Tech Tip: How to use initramfs.

By: Rob Landley

Last time, we covered why initramfs was created: because it saves memory, gives the user more control over the boot process, and simplifies the kernel's internal implementation. This article is about understanding how initramfs works, and shows how to package a root filesystem as an initramfs. The third article in the series is about creating root file systems that take advantage of initramfs.

So how does it work? (Rootfs and cpio.)

When each 2.6 kernel boots, it mounts "rootfs" as its first filesystem. This is a special instance of tmpfs which can't be moved or unmounted.[1] Most 2.6 systems just leave it empty and mount another root filesystem on top of it, but rootfs is always there (check /proc/mounts to see) and it's a fully capable ram based filesystem.

Shortly after mounting rootfs during bootup, the kernel extracts a gzipped cpio archive into it.[2] Each 2.6 kernel image has one of these archives built into it, but by default it's empty, so extracting it adds no files to rootfs. Then the kernel tries to run "/init" out of rootfs, and if that works the kernel is done booting, and the newly spawned init program takes over. Only if the kernel can't run "/init" will it fall back to the legacy behavior of parsing the "root=" kernel command line to find another filesystem to mount on top of rootfs, so it can try to run an init program out of the new filesystem.

Using initramfs just means supplying the kernel with an /init program for rootfs, either by replacing the cpio.gz archive built into the kernel or by supplying a cpio.gz archive externally the way initrd used to do it.

Enough talk, let's try something.

It's a bit hard to get excited about packaging instructions, so let's try a practical demonstration and build a kernel that actually does something we can see.

For our first init program to feed to initramfs, let's start with a modified "hello world". The contents of rootfs are just like any other root filesystem: if you like you can have /etc and /usr and /tmp, mount /proc and /sysfs under it, and so on. But all you really need is an /init that runs. This version sleeps instead of exiting, because if PID 1 exits the kernel will panic, and that's distracting.[3]

#include

int main(int argc, char \*argv[])

{

printf("Hello world\n");

sleep(999999999);

}

Compile a statically linked version of the above program, so we don't have to worry about copying shared libraries just yet[4]:

gcc -static hello.c -o hello

If you run "./hello", it should print hello world, and then hang. Type ctrl-c to get out of it. If initramfs runs this hello program as init, we'll be able to see the result at the end of the boot messages.

Note: If you're cross-compiling a linux kernel for a different processor, use your cross-compiler to compile this hello world program. The packaging steps described below are platform-independent, but executable files to be run on the target platform are not.

So how do we feed this thing to the kernel? Well, there are four basic methods: you can supply an external file, or replace the one built into the kernel in any of three different ways. When is each one appropriate? Let's go through them.

Initializing initramfs from an external file.

Most users of initramfs build it into the kernel image, but you don't have to. Any 2.6 kernel that has initrd support enabled can use a cpio.gz file instead. The kernel will autodetect the file type, and instead of creating a ramdisk block device it will extract it into rootfs, so this still has the memory efficiency advantages of initramfs.

An external initramfs archive is extracted after the kernel's built-in archive, so if the two contain any of the same files the external archive should overwrite the built-in one.[5] This means you can update or customize rootfs without replacing your kernel.

Another reason you might want to use this method is licensing. If you want to run non-GPL programs from rootfs, or want to supply non-GPL firmware to statically linked device drivers, your lawyers might be happier if it was in a separate file rather than bundled into your kernel.

