Ramfs, rootfs and initramfs

October 17, 2005

Rob Landley <rob@landley.net>

What is ramfs?

Ramfs is a very simple filesystem that exports Linux's disk caching mechanisms (the page cache and dentry cache) as a dynamically resizable RAM-based filesystem.

Normally all files are cached in memory by Linux. Pages of data read from backing store (usually the block device the filesystem is mounted on) are kept around in case it's needed again, but marked as clean (freeable) in case the Virtual Memory system needs the memory for something else. Similarly, data written to files is marked clean as soon as it has been written to backing store, but kept around for caching purposes until the VM reallocates the memory. A similar mechanism (the dentry cache) greatly speeds up access to directories.

With ramfs, there is no backing store. Files written into ramfs allocate dentries and page cache as usual, but there's nowhere to write them to. This means the pages are never marked clean, so they can't be freed by the VM when it's looking to recycle memory.

The amount of code required to implement ramfs is tiny, because all the work is done by the existing Linux caching infrastructure. Basically, you're mounting the disk cache as a filesystem. Because of this, ramfs is not an optional component removable via menuconfig, since there would be negligible space savings.

ramfs and ramdisk:

The older "ram disk" mechanism created a synthetic block device out of an area of RAM and used it as backing store for a filesystem. This block device was of fixed size, so the filesystem mounted on it was of fixed size. Using a ram disk also required unnecessarily copying memory from the fake block device into the page cache (and copying changes back out), as well as creating and destroying dentries. Plus it needed a filesystem driver (such as ext2) to format and interpret this data.

Compared to ramfs, this wastes memory (and memory bus bandwidth), creates unnecessary work for the CPU, and pollutes the CPU caches. (There are tricks to avoid this copying by playing with the page tables, but they're unpleasantly complicated and turn out to be about as expensive as the copying anyway.) More to the point, all the work ramfs is doing has to happen _anyway_, since all file access goes through the page and dentry caches. The RAM disk is simply unnecessary; ramfs is internally much simpler.

Another reason ramdisks are semi-obsolete is that the introduction of loopback devices offered a more flexible and convenient way to create synthetic block devices, now from files instead of from chunks of memory. See losetup (8) for details.

ramfs and tmpfs:

One downside of ramfs is you can keep writing data into it until you fill up all memory, and the VM can't free it because the VM thinks that files should get written to backing store (rather than swap space), but ramfs hasn't got any backing store. Because of this, only root (or a trusted user) should be allowed write access to a ramfs mount.

A ramfs derivative called tmpfs was created to add size limits, and the ability to write the data to swap space. Normal users can be allowed write access to tmpfs mounts. See Documentation/filesystems/tmpfs.rst for more information.

What is rootfs?

Rootfs is a special instance of ramfs (or tmpfs, if that's enabled), which is always present in 2.6 systems. You can't unmount rootfs for approximately the same reason you can't kill the init process; rather than having special code to check for and handle an empty list, it's smaller and simpler for the kernel to just make sure certain lists can't become empty.

Most systems just mount another filesystem over rootfs and ignore it. The amount of space an empty instance of ramfs takes up is tiny.

If CONFIG_TMPFS is enabled, rootfs will use tmpfs instead of ramfs by default. To force ramfs, add "rootfstype=ramfs" to the kernel command line.

What is initramfs?

All 2.6 Linux kernels contain a gzipped "cpio" format archive, which is extracted into rootfs when the kernel boots up. After extracting, the kernel checks to see if rootfs contains a file "init", and if so it executes it as PID 1. If found, this init process is responsible for bringing the system the rest of the way up, including locating and mounting the real root device (if any). If rootfs does not contain an init program after the embedded cpio archive is extracted into it, the kernel will fall through to the older code to locate and mount a root partition, then exec some variant of /sbin/init out of that.

All this differs from the old initrd in several ways:

- The old initrd was always a separate file, while the initramfs archive is linked into the linux kernel image. (The directory `linux-*/usr` is devoted to generating this archive during the build.)
- The old initrd file was a gzipped filesystem image (in some file format, such as ext2, that needed a driver built into the kernel), while the new initramfs archive is a gzipped cpio archive (like tar only simpler, see cpio(1) and Documentation/driver-api/early-userspace/buffer-format.rst). The kernel's cpio extraction code is not only extremely small, it's also __init text and data that can be discarded during the boot process.
- The program run by the old initrd (which was called /initrd, not /init) did some setup and then returned to the kernel, while the init program from initramfs is not expected to return to the kernel. (If /init needs to hand off control it can overmount / with a new root device and exec another init program. See the switch_root utility, below.)
- When switching another root device, initrd would pivot_root and then umount the ramdisk. But initramfs is rootfs: you can neither pivot_root rootfs, nor unmount it. Instead delete everything out of rootfs to free up the space (find -xdev / -exec rm '{}' ';'), overmount rootfs with the new root (cd /newmount; mount --move . /; chroot .), attach stdin/stdout/stderr to the new /dev/console, and exec the new init.

