FUSE
Filesystem in User space

Chian yu Tseng
2012-6-11

Outline

- Introduction
- The FUSE structure
- 如何運作
- Struct fuse_operations
- Example

Introduction(1/2)

- FUSE is a loadable kernel module for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code.
- This is achieved by running file system code in user space while the FUSE module provides only a "**bridge**" to the actual kernel interfaces.

- FUSE is particularly useful for writing **virtual file systems**. Unlike traditional file systems that essentially save data to and retrieve data from disk, virtual filesystems do not actually store data themselves. They act as a view or translation of an existing file system or storage device.

Introduction(2/2)

- The FUSE system was originally part of A Virtual Filesystem (AVFS), but has since split off into its own project on SourceForge.net.

- FUSE is available for Linux, FreeBSD, NetBSD, OpenSolaris, and Mac OS X. It was officially merged into the mainstream Linux kernel tree in kernel version 2.6.14.

Examples(1/2)

- **ExpanDrive**: A commercial filesystem implementing SFTP/FTP/FTPS using FUSE.
- **GlusterFS**: Clustered Distributed Filesystem having capability to scale up to several petabytes.
- **SSHFS**: Provides access to a remote filesystem through SSH.
- **GmailFS**: Filesystem which stores data as mail in Gmail
- **EncFS**: Encrypted virtual filesystem

Examples(2/2)

- **NTFS-3G**和**Captive NTFS**: allowing access to NTFS filesystem.
- **WikipediaFS** : View and edit Wikipedia articles as if they were real files.
- Sun Microsystems's **Lustre** cluster filesystem
- Sun Microsystems's **ZFS**
- **HDFS**: FUSE bindings exist for the open source Hadoop distributed filesystem.

FUSE Installation

- http://fuse.sourceforge.net/

- ./configure
- make
- make install

FUSE source code

- **./doc**: contains FUSE-related documentation. Ex: **how-fuse-works**
- **./include**: contains the FUSE API headers, which you need to create a file system. The only one you need now is **fuse.h**.
- **./lib**: holds the source code to create the FUSE libraries that you will be linking with your binaries to create a file system.
- **./util**: has the source code for the FUSE utility library.
- **./example:** contains samples for your reference.

FUSE structure

- FUSE kernel module (fuse.ko)
  - inode.c, dev.c, control.c, dir.c, file.c
- LibFUSE module (libfuse.*)
  - helper.c, fuse_kern_chan.c, fuse_mt.c, fuse.c, fuse_lowlevel.c, fuse_loop.c, fuse_loop_mt.c, fuse_session.c
- Mount utility(fusermount)
  - fusermount, mount.fuse.c, mount_util.c, mount.c, mount_bsd.c,

FUSE Library

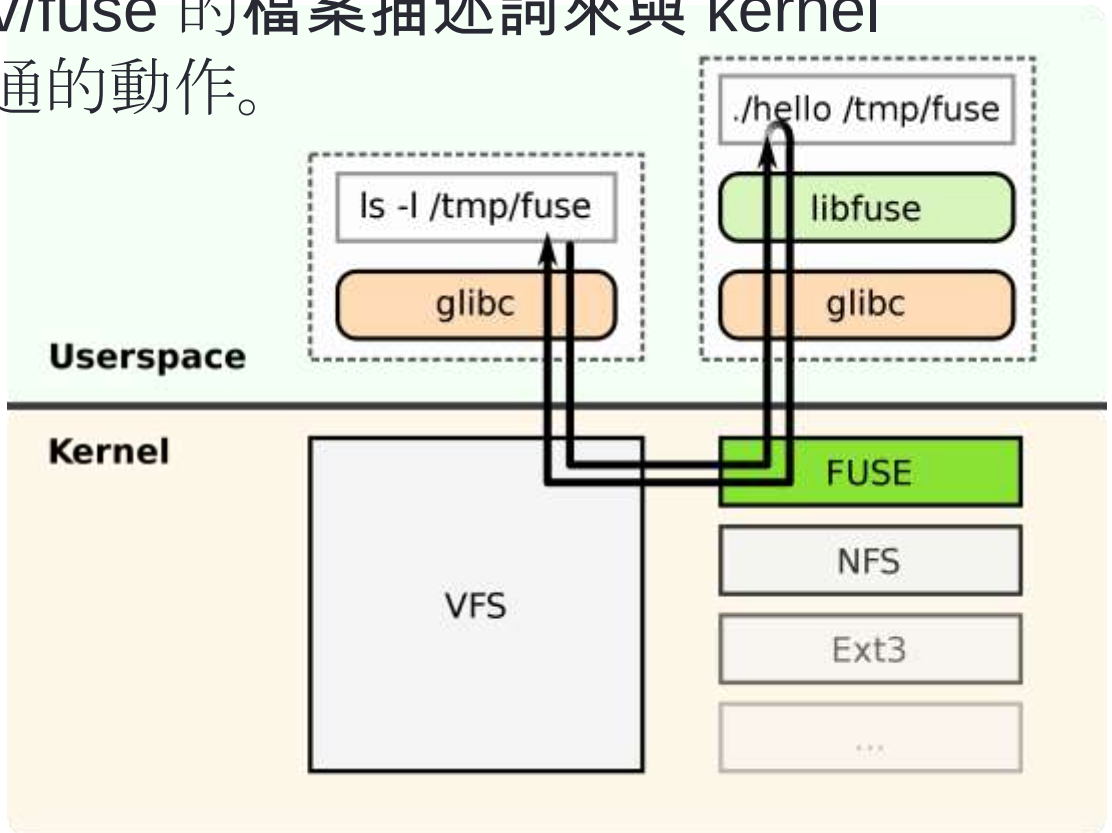
- include/fuse.h → the library interface of FUSE（HighLevel）
- include/fuse_common.h → common
- include/fuse_lowlevel.h → Lowlevel API
- include/fuse_opt.h → option parsing interface of FUSE

