Using the initial RAM disk (initrd)

Written 1996,2000 by Werner Almesberger <werner.almesberger@epfl.ch> and
Hans Lermen <lermen@fgan.de>

initrd provides the capability to load a RAM disk by the boot loader.
This RAM disk can then be mounted as the root file system and programs
can be run from it. Afterwards, a new root file system can be mounted
from a different device. The previous root (from initrd) is then moved
to a directory and can be subsequently unmounted.

initrd is mainly designed to allow system startup to occur in two phases,
where the kernel comes up with a minimum set of compiled-in drivers, and
where additional modules are loaded from initrd.

This document gives a brief overview of the use of initrd. A more detailed
discussion of the boot process can be found in [1].

Operation

When using initrd, the system typically boots as follows:

1) the boot loader loads the kernel and the initial RAM disk
2) the kernel converts initrd into a "normal" RAM disk and
   frees the memory used by initrd
3) if the root device is not /dev/ram0, the old (deprecated)
   change_root procedure is followed. see the "Obsolete root change
   mechanism" section below.
4) root device is mounted. if it is /dev/ram0, the initrd image is
   then mounted as root
5) /sbin/init is executed (this can be any valid executable, including
   shell scripts; it is run with uid 0 and can do basically everything
   init can do).
6) init mounts the "real" root file system
7) init places the root file system at the root directory using the
   pivot_root system call
8) init execs the /sbin/init on the new root filesystem, performing
   the usual boot sequence
9) the initrd file system is removed

Note that changing the root directory does not involve unmounting it.
It is therefore possible to leave processes running on initrd during that
procedure. Also note that file systems mounted under initrd continue to
be accessible.

Boot command-line options

initrd adds the following new options:

initrd=<path>   (e.g. LOADLIN)

Loads the specified file as the initial RAM disk. When using LILO, you have to specify the RAM disk image file in /etc/lilo.conf, using the INITRD configuration variable.

noinitrd

initrd data is preserved but it is not converted to a RAM disk and the "normal" root file system is mounted. initrd data can be read from /dev/initrd. Note that the data in initrd can have any structure in this case and doesn't necessarily have to be a file system image. This option is used mainly for debugging.

Note: /dev/initrd is read-only and it can only be used once. As soon as the last process has closed it, all data is freed and /dev/initrd can't be opened anymore.

root=/dev/ram0

initrd is mounted as root, and the normal boot procedure is followed, with the RAM disk mounted as root.

Compressed cpio images

Recent kernels have support for populating a ramdisk from a compressed cpio archive. On such systems, the creation of a ramdisk image doesn't need to involve special block devices or loopbacks; you merely create a directory on disk with the desired initrd content, cd to that directory, and run (as an example):

```
find . | cpio --quiet -H newc -o | gzip -9 -n > /boot/imagefile.img
```

Examining the contents of an existing image file is just as simple:

```
mkdir /tmp/imagefile
cd /tmp/imagefile
gzip -cd /boot/imagefile.img | cpio -imd --quiet
```

Installation

First, a directory for the initrd file system has to be created on the "normal" root file system, e.g.

```
# mkdir /initrd
```

The name is not relevant. More details can be found on the pivot_root(2) man page.

If the root file system is created during the boot procedure (i.e. if you're building an install floppy), the root file system creation procedure should create the /initrd directory.

If initrd will not be mounted in some cases, its content is still accessible if the following device has been created:

```
# mknod /dev/initrd b 1 250
# chmod 400 /dev/initrd
```

Second, the kernel has to be compiled with RAM disk support and with support for the initial RAM disk enabled. Also, at least all components needed to execute programs from initrd (e.g. executable format and file system) must be compiled into the kernel.

