LinuxDevices.com printer-friendly story display of:
http://www.linuxdevices.com/articles/AT9049109449.html . . .

ELJOnline:
Brian Writes about His BOEL

Brian Elliott Finley

Brian's single-floppy Linux reveals software-shrinking secrets of the Linux masters.

Brian's Own Embedded Linux, or BOEL as it is affectionately known, was created as part of the SystemImager Project. SystemImager is a tool used for, well, imaging systems and distributing live updates. Part of the imaging process involves booting a client machine in such a way that its hard disks are not in use and therefore are available for manipulation. The SystemImager autoinstall client does this by using a root filesystem that exists entirely in memory, or an initrd (initial RAM disk). This initrd and the kernel that goes with it are the embedded Linux system we call BOEL.

One of my initial requirements for BOEL was that it had to fit on a single floppy. I wanted to be able to boot a virgin machine from a floppy diskette and have it come up to a point where it could communicate with the network and access its hard disk(s). From that point, I figured I could pull over any scripts or tools necessary that didn't fit on the floppy itself.

Static Binaries? Standard Libraries? Debian's PIC Libraries!

I started by using tomsrtbt and simply added my custom commands to one of its initialization scripts. As SystemImager progressed, I began to customize tomsrtbt. One day I hit a big snag. I needed to add some new binaries to the boot diskette. I copied the binaries over and hoped they would work. Some did, but some didn't. For the ones that didn't, it was because the binaries were linked dynamically to libraries that didn't match up with the libraries on the boot diskette.

After a bit of contemplation, I tried two different hacks before stumbling upon a real solution. The first hack was to install an old version of Linux with a small libc5, custom compile all my binaries there and copy the binaries and the relevant libraries over. The second hack was to compile the binaries with statically linked libraries. By compiling the libraries directly into the binaries, it no longer mattered which libraries, if any, existed on the boot diskette. While this worked for adding one or two binaries to tomsrtbt, I quickly ran out of space as I added new binaries over time. I was able to remove components from tomsrtbt that I didn't need, and I used **strip** on my binaries to remove the symbols and make them smaller. But having pieces of the libraries compiled into each static binary is very redundant and inefficient from a disk-space perspective.

At this point, I stumbled upon Debian's PIC (Position Independent Code) libraries. These are the standard libraries, broken up into little pieces. They are broken up in such a way that you can take only the pieces you need and combine them together into a single library file. No recompiling is necessary. You don't even have to know which pieces you require--there's a little shell script that does that for you called mklibs.sh. When you run this script, it examines the binary or binaries you point it at, determines which pieces of which libraries are needed, grabs the appropriate pieces from the PIC libraries, squashes them together and produces a directory full of library files--perfectly customized to your binaries, with no more and no less than exactly what you need. This alone was enough to make me a Debian convert.

Unfortunately, there is relatively little documentation on the PIC libraries. I'm not a C-library

developer, so take this with a grain of salt, but here is a little bit of additional (unconfirmed) information I've been able to glean from PIC-related scripts, the installed PIC libraries and their supporting files and my use of the wonderful little beasties:

- PIC libraries appear to be granular down to the function.
- As of this writing, the PIC libraries are available for glibc releases 2.0 through 2.2.5.
- PIC libraries are currently available for the glibc, libnewt and slang libraries.

If you want to try out the PIC libraries, run `apt-get install boot-floppies` from a Debian system. When that completes, untar /usr/src/boot-floppies.tar.gz. You will find the mklibs.sh script in the /usr/src/boot-floppies/scripts/rootdisk/ directory. To try it out, find a dynamically linked binary, say cat, and verify that it is dynamically linked with the **file** command: `file /bin/cat`. If it is dynamically linked, you'll see something like "dynamically linked (uses shared libs)" in the output. Now let's create a temporary directory for our test libraries: `mkdir /tmp/lib`. Then cd into the directory where our script lives:

```
cd /usr/src/boot-floppies/scripts/rootdisk/
```

Next, run the command to create our new libraries:

```
./mklibs.sh -v -d /tmp/lib /bin/cat
```

If we do an `ls -l /tmp/lib/`, we can see that our new C library is about 408K (on my system). To create a minimal library for more than one binary, just include them all at the end of the command line--wild cards are okay.

Which Shell to Use?

Trying to constrain an embedded Linux system to a single floppy can take quite a bit of lingering over issues that may not even cross the minds of people who have access to full-sized system disks. A great many of the implementation decisions were affected by size. Even the shell that is used can make a huge difference. Consider the 530K for bash vs. 82K for ash (as measured on my desktop system). Alone, bash would take over two-thirds of the space I had available on the RAM disk.

BusyBox

BusyBox is a wonderful utility and goes a long way toward providing many of the most common binaries in a really small footprint. Be sure to see the first three issues of *ELJ* for a series of articles on BusyBox. In a nutshell, BusyBox provides a single binary that acts as different utilities based on what you call it. So if you issue the command `/bin/cat`, and /bin/cat is linked to /bin/busybox, then BusyBox behaves like cat. If you issue the command `/bin/ping`, and /bin/ping is linked to /bin/busybox, then it behaves like ping. Wonderful!

Hard Links vs. Soft Links

So after using BusyBox for some time, one of my developers on the SystemImager Project, Dann

Frazier, noted that if you use hard links instead of soft links (as we had been), you can fit more on the RAM disk. Why would this make a difference? It's just a soft link and has no file size, right?

Almost. Soft links have no contents, but they do have their own inodes, and each inode takes up space in the filesystem. Hard links, on the other hand, are simply additional names by which you can reference the same file, and they are all stored in the same inode. If you do an `ls -l` on a soft link, you'll see that it actually does take up a very small amount of space. This small amount of space adds up quickly when you're building a system with a small footprint.