So where do we get a cpio.gz archive? One way is to create your own with the cpio and gzip commands. It's easier and more flexible to have the kernel build do it for you (we'll get to that next), but if you want to do it yourself, packing up our hello world program using the the command line cpio archiver would look something like this:

mkdir sub

cp hello sub/init

cd sub

find . | cpio -o -H newc | gzip > ../initramfs\_data.cpio.gz

cd ..

rm -rf sub

If you pass the above initramfs\_data.cpio.gz file to a 2.6 kernel using the traditional initrd mechanism, it should display the "hello world" message at the end of the boot, and hang until reboot. Just boot it on any test system to see the result.[6]

If it doesn't work, make sure that initial ramdisk support is selected, that your init program is indeed statically linked, has the executable bit set, and has the correct name. You can extract any initramfs archive into the current directory via:

zcat initramfs\_data.cpio.gz | cpio -i -d -H newc --no-absolute-filenames

Building initramfs into the kernel

The easiest way to use initramfs is to replace the kernel's built-in empty cpio archive with a non-empty one. This doesn't require any particular kernel features to be enabled; all 2.6 kernels have this support built-in.

The kernel config option CONFIG\_INITRAMFS\_SOURCE (I.E. General setup ---> Initramfs source file(s) in menuconfig) indicates where the kernel build should get files to archive for initramfs. By default this is empty, so it builds an empty archive.[7]

CONFIG\_INITRAMFS\_SOURCE can point to an absolute path, or a path relative to the top level of the kernel's build directory. The target can be any of three things: an existing cpio.gz archive, a directory full of files to create such an archive from, or a text configuration file. The third is the most flexible, but let's go through them in order.

Copying an existing cpio.gz file into the kernel

If you already have your own initramfs\_data.cpio.gz file (because you created it yourself, or saved the cpio.gz file produced by a previous kernel build), you can point CONFIG\_INITRAMFS\_SOURCE at it and the kernel build will autodetect the file type and link it into the resulting kernel image.

You can also leave CONFIG\_INITRAMFS\_SOURCE empty, and instead copy your cpio.gz file to usr/initramfs\_data.cpio.gz in your kernel's build directory. The kernel's makefile won't generate a new archive if it doesn't need to.

Either way, if you build a kernel like this you can boot it without supplying an external initrd image, and it'll still finish its boot by running your init program out of rootfs. This is packaging method #2, if you'd like to try it now.

Supplying a directory of files for initramfs

If CONFIG\_INITRAMFS\_SOURCE points to a directory, the kernel will archive it up for you. This is a very easy way to create an initramfs archive, and is method #3.

Interestingly, the kernel build doesn't use the standard cpio command to create initramfs archives. You don't even need to have any cpio tools installed on your build system. Instead the kernel build (in usr/Makefile) generates a text file describing the directory with the script "gen\_initramfs\_list.sh", and then feeds that descript to a program called "gen\_init\_cpio" (built from C source in the kernel's usr directory), which create the cpio archive. This looks something like the following:

scripts/gen\_initramfs\_list.sh $CONFIG\_INITRAMFS\_SOURCE > usr/initramfs\_list

usr/gen\_init\_cpio usr/initramfs\_list > usr/initramfs\_data.cpio

gzip usr/initramfs\_data.cpio

To package up our hello world program, you could simply copy it into its own directory, name it "init", point CONFIG\_INITRAMFS\_SOURCE at that directory, and rebuild the kernel. The resulting kernel should end its boot by printing "hello world". And if you need to tweak the contents of that directory, rebuilding the kernel will re-package the contents of that directory if anything has changed.

The downside of this method is that it if your initramfs has device nodes, or cares about file ownership and permissions, you need to be able to create those things in a directory for it to copy. This is hard to do if you haven't got root access, or are using a cross-compile environment like cygwin. That's where the fourth and final method comes in.

Using an initramfs\_list configuration file.

This is the most flexible method. The kernel's gen\_initramfs\_list.sh script creates a text description file listing the contents of initramfs, and gen\_init\_cpio uses this file to produce an archive. This file is a standard text file, easily editable, containing one line per file. Each line starts with a keyword indicating what type of entry it describes.

The config file to create our "hello world" initramfs only needs a single line:

file /init usr/hello 500 0 0

This takes the file "hello" and packages it so it shows up as /init in rootfs, with permissions 500, with uid and gid 0. It expects to find the source file "hello" in a "usr" subdirectory under the kernel's build directory. (If you're building the kernel in a different directory than the source directory, this path would be relative to the build directory, not the source directory.)