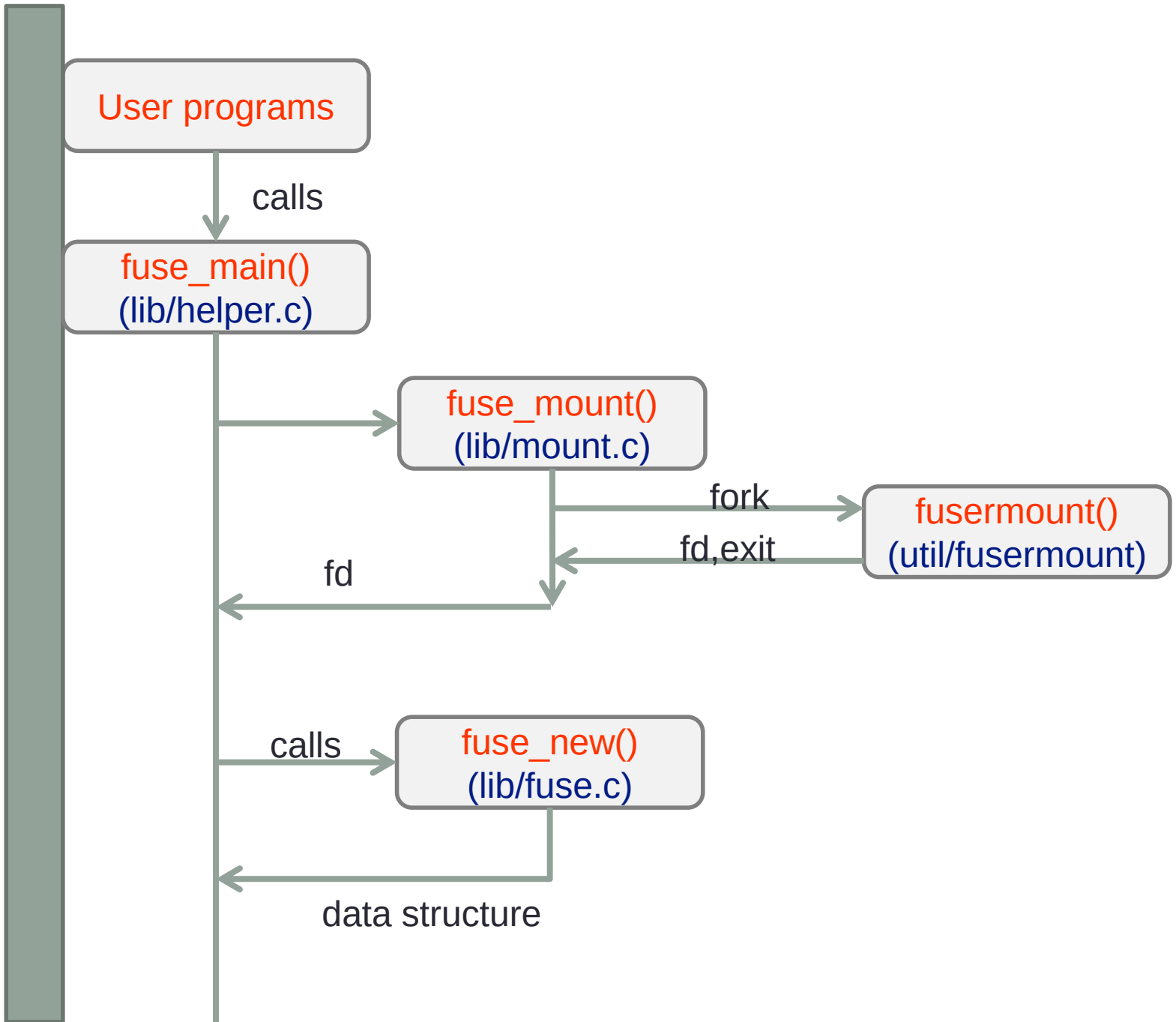
如何運作

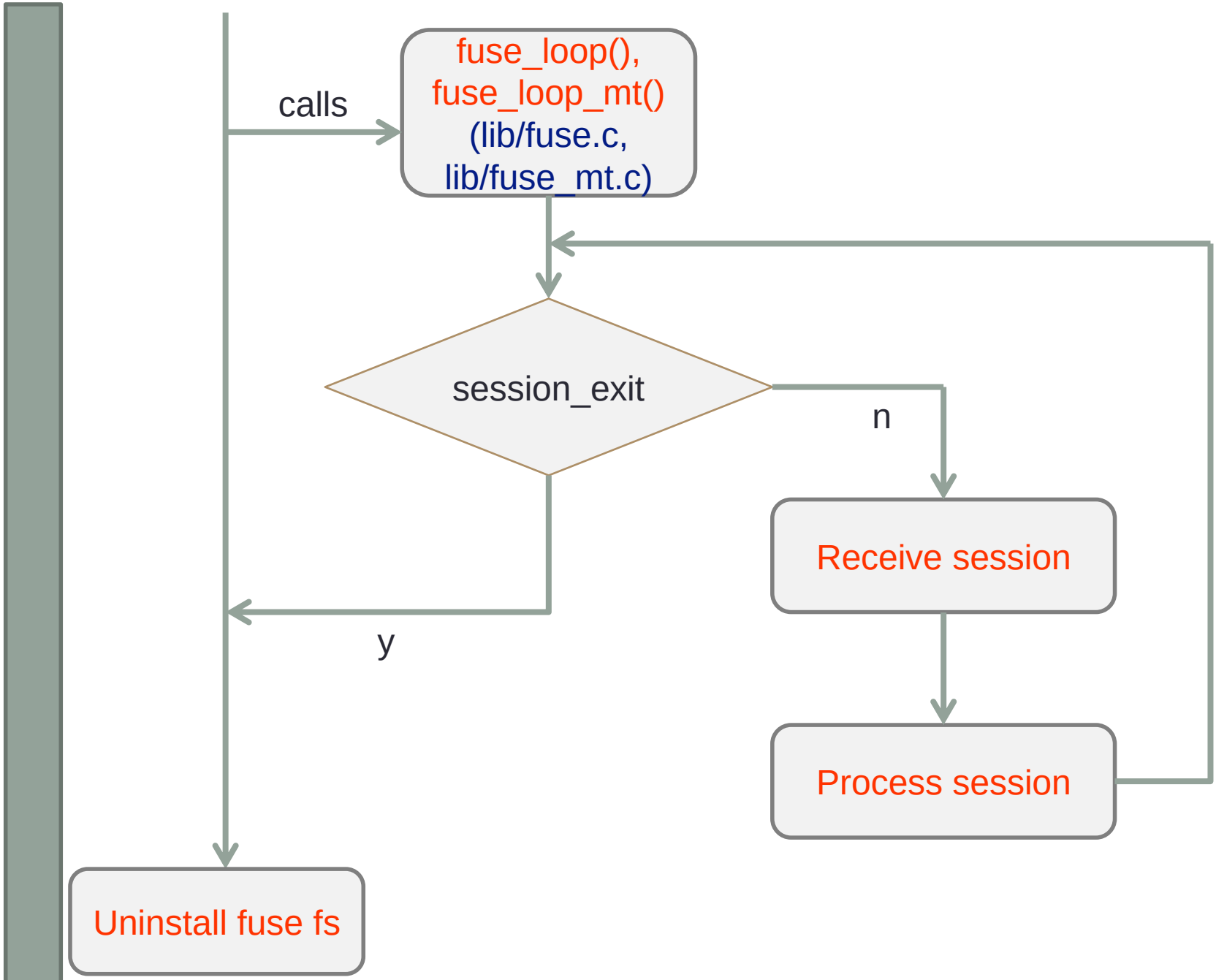
- 在 FUSE daemon 啟動的時候，會先進行掛載的動作，將 /dev/fuse 掛載到指定的目錄底下，並回傳/dev/fuse 的檔案描述詞(file descriptor)，而 FUSE daemon 在預設上會使用 multi-thread 的方式，透過/dev/fuse 的檔案描述詞來接收requests，再根據 requests 的類別來進行處理，最後透過 replies，將結果傳回去。