Third, you have to create the RAM disk image. This is done by creating a file system on a block device, copying files to it as needed, and then copying the content of the block device to the initrd file. With recent kernels, at least three types of devices are suitable for that:

- a floppy disk (works everywhere but it's painfully slow)
- a RAM disk (fast, but allocates physical memory)
- a loopback device (the most elegant solution)

We'll describe the loopback device method:

1) make sure loopback block devices are configured into the kernel
2) create an empty file system of the appropriate size, e.g.
   ```
   # dd if=/dev/zero of=initrd bs=300k count=1
   # mke2fs -F -m0 initrd
   ```
   (if space is critical, you may want to use the Minix FS instead of Ext2)
3) mount the file system, e.g.
   ```
   # mount -t ext2 -o loop initrd /mnt
   ```
4) create the console device:
   ```
   # mkdir /mnt/dev
   # mknod /mnt/dev/console c 5 1
   ```
5) copy all the files that are needed to properly use the initrd environment. Don't forget the most important file, /sbin/init
   Note that /sbin/init's permissions must include "x" (execute).
6) correct operation the initrd environment can frequently be tested even without rebooting with the command
   ```
   # chroot /mnt /sbin/init
   ```
   This is of course limited to initrds that do not interfere with the general system state (e.g. by reconfiguring network interfaces, overwriting mounted devices, trying to start already running demons, etc. Note however that it is usually possible to use pivot_root in such a chroot'ed initrd environment.)
7) unmount the file system
   ```
   # umount /mnt
   ```
8) the initrd is now in the file "initrd". Optionally, it can now be compressed
   ```
   # gzip -9 initrd
   ```

For experimenting with initrd, you may want to take a rescue floppy and only add a symbolic link from /sbin/init to /bin/sh. Alternatively, you can try the experimental newlib environment [2] to create a small initrd.

Finally, you have to boot the kernel and load initrd. Almost all Linux boot loaders support initrd. Since the boot process is still compatible with an older mechanism, the following boot command line parameters have to be given:

```
  root=/dev/ram0 rw
```

(rw is only necessary if writing to the initrd file system.)

With LOADLIN, you simply execute

```
     LOADLIN <kernel> initrd=<disk_image>
e.g. LOADLIN C:\LINUX\BZIMAGE initrd=C:\LINUX\INITRD.GZ root=/dev/ram0 rw
```

With LILO, you add the option INITRD=<path> to either the global section or to the section of the respective kernel in /etc/lilo.conf, and pass the options using APPEND, e.g.

```
  image = /bzImage
    initrd = /boot/initrd.gz
    append = "root=/dev/ram0 rw"
```

and run /sbin/lilo

For other boot loaders, please refer to the respective documentation.

Now you can boot and enjoy using initrd.

Changing the root device

When finished with its duties, init typically changes the root device and proceeds with starting the Linux system on the "real" root device.

The procedure involves the following steps:
 - mounting the new root file system
 - turning it into the root file system
 - removing all accesses to the old (initrd) root file system
 - unmounting the initrd file system and de-allocating the RAM disk

Mounting the new root file system is easy: it just needs to be mounted on a directory under the current root. Example:

```
# mkdir /new-root
# mount -o ro /dev/hda1 /new-root
```

The root change is accomplished with the pivot_root system call, which is also available via the pivot_root utility (see pivot_root(8) man page; pivot_root is distributed with util-linux version 2.10h or higher [3]). pivot_root moves the current root to a directory under the new root, and puts the new root at its place. The directory for the old root must exist before calling pivot_root. Example:

```
# cd /new-root
# mkdir initrd
# pivot_root . initrd
```

Now, the init process may still access the old root via its executable, shared libraries, standard input/output/error, and its

current root directory. All these references are dropped by the
following command:

```
# exec chroot . what-follows <dev/console >dev/console 2>&1
```

Where what-follows is a program under the new root, e.g. /sbin/init
If the new root file system will be used with udev and has no valid
/dev directory, udev must be initialized before invoking chroot in order
to provide /dev/console.

Note: implementation details of pivot_root may change with time. In order
to ensure compatibility, the following points should be observed:

- before calling pivot_root, the current directory of the invoking
  process should point to the new root directory
- use . as the first argument, and the _relative_ path of the directory
  for the old root as the second argument
- a chroot program must be available under the old and the new root
- chroot to the new root afterwards
- use relative paths for dev/console in the exec command

Now, the initrd can be unmounted and the memory allocated by the RAM
disk can be freed:

```
# umount /initrd
# blockdev --flushbufs /dev/ram0
```

It is also possible to use initrd with an NFS-mounted root, see the
pivot_root(8) man page for details.

