prashants / lwnfs.c

Created 10 years ago

Updated lwnfs

lwnfs.c

```c
/*
 * Demonstrate a trivial filesystem using libfs.
 *
 * Copyright 2002, 2003 Jonathan Corbet <corbet@lwn.net>
 * This file may be redistributed under the terms of the GNU GPL.
 *
 * Chances are that this code will crash your system, delete your
 * nethack high scores, and set your disk drives on fire.  You have
 * been warned.
 */
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/pagemap.h>      /* PAGE_CACHE_SIZE */
#include <linux/fs.h>           /* This is where libfs stuff is declared */
#include <asm/atomic.h>
#include <asm/uaccess.h>        /* copy_to_user */

/*
 * Boilerplate stuff.
 */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jonathan Corbet");

#define LFS_MAGIC 0x19980122


/*
 * Anytime we make a file or directory in our filesystem we need to
 * come up with an inode to represent it internally.  This is
 * the function that does that job.  All that's really interesting
```

```
 * is the "mode" parameter, which says whether this is a directory
 * or file, and gives the permissions.
 */
static struct inode *lfs_make_inode(struct super_block *sb, int mode)
{
        struct inode *ret = new_inode(sb);

        if (ret) {
                ret->i_mode = mode;
                ret->i_uid = ret->i_gid = 0;
                ret->i_blocks = 0;
                ret->i_atime = ret->i_mtime = ret->i_ctime = CURRENT_TIME;
        }
        return ret;
}


/*
 * The operations on our "files".
 */

/*
 * Open a file.  All we have to do here is to copy over a
 * copy of the counter pointer so it's easier to get at.
 */
static int lfs_open(struct inode *inode, struct file *filp)
{
        filp->private_data = inode->i_private;
        return 0;
}

#define TMPSIZE 20
/*
 * Read a file.  Here we increment and read the counter, then pass it
 * back to the caller.  The increment only happens if the read is done
 * at the beginning of the file (offset = 0); otherwise we end up counting
 * by twos.
 */
static ssize_t lfs_read_file(struct file *filp, char *buf,
                size_t count, loff_t *offset)
{
        atomic_t *counter = (atomic_t *) filp->private_data;
        int v, len;
        char tmp[TMPSIZE];
/*
 * Encode the value, and figure out how much of it we can pass back.
 */
        v = atomic_read(counter);
        if (*offset > 0)
```

```
81                  v -= 1;  /* the value returned when offset was zero */
82          else
83                  atomic_inc(counter);
84          len = snprintf(tmp, TMPSIZE, "%d\n", v);
85          if (*offset > len)
86                  return 0;
87          if (count > len - *offset)
88                  count = len - *offset;
89  /*
90   * Copy it back, increment the offset, and we're done.
91   */
92          if (copy_to_user(buf, tmp + *offset, count))
93                  return -EFAULT;
94          *offset += count;
95          return count;
96  }
97
98  /*
99   * Write a file.
100  */
101 static ssize_t lfs_write_file(struct file *filp, const char *buf,
102                 size_t count, loff_t *offset)
103 {
104         atomic_t *counter = (atomic_t *) filp->private_data;
105         char tmp[TMPSIZE];
106 /*
107  * Only write from the beginning.
108  */
109         if (*offset != 0)
110                 return -EINVAL;
111 /*
112  * Read the value from the user.
113  */
114         if (count >= TMPSIZE)
115                 return -EINVAL;
116         memset(tmp, 0, TMPSIZE);
117         if (copy_from_user(tmp, buf, count))
118                 return -EFAULT;
119 /*
120  * Store it in the counter and we are done.
121  */
122         atomic_set(counter, simple_strtol(tmp, NULL, 10));
123         return count;
124 }
125
126
127 /*
128  * Now we can put together our file operations structure.
129  */
```

```
130 static struct file_operations lfs_file_ops = {
131         .open   = lfs_open,
132         .read   = lfs_read_file,
133         .write  = lfs_write_file,
134 };
135
136
137 /*
138  * Create a file mapping a name to a counter.
139  */
140 static struct dentry *lfs_create_file (struct super_block *sb,
141                 struct dentry *dir, const char *name,
142                 atomic_t *counter)
143 {
144         struct dentry *dentry;
145         struct inode *inode;
146         struct qstr qname;
147 /*
148  * Make a hashed version of the name to go with the dentry.
149  */
150         qname.name = name;
151         qname.len = strlen (name);
152         qname.hash = full_name_hash(name, qname.len);
153 /*
154  * Now we can create our dentry and the inode to go with it.
155  */
156         dentry = d_alloc(dir, &qname);
157         if (! dentry)
158                 goto out;
159         inode = lfs_make_inode(sb, S_IFREG | 0644);
160         if (! inode)
161                 goto out_dput;
162         inode->i_fop = &lfs_file_ops;
163         inode->i_private = counter;