如何運作

- ls : FUSE daemon會接收到 OPENDIR、READDIR 等 requests, 並採用 userspace library(libfuse.*)的函式, 讀取 file 目錄的資訊，並將此資訊傳回去，其中 FUSE daemon 就是透過/dev/fuse 的檔案描述詞來與 kernel module(fuse.ko)作溝通的動作。




User programs
calls
fuse_main()
(lib/helper.c)
fuse_mount()
(lib/mount.c)
fork
fusermount()
(util/fusermount)
fd,exit
fd
calls
fuse_new()
(lib/fuse.c)
data structure

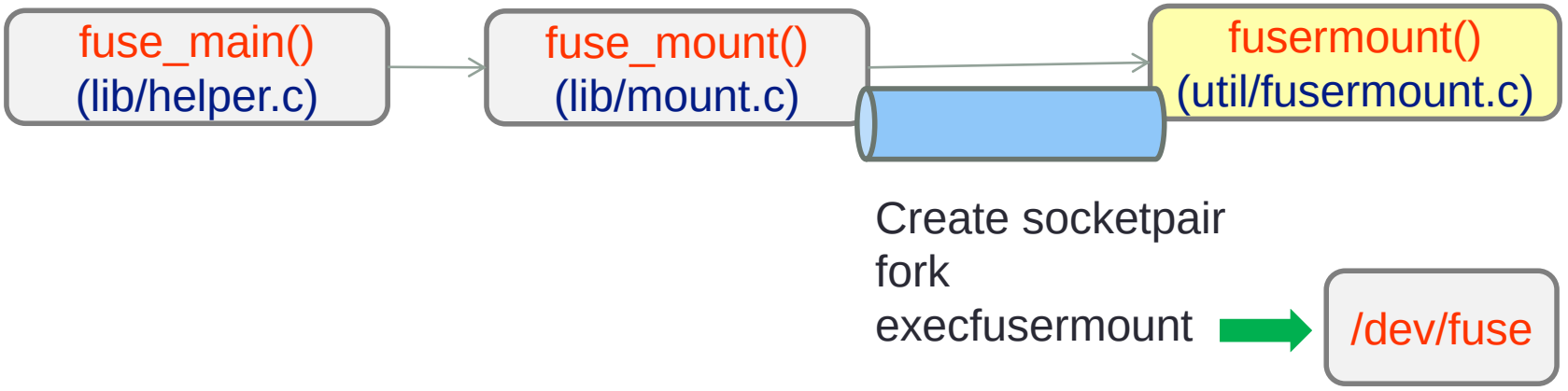

calls
fuse_loop(),
fuse_loop_mt()
(lib/fuse.c,
lib/fuse_mt.c)
session_exit
n
y
Receive session
Process session
Uninstall fuse fs

The fuse library(1/5)

- When your user mode program calls fuse_main() (lib/helper.c),fuse_main() parses the arguments passed to your user mode program, then calls fuse_mount() (lib/mount.c).

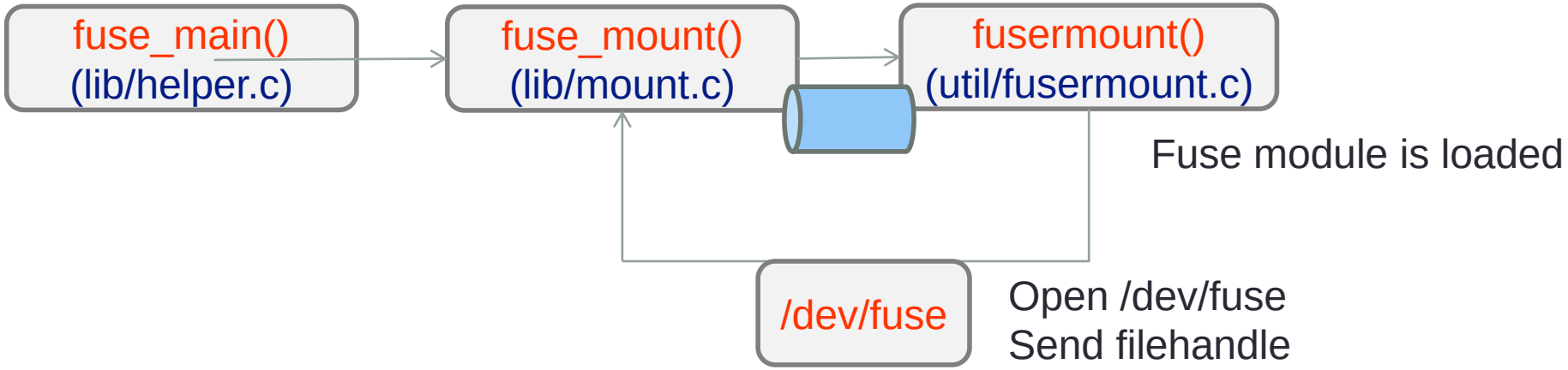
The fuse library(2/5)

