Linux Fu: Fusing Hackaday

Originalartikel

Backup

<html> Unix and, by extension, Linux, has a mantra to make everything possible look like a file. Files, of course, look like files. But also devices, network sockets, and even system information show up as things that appear to be files. There are plenty of advantages to doing that since you can use all the nice tools like

```
grep
```

and

```
find
```

to work with files. However, making your own programs expose a filesystem can be hard. Filesystem code traditionally works at the kernel module level, where mistakes can wipe out lots of things and debugging is difficult. However, there is FUSE — the file system in user space library — that allows you to write more or less ordinary code and expose anything you want as a file system. You’ve probably seen FUSE used to mount, say, remote drives via ssh or Dropbox. We’ve even looked at FUSE before, even for Windows. What’s missing, naturally, is the Hackaday RSS feed, mountable as a normal file. And that’s what we’re building today. Writing a FUSE filesystem isn’t that hard, but there are a lot of tedious jobs. You essentially have to provide callbacks that FUSE uses to do things when the operating system asks for them. Open a file, read a file, list a directory, etc. The problem is that for some simple projects, you don’t care about half of these things, but you still have to provide them. Luckily, there are libraries that can make it a lot easier. I’m going to show you a simple C++ program that can mount your favorite RSS feed (assuming your favorite one is

Hackaday, of course) as a file system. Granted, that’s not amazing, but it is kind of neat to be able to grep through the front page stories from the command line or view the last few articles using Dolphin. <h2>Pick a Library</h2> There are plenty of libraries and wrappers around FUSE. I picked one by [jachappell] over on GitHub. It was pretty simple and hides just enough of FUSE to be handy, but not so much as to be a problem. All the code is hidden around in

```
Fuse.h
```

. One thing to note is that the library assumes you are using

```
libfuse
```

3.0. If you don’t already have it, you’ll have to install the

```
libfuse
```

3.0 development package from your package manager. There are other choices of libraries, of course, and you could just write to the underlying

```
libfuse
```

implementation, but a good library can make it much simpler to get started. Just to keep things simple, I forked the original project on GitHub and added a fusehad directory. <h2>Constraints</h2> To keep things simple, I decided not to worry about performance too much. Since the data is traveling over the network, I do attempt to cache it, and I don’t refresh data later. Of course, you can’t write to the filesystem at all. This is purely for reading Hackaday. These constraints make things easier. Of course, if you were writing your own filesystem, you might relax some of these, but it still helps to get something as simple as possible working first. <h2>Making it Work First</h2> Speaking of which, the first order of business is to be able to read the Hackaday RSS feed and pull out the parts we need. Again, not worrying about performance, I decided to do that with a pipe and calling out to

```
curl
```

. Yes, that’s cheating, but it works just fine, and that’s why we have tools in the first place. The

```
HaDFS.cpp
```

file has a few functions related to FUSE and some helper functions, too. However, I wanted to focus on getting the RSS feed working so I put the related code into a function I made up called

```
userinit
```

. I found out the hard way that naming it

```
init
```

would conflict with the library. The normal FUSE system processes your command line arguments — a good thing, as you’ll see soon. So the main in

```
HaD.cpp
```

is really simple: <pre class=„brush: cpp; title: ; notranslate“ title=„“>#include <stdio.h>#include „HaDFS.h“int main(int argc, char *argv[]){ HaDFS fs; if (fs.userinit()) { fprintf(stderr,„Can't fetch feed\n“); return 99; }; int status; status= fs.run(argc, argv); return status;}</pre> However, for now, I simply commented out the line that calls

```
fs.run
```

. That left me with a simple program that just calls

```
userinit
```

. Reading the feed isn’t that hard since I’m conscripting

```
curl
```

. Each topic is in a structure and there is an array of these structures. If you try to load too many stories, the code just quietly discards the excess (see

```
MAXTOPIC
```