  Since this is a remarkably persnickety process (and involves deleting commands before you can run them), the klibc package introduced a helper program (utils/run_init.c) to do all this for you. Most other packages (such as busybox) have named this command "switch_root".

Populating initramfs:

The 2.6 kernel build process always creates a gzipped cpio format initramfs archive and links it into the resulting kernel binary. By default, this archive is empty (consuming 134 bytes on x86).

The config option CONFIG_INITRAMFS_SOURCE (in General Setup in menuconfig, and living in usr/Kconfig) can be used to specify a source for the initramfs archive, which will automatically be incorporated into the resulting binary. This option can point to an existing gzipped cpio archive, a directory containing files to be archived, or a text file specification such as the following example:

```
dir /dev 755 0 0
nod /dev/console 644 0 0 c 5 1
nod /dev/loop0 644 0 0 b 7 0
dir /bin 755 1000 1000
slink /bin/sh busybox 777 0 0
file /bin/busybox initramfs/busybox 755 0 0
dir /proc 755 0 0
dir /sys 755 0 0
dir /mnt 755 0 0
file /init initramfs/init.sh 755 0 0
```

Run "usr/gen_init_cpio" (after the kernel build) to get a usage message documenting the above file format.

One advantage of the configuration file is that root access is not required to set permissions or create device nodes in the new archive. (Note that those two example "file" entries expect to find files named "init.sh" and "busybox" in a directory called "initramfs", under the linux-2.6.* directory. See Documentation/driver-api/early-userspace/early_userspace_support.rst for more details.)

The kernel does not depend on external cpio tools. If you specify a directory instead of a configuration file, the kernel's build infrastructure creates a configuration file from that directory (usr/Makefile calls usr/gen_initramfs.sh), and proceeds to package up that directory using the config file (by feeding it to usr/gen_init_cpio, which is created from usr/gen_init_cpio.c). The kernel's build-time cpio creation code is entirely self-contained, and the kernel's boot-time extractor is also (obviously) self-contained.

The one thing you might need external cpio utilities installed for is creating or extracting your own preprepared cpio files to feed to the kernel build (instead of a config file or directory).

The following command line can extract a cpio image (either by the above script or by the kernel build) back into its component files:

```
cpio -i -d -H newc -F initramfs_data.cpio --no-absolute-filenames
```

The following shell script can create a prebuilt cpio archive you can use in place of the above config file:

```
#!/bin/sh

# Copyright 2006 Rob Landley <rob@landley.net> and TimeSys Corporation.
# Licensed under GPL version 2

if [ $# -ne 2 ]
then
  echo "usage: mkinitramfs directory imagename.cpio.gz"
  exit 1
fi

if [ -d "$1" ]
then
  echo "creating $2 from $1"
  (cd "$1"; find . | cpio -o -H newc | gzip) > "$2"
else
  echo "First argument must be a directory"
  exit 1
fi
```

Note

The cpio man page contains some bad advice that will break your initramfs archive if you follow it. It says "A typical way to generate the list of filenames is with the find command; you should give find the -depth option to minimize problems with permissions on directories that are unwritable or not searchable." Don't do this when creating initramfs.cpio.gz images, it won't work. The Linux kernel cpio extractor won't create files in a directory that doesn't exist, so the directory entries must go before the files that go in those directories. The above script gets them in the right order.

External initramfs images:

If the kernel has initrd support enabled, an external cpio.gz archive can also be passed into a 2.6 kernel in place of an initrd. In this case, the kernel will autodetect the type (initramfs, not initrd) and extract the external cpio archive into rootfs before trying to run /init.

This has the memory efficiency advantages of initramfs (no ramdisk block device) but the separate packaging of initrd (which is nice if you have non-GPL code you'd like to run from initramfs, without conflating it with the GPL licensed Linux kernel binary).

It can also be used to supplement the kernel's built-in initramfs image. The files in the external archive will overwrite any conflicting files in the built-in initramfs archive. Some distributors also prefer to customize a single kernel image with task-specific initramfs images, without recompiling.