Usage scenarios

The main motivation for implementing initrd was to allow for modular
kernel configuration at system installation. The procedure would work
as follows:

1) system boots from floppy or other media with a minimal kernel
   (e.g. support for RAM disks, initrd, a.out, and the Ext2 FS) and
   loads initrd
2) /sbin/init determines what is needed to (1) mount the "real" root FS
   (i.e. device type, device drivers, file system) and (2) the
   distribution media (e.g. CD-ROM, network, tape, ...). This can be
   done by asking the user, by auto-probing, or by using a hybrid
   approach.
3) /sbin/init loads the necessary kernel modules
4) /sbin/init creates and populates the root file system (this doesn't
   have to be a very usable system yet)
5) /sbin/init invokes pivot_root to change the root file system and
   execs - via chroot - a program that continues the installation
6) the boot loader is installed
7) the boot loader is configured to load an initrd with the set of
   modules that was used to bring up the system (e.g. /initrd can be
   modified, then unmounted, and finally, the image is written from
   /dev/ram0 or /dev/rd/0 to a file)

8) now the system is bootable and additional installation tasks can be performed

The key role of initrd here is to re-use the configuration data during normal system operation without requiring the use of a bloated "generic" kernel or re-compiling or re-linking the kernel.

A second scenario is for installations where Linux runs on systems with different hardware configurations in a single administrative domain. In such cases, it is desirable to generate only a small set of kernels (ideally only one) and to keep the system-specific part of configuration information as small as possible. In this case, a common initrd could be generated with all the necessary modules. Then, only /sbin/init or a file read by it would have to be different.

A third scenario is more convenient recovery disks, because information like the location of the root FS partition doesn't have to be provided at boot time, but the system loaded from initrd can invoke a user-friendly dialog and it can also perform some sanity checks (or even some form of auto-detection).

Last not least, CD-ROM distributors may use it for better installation from CD, e.g. by using a boot floppy and bootstrapping a bigger RAM disk via initrd from CD; or by booting via a loader like LOADLIN or directly from the CD-ROM, and loading the RAM disk from CD without need of floppies.

Obsolete root change mechanism

The following mechanism was used before the introduction of pivot_root. Current kernels still support it, but you should _not_ rely on its continued availability.

It works by mounting the "real" root device (i.e. the one set with rdev in the kernel image or with root=... at the boot command line) as the root file system when linuxrc exits. The initrd file system is then unmounted, or, if it is still busy, moved to a directory /initrd, if such a directory exists on the new root file system.

In order to use this mechanism, you do not have to specify the boot command options root, init, or rw. (If specified, they will affect the real root file system, not the initrd environment.)

If /proc is mounted, the "real" root device can be changed from within linuxrc by writing the number of the new root FS device to the special file /proc/sys/kernel/real-root-dev, e.g.

```
# echo 0x301 >/proc/sys/kernel/real-root-dev
```

Note that the mechanism is incompatible with NFS and similar file systems.

This old, deprecated mechanism is commonly called "change_root", while the new, supported mechanism is called "pivot_root".

Mixed change_root and pivot_root mechanism

In case you did not want to use root=/dev/ram0 to trigger the pivot_root mechanism, you may create both /linuxrc and /sbin/init in your initrd image.

/linuxrc would contain only the following:

```
#! /bin/sh
mount -n -t proc proc /proc
echo 0x0100 >/proc/sys/kernel/real-root-dev
umount -n /proc
```

Once linuxrc exited, the kernel would mount again your initrd as root, this time executing /sbin/init. Again, it would be the duty of this init to build the right environment (maybe using the root= device passed on the cmdline) before the final execution of the real /sbin/init.

Resources
---

[1] Almesberger, Werner; "Booting Linux: The History and the Future"
    http://www.almesberger.net/cv/papers/ols2k-9.ps.gz
[2] newlib package (experimental), with initrd example
    http://sources.redhat.com/newlib/
[3] Brouwer, Andries; "util-linux: Miscellaneous utilities for Linux"
    ftp://ftp.win.tue.nl/pub/linux-local/utils/util-linux/