- fuse_mount() creates a UNIX domain socket pair, then forks and execsfusermount (util/fusermount.c) passing it one end of the socket in the FUSE_COMMFD_ENV environment variable.



The fuse library(3/5)

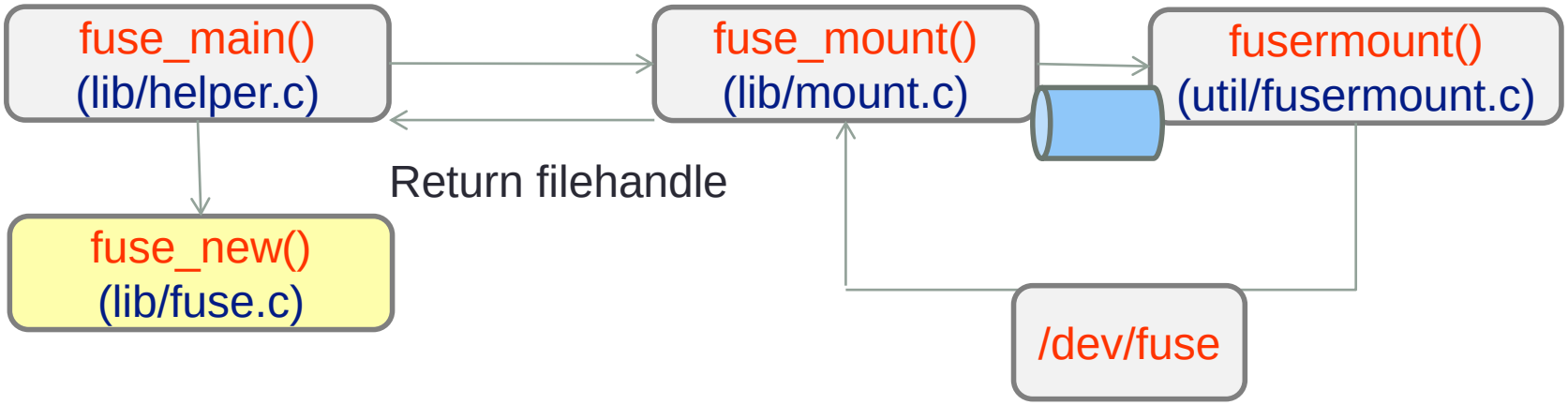
- fusermount (util/fusermount.c) makes sure that the fuse module is loaded. fusermount then open /dev/fuse and send the file handle over a UNIX domain socket back to fuse_mount().



The fuse library(4/5)

- fuse_mount() returns the file handle for /dev/fuse to fuse_main().

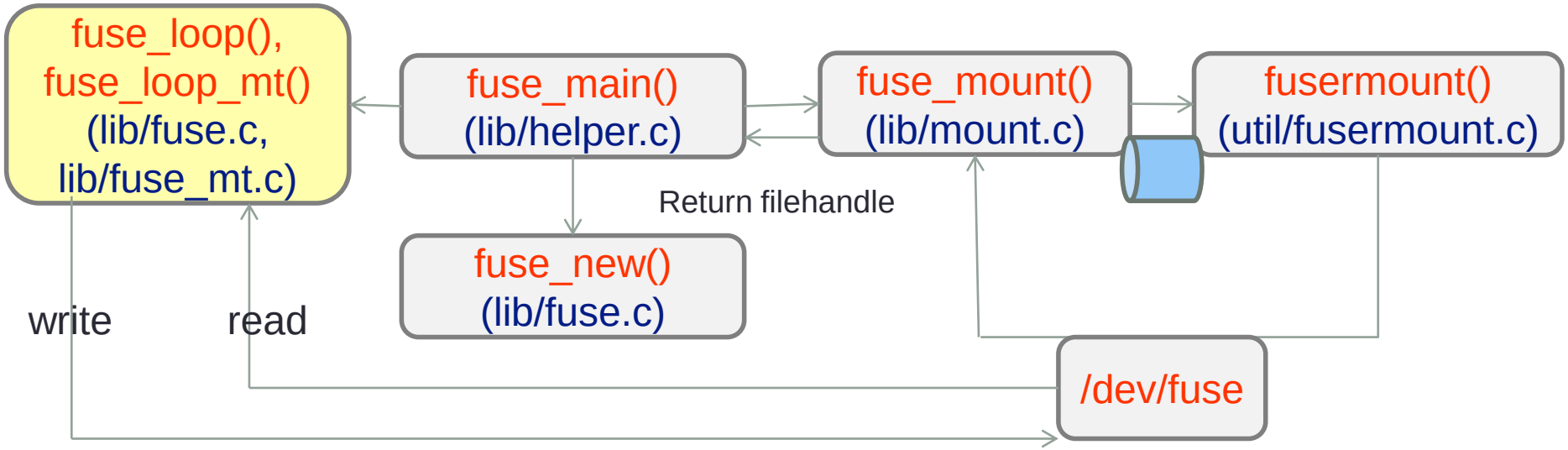
- fuse_main() calls fuse_new() (lib/fuse.c) which allocates the struct fuse data structure that stores and maintains a cached image of the filesystem data.