On the other hand, if you do an `ls -l` on a hard link, you might be taken aback to see that it appears to take up the same amount of space as the original file. Understanding why requires a slight paradigm shift. It is the same file as the original file. If you have a 10K file and create a dozen hard links to it, `ls -l` will tell you that each file takes up 10K, and the natural conclusion is that a total of 120K is used. But if you do an `ls -i` (show inode numbers), you'll see that all of the hard links share a single inode, which points to a single file. So a total of only 10K actually is used.

Creating the Initial RAM Disk

Now that I have my binaries, libraries and other files ready, it's time to create the actual RAM disk and copy my system to it. I issue the command

```
dd if=/dev/zero of=initrd bs=1024 count=2000
```

This tells dd to read 2,000, 1,024 byte blocks (1K blocks) from the /dev/zero device (which produces all zeros when you read from it) and write all those zeros to the file initrd.

After running this command, I have a 2MB file with nothing in it--literally. Starting out with this file filled with zeros is important because when we go to compress it later, the zeros compress really, really well. When we create the filesystem on this RAM disk, some of these zeros will be turned into patterns of 1s and 0s to hold the filesystem information. The same will happen when we start copying files to the filesystem. But if we start out with zeros, then parts of the RAM disk that are not touched by files or filesystem information will remain as zeros.

Which Filesystem Should I Use?

Now we need to create the filesystem on our RAM disk. Which filesystem should you use? Well, this depends on your requirements, and you must balance the size of the filesystem driver in the kernel (which must also fit on our floppy) with the space that its filesystem uses.

Minix, for example, is a filesystem that uses relatively little space for its filesystem information. On the other hand, ext2 takes up a fair amount more as it is a more advanced filesystem and has more features. However, in the case of BOEL, the ext2 driver must be in the kernel anyway (to create filesystems on autoinstall clients), and it turns out that adding the Minix driver to the kernel takes up more valuable space on the floppy then the amount of space saved by using the Minix filesystem.

Use Only the Inodes You Need

If you weren't being so careful about the size of things, you normally would create an ext2 filesystem

by simply issuing the command `mke2fs <device_name>`, in which case mke2fs figures out reasonable defaults for a number of filesystem properties. But since we're being so anal about this size thing, I'm going to give mke2fs some specific parameters--in particular, how many inodes to use.

How do I know how many inodes I'll need? Well, I can ask my computer. I can cd into my BOEL development directory and issue the following command:

```
find . -printf "%i\n" | sort -u | wc -l
```

This prints out the inode number for every file in the directory hierarchy, sorts out duplicates (from hard links) and counts the lines. The result is the exact number of inodes used. In my case I am using 499 inodes, but I want to add a few extra inodes for temporary files and directories that are created as part of BOEL's initialization process. So I add ten to the inode count and issue the command `mke2fs -N 509 initrd`.

Using the Loopback Driver to Mount the Initial RAM Disk

Now we have a file with a filesystem on it. Kind of funny, huh? Not really a disk, and certainly not a typical file, but we can mount it just like we would mount a normal disk. The only exception is that we have to use a special mount option (-o loop) that allows us to use the loopback driver to mount the file as if it were an actual disk device. So I create a mountpoint, `mkdir mount_point` and type

```
mount -o loop ./initrd ./mount_point
```

Putting Your BOEL on the Inital RAM Disk

The next step is to copy our embedded Linux system to our RAM disk. At first I used `cp -a` to copy the files to the RAM disk, but I noticed that the RAM disk didn't compress down as small as I expected. It turns out that as cp is transferring a file, it uses a temporary file until it finishes that file's transfer, then moves it to the proper location and filename.

But for our little system this just won't do. Every one of those temporary files turns some of our clean zeros into ones, and those ones don't go away when the temporary file is removed. Removing a file on an ext2 filesystem simply removes the reference to it, or the file's name. The file itself remains until it gets overwritten by a new file. Apparently tar just lays the files down in place. Being designed as a tape archive utility, tar couldn't use temporary files. That would require tape drives to constantly back up and erase the temporary files. Backups would take forever! So the commands I now use look something like this:

```
cd ./boel_development
tar -cvf /tmp/boel.tar

cd ../mount_point
tar -xvf /tmp/boel.tar
```

Our final step is to unmount the RAM disk, `umount ./mount_point`, and to squeeze it a bit. We have to use gzip for the squeezing, as that is the compression format recognized by the boot loaders we use. I also make sure to use the -9 option, which gives us the maximum compression possible: `gzip -9 initrd`. And we now have our completed RAM disk.

Automating the Initial RAM Disk Creation Processes

Now that we've been through the complete process of creating the RAM disk, there's a good chance that we'll issue all those commands again and again as we edit the system and test our changes. The SystemImager team has created some scripts to automate this process. You can find these scripts in the ./initrd_source directory in the SystemImager source code, or you can download them directly from the SystemImager CVS tree, here.

Editing Your Initial RAM Disk -- Don't Do It!

When you need to make changes to your RAM disk, it's tempting to just mount it, make the changes, unmount it and be done. But again creeps in our nonzero nemesis. When you edit configuration files, or move things around, you leave behind a lot of nonzero bits. Go ahead and make the changes in your development directory, and use your automation scripts to create your new RAM disk. This will help ensure that you get the smallest possible footprint.

About the author: *Brian Elliott Finley has been working in the computer industry since 1990, specializing in operating systems, system architecture and system integration. In 1993 he found Linux and fell in love. He is active in the Dallas/Fort Worth Linux Users Group, and recently served a year as President. SystemImager is a project he started in 1998 as a way to update large numbers of Solaris machines scattered across the US. Over time it turned into the autoinstall and update tool it is today.*

. . . For further information