164 /*
165  * Put it all into the dentry cache and we're done.
166  */
167         d_add(dentry, inode);
168         return dentry;
169 /*
170  * Then again, maybe it didn't work.
171  */
172   out_dput:
173         dput(dentry);
174   out:
175         return 0;
176 }
177
178
```

```
179 /*
180  * Create a directory which can be used to hold files.  This code is
181  * almost identical to the "create file" logic, except that we create
182  * the inode with a different mode, and use the libfs "simple" operations.
183  */
184 static struct dentry *lfs_create_dir (struct super_block *sb,
185                 struct dentry *parent, const char *name)
186 {
187         struct dentry *dentry;
188         struct inode *inode;
189         struct qstr qname;
190
191         qname.name = name;
192         qname.len = strlen (name);
193         qname.hash = full_name_hash(name, qname.len);
194         dentry = d_alloc(parent, &qname);
195         if (! dentry)
196                 goto out;
197
198         inode = lfs_make_inode(sb, S_IFDIR | 0644);
199         if (! inode)
200                 goto out_dput;
201         inode->i_op = &simple_dir_inode_operations;
202         inode->i_fop = &simple_dir_operations;
203
204         d_add(dentry, inode);
205         return dentry;
206
207   out_dput:
208         dput(dentry);
209   out:
210         return 0;
211 }
212
213
214
215 /*
216  * OK, create the files that we export.
217  */
218 static atomic_t counter, subcounter;
219
220 static void lfs_create_files (struct super_block *sb, struct dentry *root)
221 {
222         struct dentry *subdir;
223 /*
224  * One counter in the top-level directory.
225  */
226         atomic_set(&counter, 0);
227         lfs_create_file(sb, root, "counter", &counter);
```

```
228 /*
229  * And one in a subdirectory.
230  */
231         atomic_set(&subcounter, 0);
232         subdir = lfs_create_dir(sb, root, "subdir");
233         if (subdir)
234                 lfs_create_file(sb, subdir, "subcounter", &subcounter);
235 }
236
237
238
239 /*
240  * Superblock stuff.  This is all boilerplate to give the vfs something
241  * that looks like a filesystem to work with.
242  */
243
244 /*
245  * Our superblock operations, both of which are generic kernel ops
246  * that we don't have to write ourselves.
247  */
248 static struct super_operations lfs_s_ops = {
249         .statfs         = simple_statfs,
250         .drop_inode     = generic_delete_inode,
251 };
252
253 /*
254  * "Fill" a superblock with mundane stuff.
255  */
256 static int lfs_fill_super (struct super_block *sb, void *data, int silent)
257 {
258         struct inode *root;
259         struct dentry *root_dentry;
260 /*
261  * Basic parameters.
262  */
263         sb->s_blocksize = PAGE_CACHE_SIZE;
264         sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
265         sb->s_magic = LFS_MAGIC;
266         sb->s_op = &lfs_s_ops;
267 /*
268  * We need to conjure up an inode to represent the root directory
269  * of this filesystem.  Its operations all come from libfs, so we
270  * don't have to mess with actually *doing* things inside this
271  * directory.
272  */
273         root = lfs_make_inode (sb, S_IFDIR | 0755);
274         if (! root)
275                 goto out;
276         root->i_op = &simple_dir_inode_operations;
```

```
277         root->i_fop = &simple_dir_operations;
278 /*
279  * Get a dentry to represent the directory in core.
280  */
281         root_dentry = d_alloc_root(root);
282         if (! root_dentry)
283                 goto out_iput;
284         sb->s_root = root_dentry;
285 /*
286  * Make up the files which will be in this filesystem, and we're done.
287  */
288         lfs_create_files (sb, root_dentry);
289         return 0;
290
291   out_iput:
292         iput(root);
293   out:
294         return -ENOMEM;
295 }
296
297
298 /*
299  * Stuff to pass in when registering the filesystem.
300  */
301 static struct dentry *lfs_get_super(struct file_system_type *fst,
302                 int flags, const char *devname, void *data)
303 {
304         return mount_bdev(fst, flags, devname, data, lfs_fill_super);
305 }
306
307 static struct file_system_type lfs_type = {
308         .owner          = THIS_MODULE,
309         .name           = "lwnfs",
310         .mount          = lfs_get_super,
311         .kill_sb        = kill_litter_super,
312 };
313
314
315
316
317 /*
318  * Get things set up.
319  */
320 static int __init lfs_init(void)
321 {
322         return register_filesystem(&lfs_type);
323 }
324
325 static void __exit lfs_exit(void)
```

```
326 {
327         unregister_filesystem(&lfs_type);
328 }
329
330 module_init(lfs_init);
331 module_exit(lfs_exit);
```

mbaynton commented on Jan 28, 2014

Try my fork if this code produces an error when mounting

RadNi commented on Jan 10, 2019

For kernel version up to 4.15 use my fork.