The fuse library (5/5)

- Lastly, fuse_main() calls either fuse_loop() (lib/fuse.c) or fuse_loop_mt() (lib/fuse_mt.c) which both start to read the file system system calls from the /dev/fuse, call the user mode functions stored in **struct fuse_operations** data structure before calling fuse_main().

- The results of those calls are then written back to the /dev/fuse file where they can be forwarded back to the system calls.



Struct fuse_operations (1/9)

- int (*getattr) (const char *, struct stat *);
  - Get file attributes.
- int (*readlink) (const char *, char *, size_t);
  - Read the target of a symbolic link
- int (*mknod) (const char *, mode_t, dev_t);
  - Create a file node.
- int (*mkdir) (const char *, mode_t);
  - Create a directory. Note that the mode argument may not have the type specification bits set, i.e. S_ISDIR(mode) can be false. To obtain the correct directory type bits use mode | S_IFDIR

Struct fuse_operations (2/9)

- int (*unlink) (const char *);
  - Remove a file
- int (*rmdir) (const char *);
  - Remove a directory
- int (*symlink) (const char *, const char *);
  - Create a symbolic link
- int (*rename) (const char *, const char *);
  - Rename a file
- int (*link) (const char *, const char *);
  - Create a hard link to a file

Struct fuse_operations (3/9)

- int (*chmod) (const char *, mode_t);
  - Change the permission bits of a file
- int (*chown) (const char *, uid_t, gid_t);
  - Change the owner and group of a file
- int (*truncate) (const char *, off_t);
  - Change the size of a file
- int (*open) (const char *, struct fuse_file_info *);
  - File open operation.

Struct fuse_operations (4/9)

- int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
  - Read data from an open file.
- int (*write) (const char *, const char *, size_t, off_t, struct fuse_file_info *);
  - Write data to an open file
- int (*statfs) (const char *, struct statvfs *);
  - Get file system statistics
- int (*flush) (const char *, struct fuse_file_info *);
  - Possibly flush cached data

Struct fuse_operations (5/9)

- int (*release) (const char *, struct fuse_file_info *);
  - Release an open file. Release is called when there are no more references to an open file: all file descriptors are closed and all memory mappings are unmapped.
- int (*fsync) (const char *, int, struct fuse_file_info *);
  - Synchronize file contents
- int (*setxattr) (const char *, const char *, const char *, size_t, int);
  - Set extended attributes
- int (*getxattr) (const char *, const char *, char *, size_t);
  - Get extended attributes

Struct fuse_operations (6/9)

- int (*listxattr) (const char *, char *, size_t);
  - List extended attributes
- int (*removexattr) (const char *, const char *);
  - Remove extended attributes

- int (*opendir) (const char *, struct fuse_file_info *);
  - Open directory. Unless the 'default_permissions' mount option is given, this method should check if opendir is permitted for this directory. Optionally opendir may also return an arbitrary **filehandle** in the **fuse_file_info** structure, which will be passed to readdir, closedir and fsyncdir.

Struct fuse_operations (7/9)

- int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *);
  - Read directory
- int (*releasedir) (const char *, struct fuse_file_info *);
  - Release directory
- int (*fsyncdir) (const char *, int, struct fuse_file_info *);
  - Synchronize directory contents

- void *(*init) (struct fuse_conn_info *conn);
  - Initialize file system.

Struct fuse_operations (8/9)

- void (*destroy) (void *);
  - Clean up filesystem
- int (*access) (const char *, int);
  - Check file access permissions
- int (*create) (const char *, mode_t, struct fuse_file_info *);
  - Create and open a file. If the file does not exist, first create it with the specified mode, and then open it.
- int (*ftruncate) (const char *, off_t, struct fuse_file_info *);
  - Change the size of an open file
- int (*fgetattr) (const char *, struct stat *, struct fuse_file_info *);
  - Get attributes from an open file

Struct fuse_operations(9/9)

- int (*lock) (const char *, struct fuse_file_info *, int cmd, struct flock *);
  - Perform POSIX file locking operation
- int (*utimens) (const char *, const struct timespec tv[2]);
  - Change the access and modification times of a file with nanosecond resolution
- int (*bmap) (const char *, size_t blocksize, uint64_t *idx);
  - Map block index within file to block index within device

Example1: Hello.c

```
11  #define FUSE_USE_VERSION 26
12
13  #include <fuse.h>
14  #include <stdio.h>
15  #include <string.h>
16  #include <errno.h>
17  #include <fcntl.h>
18
19  static const char *hello_str = "Hello World!\n";
20  static const char *hello_path = "/hello";
21
```

```
86  static struct fuse_operations hello_oper = {
87      .getattr    = hello_getattr,
88      .readdir    = hello_readdir,
89      .open       = hello_open,
90      .read       = hello_read,
91  };
92
93  int main(int argc, char *argv[])
94  {
95      return fuse_main(argc, argv, &hello_oper, NULL);
96  }
97
```

hello-getattr()

```
22  static int hello_getattr(const char *path, struct stat *stbuf)
23  {
24      int res = 0;
25
26      memset(stbuf, 0, sizeof(struct stat));
27      if (strcmp(path, "/") == 0) {
28          stbuf->st_mode = S_IFDIR | 0755;
29          stbuf->st_nlink = 2;
30      } else if (strcmp(path, hello_path) == 0) {
31          stbuf->st_mode = S_IFREG | 0444;
32          stbuf->st_nlink = 1;
33          stbuf->st_size = strlen(hello_str);
34      } else
35          res = -ENOENT;   A component of the path path does not exis
36
37      return res;
38  }
```

```
danny@danny-desktop:/tmp$ ls -l
total 8
drwxr-xr-x 2 root root    0 1970-01-01 08:00 fuse
drwx------ 2 gdm  gdm  4096 2012-05-22 11:41 orbit-gdm
drwx------ 2 gdm  gdm  4096 2012-05-22 11:11 pulse-PKdhtXMmr18n
danny@danny-desktop:/tmp$ ls -l fuse/
total 0
-r--r--r-- 1 root root 13 1970-01-01 08:00 hello
```

hello_readdir()

```
40  static int hello_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
41                off_t offset, struct fuse_file_info *fi)
42  {
43      (void) offset;
44      (void) fi;
45
46      if (strcmp(path, "/") != 0)
47          return -ENOENT;
48
49      filler(buf, ".", NULL, 0);
50      filler(buf, "..", NULL, 0);
51      filler(buf, hello_path + 1, NULL, 0);
52
53      return 0;
54  }
```

typedef int(* fuse_fill_dir_t)(void *buf, const char *name, const struct stat *stbuf, off_t off)

Function to add an entry in a readdir() operation

Parameters:

buf	the buffer passed to the readdir() operation
name	the file name of the directory entry
stat	file attributes, can be NULL
off	offset of the next entry or zero

Returns:

1 if buffer is full, zero otherwise

hello_open()

- This function checks whatever user is ***permitted*** to open the /hello file with flags given in the fuse_file_info structure.

