisk for one RAID array.

You can then manipulate your RAID array (in ram) exactly like a regular partition (once you created it). Some example of typical manipulation on your disk array :

- **mkraid /dev/md0** or **raid0run /dev/md0** : creation of the disk array
- **mke2fs /dev/md0** : Format the disk array
- **mount -t ext2 /dev/md0 /mnt/raidram** : Mount the disk array

In our case, their is no persistent super-block and the `/etc/raidtab` has to be altered by changing the fourth line for this one :

```
persistent-superblock 0
```

You can notice that we use in our scripts the command **raid0run /dev/md0**, which is the command to start up old (super-block-less) RAID0/LINEAR arrays. This is just a symlink to the binary **mkraid** in the raidtools package.

5.2 Integration of the necessary binaries, libraries and modules

As any tools that we want to include in our initial ramdisk, we have to include some binaries and the libraries that they use. We need the `raid0run`, as well as the additional libraries `libcrypto.so.0`, `libssl.so.0` and for instance, the network module `eepro100.o`. The additional files required by those new utilities are `/etc/raidtab` and the RAID device `/dev/md0`.

5.3 Module loading and RAID in ram technique integration

The complete `linuxrc` file is given figure 10.

6 Conclusion

We have exposed and tune all the Linux boot process so that we can control effectively the root device. In the first part, all the necessary steps to support large ram disks and transfer the initial image have been reviewed. In order to support kernel with default compilation options (16 disks of 4 Mb, and modules), we created a RAID file-system in RAM so that larger images (up to 64 Mb) can be loaded within this framework with any module necessary for the NIC. This work has been done for generic diskless support and will be used by the thin-OSCAR project to support effectively diskless clients for OSCAR without the need to recompile a new kernel. We have tested the R^3 How-to on several distributions (Mandrake 8.2, 9.0, Red-hat 7.1, Debian Woody and SID) and two architectures, IA32 and IA64 (for Raid Ram only).

Acknowledgments

We would like to thank HP Canada for the IA64 (Itanium2) they sent to us for testing.

References

1. NFS-Root mini-HOW-TO
   `http://www.tldp.org/HOWTO/mini/NFS-Root.html`
2. The Kernel How-to
   `http://www.tldp.org/HOWTO/Kernel-HOWTO.html`
3. Embedded Linux Consortium
   `http://www.embedded-linux.org/`
4. μlibc, a C library for embedded systems
   `http://www.uclibc.org/`
5. Software-RAID how-to
   `http://www.tldp.org/HOWTO/Software-RAID-HOWTO.html`
6. Kernel Documentation on initrd
   `/usr/src/linux/Documentation/initrd.txt`
7. Kernel Documentation on devices
   `/usr/src/linux/Documentation/devices.txt`
8. System Installation Suite
   `http://www.sisuite.org/`
9. Open Source Cluster Application Resource(OSCAR)
   `http ://oscar.sf.net`
10. Diskless support for OSCAR
    `http://thin-oscar.ccs.usherbrooke.ca`
11. raidtools download (documentation inside)
    `http://people.redhat.com/mingo/raidtools/`

```
#!/bin/bash

# Default Path
export PATH=/bin:/sbin

#Mounting the /proc file-system
mount -t proc /proc /proc

# Creation of the RAID ramdisk
insmod /lib/raid0.o
raid0run -a

# Network initialisation

#Loading appropriate module(s)
insmod /lib/sunrpc.o
insmod /lib/lockd.o
insmod /lib/nfs.o
insmod /lib/eepro100.o
echo "Step loading network module END"

#Loading module for dhcpcd
insmod /lib/af_packet.o
echo "etape af_packet.o END"

#Running dhcpcd
dhcpcd
echo "DHCPCD END"

#Verifying if the eth0 adapter is working
ifconfig

portmap -v &
# Download the RUN image and copy it in the raid device
mount -t nfs -o nolock 10.0.1.100:/tftpboot /mnt
cat /mnt/thinimage_RUN.img.gz | gzip -dc > /dev/md0

killall -9 dhcpcd
killall -9 portmap
umount /tftpboot

# Mount the md0

echo "Mounting md0"
if mount -t ext2 -o ro -o nocheck /dev/md0 /mnt; then
  echo "Mounted"
else
  error="Cannot mount, RAID and/or network problems?"
fi


# Pivot the root, umount /proc

echo "Unmounting /proc, changing root from"
echo "initial ram disk to md0 root"
if umount /proc && cd /mnt && pivot_root . initrd; then
  echo "Root changed"
else
  error="Problem using the new root"
fi


# Get rid of the original console device file still
# in the initrd
if exec </dev/console >/dev/console 2>/dev/console; then
  echo "Console switched"
fi

# Pass control to the normal init script in the raid root
exec /sbin/init 3
```

Figure 10. Complete functional linuxrc file for RAID in ram, loadable NIC support and network retrieval of the image