ROMFS - ROM File System

This is a quite dumb, read only filesystem, mainly for initial RAM disks of installation disks. It has grown up by the need of having modules linked at boot time. Using this filesystem, you get a very similar feature, and even the possibility of a small kernel, with a file system which doesn't take up useful memory from the router functions in the basement of your office.

For comparison, both the older minix and xiafs (the latter is now defunct) filesystems, compiled as module need more than 20000 bytes, while romfs is less than a page, about 4000 bytes (assuming i586 code). Under the same conditions, the msdos filesystem would need about 30K (and does not support device nodes or symlinks), while the nfs module with nfsroot is about 57K. Furthermore, as a bit unfair comparison, an actual rescue disk used up 3202 blocks with ext2, while with romfs, it needed 3079 blocks.

To create such a file system, you'll need a user program named genromfs. It is available on http://romfs.sourceforge.net/

As the name suggests, romfs could be also used (space-efficiently) on various read-only media, like (E)EPROM disks if someone will have the motivation.. :)

However, the main purpose of romfs is to have a very small kernel, which has only this filesystem linked in, and then can load any module later, with the current module utilities. It can also be used to run some program to decide if you need SCSI devices, and even IDE or floppy drives can be loaded later if you use the "initrd"--initial RAM disk--feature of the kernel. This would not be really news flash, but with romfs, you can even spare off your ext2 or minix or maybe even affs filesystem until you really know that you need it.

For example, a distribution boot disk can contain only the cd disk drivers (and possibly the SCSI drivers), and the ISO 9660 filesystem module. The kernel can be small enough, since it doesn't have other filesystems, like the quite large ext2fs module, which can then be loaded off the CD at a later stage of the installation. Another use would be for a recovery disk, when you are reinstalling a workstation from the network, and you will have all the tools/modules available from a nearby server, so you don't want to carry two disks for this purpose, just because it won't fit into ext2.

romfs operates on block devices as you can expect, and the underlying structure is very simple. Every accessible structure begins on 16 byte boundaries for fast access. The minimum space a file will take is 32 bytes (this is an empty file, with a less than 16 character name). The maximum overhead for any non-empty file is the header, and the 16 byte padding for the name and the contents, also 16+14+15 = 45 bytes. This is quite rare however, since most file names are longer than 3 bytes, and shorter than 15 bytes.

The layout of the filesystem is the following:

```
offset     content

       +---+---+---+---+
 0     | - | r | o | m |  \
       +---+---+---+---+       The ASCII representation of those bytes
 4     | 1 | f | s | - |  /    (i.e. "-rom1fs-")
       +---+---+---+---+
 8     |   full size   |       The number of accessible bytes in this fs.
       +---+---+---+---+
12     |    checksum   |       The checksum of the FIRST 512 BYTES.
       +---+---+---+---+
16     | volume name   |       The zero terminated name of the volume,
       :               :       padded to 16 byte boundary.
       +---+---+---+---+
xx     |     file      |
       :    headers    :
```

Every multi byte value (32 bit words, I'll use the longwords term from now on) must be in big endian order.

The first eight bytes identify the filesystem, even for the casual inspector. After that, in the 3rd longword, it contains the number of bytes accessible from the start of this filesystem. The 4th longword is the checksum of the first 512 bytes (or the number of bytes accessible, whichever is smaller). The applied algorithm is the same as in the AFFS filesystem, namely a simple sum of the longwords (assuming bigendian quantities again). For details, please consult the source. This algorithm was chosen because although it's not quite reliable, it does not require any tables, and it is very simple.

The following bytes are now part of the file system; each file header must begin on a 16 byte boundary:

```
offset     content

       +---+---+---+---+
 0     | next filehdr|X|       The offset of the next file header
       +---+---+---+---+         (zero if no more files)
 4     |   spec.info   |       Info for directories/hard links/devices
       +---+---+---+---+
 8     |     size      |       The size of this file in bytes
       +---+---+---+---+
12     |   checksum    |       Covering the meta data, including the file
       +---+---+---+---+         name, and padding
16     | file name     |       The zero terminated name of the file,
       :               :       padded to 16 byte boundary
```

```
          +---+---+---+---+
    xx    | file data     |
          :               :
```

Since the file headers begin always at a 16 byte boundary, the lowest 4 bits would be always zero in the next filehdr pointer. These four bits are used for the mode information. Bits 0..2 specify the type of the file; while bit 4 shows if the file is executable or not. The permissions are assumed to be world readable, if this bit is not set, and world executable if it is; except the character and block devices, they are never accessible for other than owner. The owner of every file is user and group 0, this should never be a problem for the intended use. The mapping of the 8 possible values to file types is the following:

0	hard link	link destination [file header]
1	directory	first file's header
2	regular file	unused, must be zero [MBZ]
3	symbolic link	unused, MBZ (file data is the link content)
4	block device	16/16 bits major/minor number
5	char device	• " -
6	socket	unused, MBZ
7	fifo	unused, MBZ

Note that hard links are specifically marked in this filesystem, but they will behave as you can expect (i.e. share the inode number). Note also that it is your responsibility to not create hard link loops, and creating all the . and .. links for directories. This is normally done correctly by the genromfs program. Please refrain from using the executable bits for special purposes on the socket and fifo special files, they may have other uses in the future. Additionally, please remember that only regular files, and symlinks are supposed to have a nonzero size field; they contain the number of bytes available directly after the (padded) file name.

Another thing to note is that romfs works on file headers and data aligned to 16 byte boundaries, but most hardware devices and the block device drivers are unable to cope with smaller than block-sized data. To overcome this limitation, the whole size of the file system must be padded to an 1024 byte boundary.

If you have any problems or suggestions concerning this file system, please contact me. However, think twice before wanting me to add features and code, because the primary and most important advantage of this file system is the small code. On the other hand, don't be alarmed, I'm not getting that much romfs related mail. Now I can understand why Avery wrote poems in the ARCnet docs to get some more feedback. :)

romfs has also a mailing list, and to date, it hasn't received any traffic, so you are welcome to join it to discuss your ideas. :)

It's run by ezmlm, so you can subscribe to it by sending a message to romfs-subscribe@shadow.banki.hu, the content is irrelevant.

Pending issues:

- Permissions and owner information are pretty essential features of a Un*x like system, but romfs does not provide the full possibilities. I have never found this limiting, but others might.
- The file system is read only, so it can be very small, but in case one would want to write _anything_ to a file system, he still needs a writable file system, thus negating the size advantages. Possible solutions: implement write access as a compile-time option, or a new, similarly small writable filesystem for RAM disks.
- Since the files are only required to have alignment on a 16 byte boundary, it is currently possibly suboptimal to read or execute files from the filesystem. It might be resolved by reordering file data to have most of it (i.e. except the start and the end) laying at "natural" boundaries, thus it would be possible to directly map a big portion of the file contents to the mm subsystem.
- Compression might be an useful feature, but memory is quite a limiting factor in my eyes.
- Where it is used?
- Does it work on other architectures than intel and motorola?

Have fun,

Janos Farkas <chexum@shadow.banki.hu>