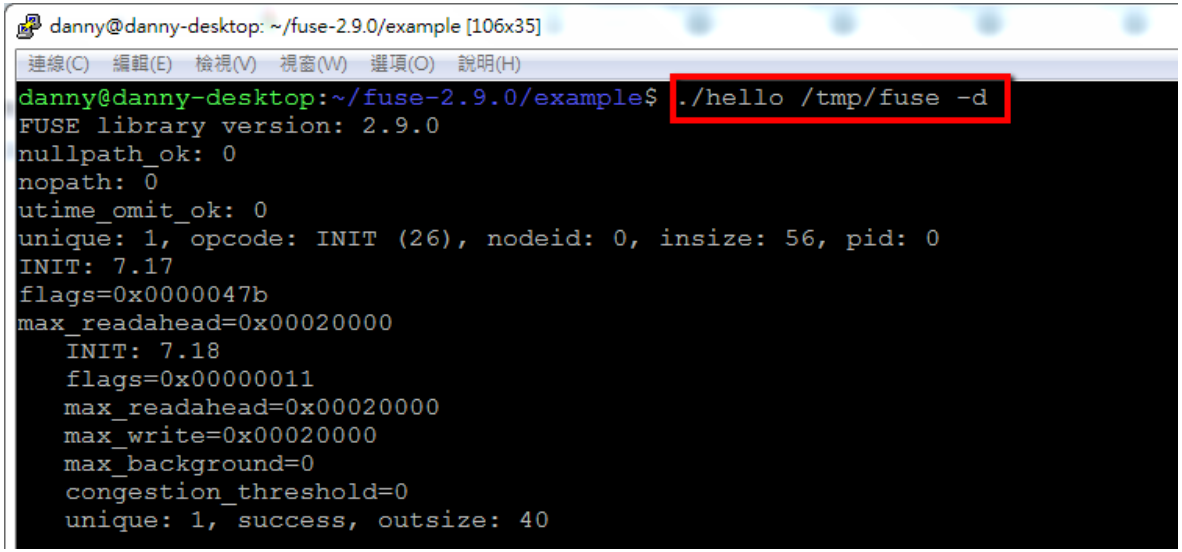
```
56  static int hello_open(const char *path, struct fuse_file_info *fi)
57  {
58      if (strcmp(path, hello_path) != 0)
59          return -ENOENT;
60
61      if ((fi->flags & 3) != O_RDONLY)
62          return -EACCES;
63
64      return 0;
65  }
```

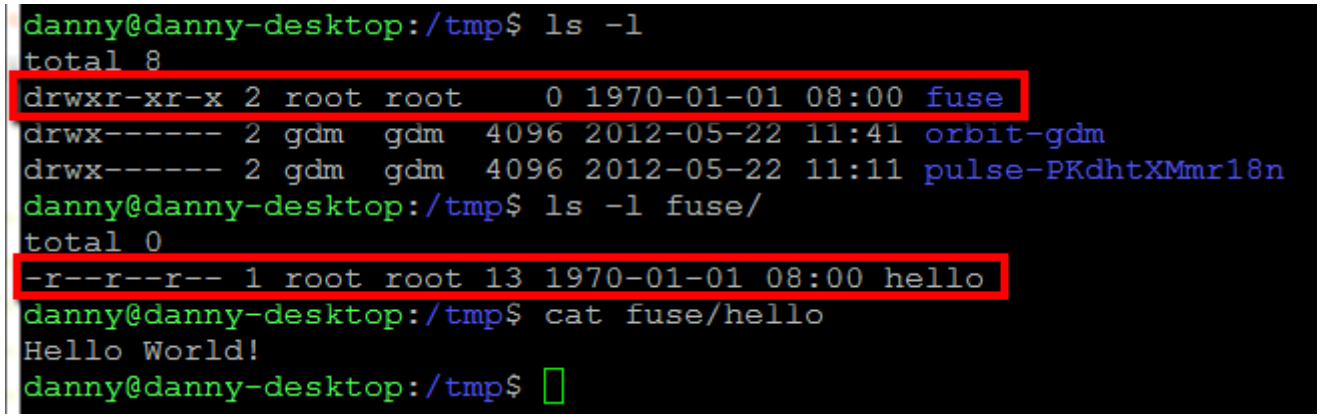
hello_read()

```
67  static int hello_read(const char *path, char *buf, size_t size, off_t offset,
68                        struct fuse_file_info *fi)
69  {
70      size_t len;
71      (void) fi;
72      if(strcmp(path, hello_path) != 0)
73          return -ENOENT;
74
75      len = strlen(hello_str);
76      if (offset < len) {
77          if (offset + size > len)
78              size = len - offset;
79          memcpy(buf, hello_str + offset, size);
80      } else
81          size = 0;
82
83      return size;
84  }
```