Contents of initramfs:

An initramfs archive is a complete self-contained root filesystem for Linux. If you don't already understand what shared libraries, devices, and paths you need to get a minimal root filesystem up and running, here are some references:

- https://www.tldp.org/HOWTO/Bootdisk-HOWTO/
- https://www.tldp.org/HOWTO/From-PowerUp-To-Bash-Prompt-HOWTO.html
- http://www.linuxfromscratch.org/lfs/view/stable/

The "klibc" package (https://www.kernel.org/pub/linux/libs/klibc) is designed to be a tiny C library to statically link early userspace code against, along with some related utilities. It is BSD licensed.

I use uClibc (https://www.uclibc.org) and busybox (https://www.busybox.net) myself. These are LGPL and GPL, respectively. (A self-contained initramfs package is planned for the busybox 1.3 release.)

In theory you could use glibc, but that's not well suited for small embedded uses like this. (A "hello world" program statically linked against glibc is over 400k. With uClibc it's 7k. Also note that glibc dlopens libnss to do name lookups, even when otherwise statically linked.)

A good first step is to get initramfs to run a statically linked "hello world" program as init, and test it under an emulator like qemu (www.qemu.org) or User Mode Linux, like so:

```
cat > hello.c << EOF
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  printf("Hello world!\n");
  sleep(999999999);
}
EOF
gcc -static hello.c -o init
echo init | cpio -o -H newc | gzip > test.cpio.gz
# Testing external initramfs using the initrd loading mechanism.
qemu -kernel /boot/vmlinuz -initrd test.cpio.gz /dev/zero
```

When debugging a normal root filesystem, it's nice to be able to boot with "init=/bin/sh". The initramfs equivalent is "rdinit=/bin/sh", and it's just as useful.

Why cpio rather than tar?

This decision was made back in December, 2001. The discussion started here:

> http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.2/1538.html

And spawned a second thread (specifically on tar vs cpio), starting here:

> http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.2/1587.html

The quick and dirty summary version (which is no substitute for reading the above threads) is:

1. cpio is a standard. It's decades old (from the AT&T days), and already widely used on Linux (inside RPM, Red Hat's device driver disks). Here's a Linux Journal article about it from 1996:

   http://www.linuxjournal.com/article/1213

   It's not as popular as tar because the traditional cpio command line tools require _truly_hideous_ command line arguments. But that says nothing either way about the archive format, and there are alternative tools, such as:

   http://freecode.com/projects/afio

2. The cpio archive format chosen by the kernel is simpler and cleaner (and thus easier to create and parse) than any of the (literally dozens of) various tar archive formats. The complete initramfs archive format is explained in buffer-format.txt, created in usr/gen_init_cpio.c, and extracted in init/initramfs.c. All three together come to less than 26k total of human-readable text.
3. The GNU project standardizing on tar is approximately as relevant as Windows standardizing on zip. Linux is not part of either, and is free to make its own technical decisions.
4. Since this is a kernel internal format, it could easily have been something brand new. The kernel provides its own tools to create and extract this format anyway. Using an existing standard was preferable, but not essential.
5. Al Viro made the decision (quote: "tar is ugly as hell and not going to be supported on the kernel side"):

   http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.2/1540.html

   explained his reasoning:

   - http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.2/1550.html
   - http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.2/1638.html

   and, most importantly, designed and implemented the initramfs code.

Future directions:

Today (2.6.16), initramfs is always compiled in, but not always used. The kernel falls back to legacy boot code that is reached only if initramfs does not contain an /init program. The fallback is legacy code, there to ensure a smooth transition and allowing early boot functionality to gradually move to "early userspace" (I.E. initramfs).

The move to early userspace is necessary because finding and mounting the real root device is complex. Root partitions can span multiple devices (raid or separate journal). They can be out on the network (requiring dhcp, setting a specific MAC address, logging into a server, etc). They can live on removable media, with dynamically allocated major/minor numbers and persistent naming issues requiring a full udev implementation to sort out. They can be compressed, encrypted, copy-on-write, loopback mounted, strangely partitioned, and so on.

This kind of complexity (which inevitably includes policy) is rightly handled in userspace. Both klibc and busybox/uClibc are working on simple initramfs packages to drop into a kernel build.

The klibc package has now been accepted into Andrew Morton's 2.6.17-mm tree. The kernel's current early boot code (partition detection, etc) will probably be migrated into a default initramfs, automatically created and used by the kernel build.