). The

```
topics
```

global variable tells how many stories we’ve actually loaded. <pre class=„brush: cpp; title: ; notranslate“ title=„“> *The curl line to read our feedstatic char cmd[]=„curl https://hackaday.com/feed/ 2>/dev/null | egrep '(<title>|)|(<link>)'“;* User initialization–read the feed (note that init is reserved by the FUSE library)int HaDFS::userinit(void){ FILE *fp; char buf[1024]; *working buffer for reading strings if (!(fp = popen(cmd,„r“))) return 1;* open pipe while (fgets(buf,sizeof(buf),fp)) { string line = buf; line = trimrss(line); *trim off extra stuff if (line.substr(0,7) == „<title>“)* identify line type and process { topic[topics].title = line.substr(7); topic[topics].title += „.html“; } else if (line.substr(0,6)==„<link>“) { topic[topics].url = line.substr(6); topics++; if (topics == MAXTOPIC) break; *quietly truncate a long feed } } pclose(fp); return 0;} </pre> The <code>popen</code> function runs a command line and gives us the <code>stdout</code> stream as a bunch of lines. Processing the lines is just brute force looking for <title> and <link> to identify the data we need. I filtered <code>curl</code> through <code>grep</code> to make sure I didn’t get a lot of extra lines, by the way, and I assumed lowercase, but a <code>-i</code> option could easily fix that. The redirect is to prevent <code>curl</code> from polluting <code>stderr</code>, although normally FUSE will disconnect the output streams so it doesn’t really matter. Note that I add an HTML extension to each fake file name so opening one is more likely to get to the browser. By putting a <code>printf</code> in the code I was able to make sure the feed fetching was working the way I expected. Note that I don’t fetch the actual pages until later in the process. For now, I just

*want the titles and the URL links. <h2>The Four Functions</h2> There are four functions we need to create in a subclass to get a minimal read-only filesystem going: <code>getattr</code>, <code>readdir</code>, <code>open</code>, and <code>read</code>. These functions pretty much do what you expect. The <code>getattr</code> call will return 755 for our root (and only) directory and 444 for any other file that exists. The <code>readdir</code> outputs entries for . and .. along with our “files.” <code>Open</code> and <code>read</code> do just what you think they do. There are some other functions, but those are ones I added to help myself: <code>userinit</code> – Called to kick off the file system data
<code>trimrss</code> – Trim an RSS line to make it easier to parse
<code>pathfind</code> – Turn a file name into a descriptor (an index into the array of topics)
<code>readurl</code> – Return a string with the contents of a URL (uses curl) There’s not much to it. You’ll see in the code that there are a few things to look out for like catching someone trying to write to a file since that isn’t allowed. <h2>Debugging and Command Line Options</h2> Of course, it doesn’t matter how simple it is, it isn’t going to work the first time is it? Of course, first, you have to remember to put the call to <code>fs.run</code> back in the main function. But, of course, things won’t work like you expect for any of a number of reasons. There are a few things to remember as you go about running and debugging. When you build your executable, you simply run it and provide a command line argument to specify the mount point which, of course, should exist. I have a habit of using <code>/tmp/mnt</code> while debugging, but it can be anywhere you have permissions. Under normal operation, FUSE detaches your program so you can’t just kill it. You’ll need to use unmount command (<code>fusermount -u</code>) with the mount point as an argument. Even if your program dies with a segment fault, you’ll need to use the unmount command or you will probably get the dreaded “disconnected endpoint” error message. Being detached leads to a problem. If you put <code>printf</code> statements in your code, they will never show up after detachment. For this reason, FUSE understands the <code>-f</code> flag which tells the system to keep your filesystem running in the foreground. Then you can see messages and a clean exit, like a Control+C, will cleanly unmount the filesystem. You can also use <code>-d</code> which enables some built-in debugging and implies <code>-f</code>. The <code>-s</code> flag turns off threading which can make debugging easier, or harder if you are dealing with a thread-related problem. You can use <code>gdb</code>, and there are some good articles about that. But for such a simple piece of code, it isn’t really necessary. <h2>What’s Next?</h2> The documentation for the library is almost nothing. However, the library closely mirrors the <code>libfuse</code> API so the documentation for that (mostly in fuse.h) will help you go further. If you want to graduate from FUSE to a “real” file system, you have a long road. The video below gives some background on Linux VFS, but that’s just the start down that path. <iframe class=„youtube-player c1“ width=„800“ height=„480“ src=„https://www.youtube.com/embed/J4qWNNISdJk?version=3&rel=1&showsearch=0&showinfo=1&iv_load_policy=1&fs=1&hl=en-US&autohide=2&wmode=transparent“ allowfullscreen=„allowfullscreen“ sandbox=„allow-scripts allow-same-origin allow-popups allow-presentation“>[embedded content]</iframe> Maybe stick to FUSE for a while. If you prefer Python, no problem. FUSE is very popular for mapping cloud storage into your*

filesystem, but with your own coding, you could just as easily expose your Arduino or anything else your computer can communicate with.</p> </html>