Example1: Hello.c 執行

- ./hello /tmp/fuse -d





Example2: fusexmp_fh.c

```
500 static struct fuse_operations xmp_oper = {
501     .getattr    = xmp_getattr,
502     .fgetattr   = xmp_fgetattr,
503     .access     = xmp_access,
504     .readlink   = xmp_readlink,
505     .opendir    = xmp_opendir,
506     .readdir    = xmp_readdir,
507     .releasedir = xmp_releasedir,
508     .mknod      = xmp_mknod,
509     .mkdir      = xmp_mkdir,
510     .symlink    = xmp_symlink,
511     .unlink     = xmp_unlink,
512     .rmdir      = xmp_rmdir,
513     .rename     = xmp_rename,
514     .link       = xmp_link,
515     .chmod      = xmp_chmod,
516     .chown      = xmp_chown,
517     .truncate   = xmp_truncate,
518     .ftruncate  = xmp_ftruncate,
519 #ifdef HAVE_UTIMENSAT
520     .utimens    = xmp_utimens,
521 #endif
522     .create     = xmp_create,
523     .open       = xmp_open,
524     .read       = xmp_read,
525     .read_buf   = xmp_read_buf,
526     .write      = xmp_write,
```

```
526     .write      = xmp_write,
527     .write_buf  = xmp_write_buf,
528     .statfs     = xmp_statfs,
529     .flush      = xmp_flush,
530     .release    = xmp_release,
531     .fsync      = xmp_fsync,
532 #ifdef HAVE_SETXATTR
533     .setxattr   = xmp_setxattr,
534     .getxattr   = xmp_getxattr,
535     .listxattr  = xmp_listxattr,
536     .removexattr    = xmp_removexattr,
537 #endif
538     .lock       = xmp_lock,
539     .flock      = xmp_flock,
540
541     .flag_nullpath_ok = 1,
542 #if HAVE_UTIMENSAT
543     .flag_utime_omit_ok = 1,
544 #endif
545 };
546
547 int main(int argc, char *argv[])
548 {
549     umask(0);
550     return fuse_main(argc, argv, &xmp_oper, NULL);
551 }
552
```

xmp_getattr(), xmp_fgetattr()

```
36  static int xmp_getattr(const char *path, struct stat *stbuf)
37  {
38      int res;
39
40      res = lstat(path, stbuf);
41      if (res == -1)
42          return -errno;
43
44      return 0;
45  }
46
47  static int xmp_fgetattr(const char *path, struct stat *stbuf,
48              struct fuse_file_info *fi)
49  {
50      int res;
51
52      (void) path;
53
54      res = fstat(fi->fh, stbuf);
55      if (res == -1)
56          return -errno;
57
58      return 0;
59  }
```

xmp_access(), xmp_readlink()

```
61  static int xmp_access(const char *path, int mask)
62  {
63      int res;
64
65      res = access(path, mask);
66      if (res == -1)
67          return -errno;
68
69      return 0;
70  }
71
72  static int xmp_readlink(const char *path, char *buf, size_t size)
73  {
74      int res;
75
76      res = readlink(path, buf, size - 1);
77      if (res == -1)
78          return -errno;
79
80      buf[res] = '\0';
81      return 0;
82  }
```

Struct xmp_dirp, xmp_opendir()

```
84  struct xmp_dirp {
85      DIR *dp;
86      struct dirent *entry;
87      off_t offset;
88  };
89
90  static int xmp_opendir(const char *path, struct fuse_file_info *fi)
91  {
92      int res;
93      struct xmp_dirp *d = malloc(sizeof(struct xmp_dirp));
94      if (d == NULL)
95          return -ENOMEM;
96
97      d->dp = opendir(path);
98      if (d->dp == NULL) {
99          res = -errno;
100         free(d);
101         return res;
102     }
103     d->offset = 0;
104     d->entry = NULL;
105
106     fi->fh = (unsigned long) d;
107     return 0;
108 }
```

xmp_readdir() (1/2)

```
114
115 static int xmp_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
116                         off_t offset, struct fuse_file_info *fi)
117 {
118     struct xmp_dirp *d = get_dirp(fi);
119
120     (void) path;
121     if (offset != d->offset) {
122         seekdir(d->dp, offset);
123         d->entry = NULL;
124         d->offset = offset;
125     }
126     while (1) {
127         struct stat st;
128         off_t nextoff;
129
130         if (!d->entry) {
131             d->entry = readdir(d->dp);
132             if (!d->entry)
133                 break;
134         }
135
136         memset(&st, 0, sizeof(st));
137         st.st_ino = d->entry->d_ino;
138         st.st_mode = d->entry->d_type << 12;
139         nextoff = telldir(d->dp);
```

xmp_readdir() (2/2)

```
140             if (filler(buf, d->entry->d_name, &st, nextoff))
141                 break;
142
143             d->entry = NULL;
144             d->offset = nextoff;
145         }
146
147         return 0;
148     }
```

xmp_releasedir(), xmp_mknod()

```
150  static int xmp_releasedir(const char *path, struct fuse_file_info *fi)
151  {
152      struct xmp_dirp *d = get_dirp(fi);
153      (void) path;
154      closedir(d->dp);
155      free(d);
156      return 0;
157  }
158
159  static int xmp_mknod(const char *path, mode_t mode, dev_t rdev)
160  {
161      int res;
162
163      if (S_ISFIFO(mode))
164          res = mkfifo(path, mode);
165      else
166          res = mknod(path, mode, rdev);
167      if (res == -1)
168          return -errno;
169
170      return 0;
171  }
```

xmp_mkdir(), xmp_unlink()

```
173  static int xmp_mkdir(const char *path, mode_t mode)
174  {
175      int res;
176
177      res = mkdir(path, mode);
178      if (res == -1)
179          return -errno;
180
181      return 0;
182  }
183
184  static int xmp_unlink(const char *path)
185  {
186      int res;
187
188      res = unlink(path);
189      if (res == -1)
190          return -errno;
191
192      return 0;
193  }
```

xmp_rmdir(), xmp_symlink()

```
195  static int xmp_rmdir(const char *path)
196  {
197      int res;
198
199      res = rmdir(path);
200      if (res == -1)
201          return -errno;
202
203      return 0;
204  }
205
206  static int xmp_symlink(const char *from, const char *to)
207  {
208      int res;
209
210      res = symlink(from, to);
211      if (res == -1)
212          return -errno;
213
214      return 0;
215  }
```