To try it yourself, copy "hello" into usr in the kernel's build directory, copy the above configuration line to its own file, use "make menuconfig" to point CONFIG\_INITRAMFS\_SOURCE to that file, run the kernel build, and test boot the new kernel. Alternately, you can put the "hello" file in its own directory and use "scripts/gen\_initramfs\_list.sh dirname" to create a configuration file (where dirname is the path to your directory, from the kernel's build directory). For large projects, you may want to generate a starting configuration with the script, and then customize it with any text editor.

This configuration file can also specify device nodes (with the "nod" keyword), directories ("dir"), symbolic links ("slink"), named FIFO pipes ("pipe"), and unix domain sockets ("sock"). Full documentation on this file's format is available by running "usr/gen\_init\_cpio" (with no arguments) after a kernel build.

A more complicated example containing device nodes and symlinks could look like this:

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

One significant advantage of the configuration file method is that any regular user can create one, specifying ownership and permissions and the creation of device nodes in initramfs, without any special permissions on the build system. Creating a cpio archive using the cpio command line tool, or pointing the kernel build at a directory, requires a directory that contains everything initramfs will contain. The configuration file method merely requires a few source files to get data from, and a description file.

This also comes in handy cross-compiling from other environments such as cygwin, where the local filesystem may not even be capable of reproducing everything initramfs should have in it.

**Conclusion**

The four different ways to populate rootfs all have the same result: a set of files are extracted into rootfs during the kernel's boot process, and if one of those files is an executable "/init" then the kernel runs that instead of mounting whatever root= points to.

Once the init program in initramfs starts up, the kernel considers the boot process finished. The new process is in charge, it's running as the special process ID #1 which is reserved for init, and anything else the system does must be started by init. As always, if PID 1 exits the kernel will panic.

Next week, we'll cover several things an init process can do while running from rootfs.

--------------------------------------------------------------------------------

Footnote 1: The kernel doesn't allow rootfs to be unmounted for the same reason the same reason it won't let the first process (PID 1, generally running init) be killed. The fact the lists of mounts and processes are never empty simplifies the kernel's implementation.

Footnote 2: The cpio format is another way of combining files together, like tar and zip. It's an older and simpler storage format that dates back to the original unix, and it's the storage format used inside RPM packages. It's not as widely used as tar or zip because the command line syntax of the cpio command is unnecessarily complicated (type "man 1 cpio" at a Linux or Cygwin command line if you have a strong stomach). Luckily, we don't need to use this command.

Footnote 3: The kernel will always panic if PID 1 exits; this is unrelated to initramfs. All of the signals that might kill init are blocked, even "kill -9" which will reliably kill any other process. But init can still call the exit() syscall itself, and the kernel panics if this happens in PID 1. Avoiding it here is mostly a cosmetic issue: we don't want the panic scrolling our "Hello World!" message off the top of the screen.

Footnote 4: Statically linking programs against glibc produces enormous, bloated binaries. Yes, this is expected to be over 400k for a hello world proram. You can try using the "strip" command on the resulting binary, but it won't help much. This sort of bloat is why uClibc exists.

Footnote 5: Older 2.6 kernels had a bug where they would append to duplicate files rather than overwriting. Test your kernel version before depending on this behavior.

Footnote 6:User Mode Linux or QEMU can be very helpful testing out initramfs, but are beyond the scope of this article.

Footnote 7: Well, sort of. The default one is probably meant to be empty, but due to a small bug (gen\_initramfs\_list.sh spits out an example file when run with no arguments) the version in the 2.6.16 kernel actually contains a "/dev/console" node and a "/root" directory, which aren't used for anything. It gzips down to about 135 bytes, and might as well actually be empty. On Intel you can run "readelf -S vmlinux" and look for section ".init.ramfs" to see the cpio.gz archive linked into a 2.6 kernel. Elf section names might vary a bit on other platforms.