xmp_rename(), xmp_link()

```
217  static int xmp_rename(const char *from, const char *to)
218  {
219      int res;
220
221      res = rename(from, to);
222      if (res == -1)
223          return -errno;
224
225      return 0;
226  }
227
228  static int xmp_link(const char *from, const char *to)
229  {
230      int res;
231
232      res = link(from, to);
233      if (res == -1)
234          return -errno;
235
236      return 0;
237  }
```

xmp_chmod(), xmp_chown()

```
239  static int xmp_chmod(const char *path, mode_t mode)
240  {
241      int res;
242
243      res = chmod(path, mode);
244      if (res == -1)
245          return -errno;
246
247      return 0;
248  }
249
250  static int xmp_chown(const char *path, uid_t uid, gid_t gid)
251  {
252      int res;
253
254      res = lchown(path, uid, gid);
255      if (res == -1)
256          return -errno;
257
258      return 0;
259  }
```

xmp_truncate(), xmp_ftruncate()

```
261  static int xmp_truncate(const char *path, off_t size)
262  {
263      int res;
264
265      res = truncate(path, size);
266      if (res == -1)
267          return -errno;
268
269      return 0;
270  }
271
272  static int xmp_ftruncate(const char *path, off_t size,
273                           struct fuse_file_info *fi)
274  {
275      int res;
276
277      (void) path;
278
279      res = ftruncate(fi->fh, size);
280      if (res == -1)
281          return -errno;
282
283      return 0;
284  }
```

xmp_utimens(), xmp_create()

```
286 #ifdef HAVE_UTIMENSAT
287 static int xmp_utimens(const char *path, const struct timespec ts[2])
288 {
289     int res;
290
291     /* don't use utime/utimes since they follow symlinks */
292     res = utimensat(0, path, ts, AT_SYMLINK_NOFOLLOW);
293     if (res == -1)
294         return -errno;
295
296     return 0;
297 }
298 #endif
299
300 static int xmp_create(const char *path, mode_t mode, struct fuse_file_info *fi)
301 {
302     int fd;
303
304     fd = open(path, fi->flags, mode);
305     if (fd == -1)
306         return -errno;
307
308     fi->fh = fd;
309     return 0;
310 }
```

xmp_open(), xmp_read()

```
312  static int xmp_open(const char *path, struct fuse_file_info *fi)
313  {
314      int fd;
315
316      fd = open(path, fi->flags);
317      if (fd == -1)
318          return -errno;
319
320      fi->fh = fd;
321      return 0;
322  }
323
324  static int xmp_read(const char *path, char *buf, size_t size, off_t offset,
325              struct fuse_file_info *fi)
326  {
327      int res;
328
329      (void) path;
330      res = pread(fi->fh, buf, size, offset);
331      if (res == -1)
332          res = -errno;
333
334      return res;
335  }
```

xmp_read_buf()

```
337  static int xmp_read_buf(const char *path, struct fuse_bufvec **bufp,
338              size_t size, off_t offset, struct fuse_file_info *fi)
339  {
340      struct fuse_bufvec *src;
341
342      (void) path;
343
344      src = malloc(sizeof(struct fuse_bufvec));
345      if (src == NULL)
346          return -ENOMEM;
347
348      *src = FUSE_BUFVEC_INIT(size);
349
350      src->buf[0].flags = FUSE_BUF_IS_FD | FUSE_BUF_FD_SEEK;
351      src->buf[0].fd = fi->fh;
352      src->buf[0].pos = offset;
353
354      *bufp = src;
355
356      return 0;
357  }
```

xmp_write(), xmp_write_buf()

```
359  static int xmp_write(const char *path, const char *buf, size_t size,
360               off_t offset, struct fuse_file_info *fi)
361  {
362      int res;
363
364      (void) path;
365      res = pwrite(fi->fh, buf, size, offset);
366      if (res == -1)
367          res = -errno;
368
369      return res;
370  }
371
372  static int xmp_write_buf(const char *path, struct fuse_bufvec *buf,
373               off_t offset, struct fuse_file_info *fi)
374  {
375      struct fuse_bufvec dst = FUSE_BUFVEC_INIT(fuse_buf_size(buf));
376
377      (void) path;
378
379      dst.buf[0].flags = FUSE_BUF_IS_FD | FUSE_BUF_FD_SEEK;
380      dst.buf[0].fd = fi->fh;
381      dst.buf[0].pos = offset;
382
383      return fuse_buf_copy(&dst, buf, FUSE_BUF_SPLICE_NONBLOCK);
384  }
```

xmp_statfs(), xmp_flush()

```
386  static int xmp_statfs(const char *path, struct statvfs *stbuf)
387  {
388      int res;
389
390      res = statvfs(path, stbuf);
391      if (res == -1)
392          return -errno;
393
394      return 0;
395  }
396
397  static int xmp_flush(const char *path, struct fuse_file_info *fi)
398  {
399      int res;
400
401      (void) path;
402      /* This is called from every close on an open file, so call the
403         close on the underlying filesystem. But since flush may be
404         called multiple times for an open file, this must not really
405         close the file.  This is important if used on a network
406         filesystem like NFS which flush the data/metadata on close() */
407      res = close(dup(fi->fh));
408      if (res == -1)
409          return -errno;
410
411      return 0;
412  }
```

xmp_release(), xmp_fsync()

```
414  static int xmp_release(const char *path, struct fuse_file_info *fi)
415  {
416      (void) path;
417      close(fi->fh);
418
419      return 0;
420  }
421
422  static int xmp_fsync(const char *path, int isdatasync,
423               struct fuse_file_info *fi)
424  {
425      int res;
426      (void) path;
427
428  #ifndef HAVE_FDATASYNC
429      (void) isdatasync;
430  #else
431      if (isdatasync)
432          res = fdatasync(fi->fh);
433      else
434  #endif
435          res = fsync(fi->fh);
436      if (res == -1)
437          return -errno;
438
439      return 0;
440  }
```

xmp_setattr(), xmp_getattr()

```
443  /* xattr operations are optional and can safely be left unimplemented */
444  static int xmp_setxattr(const char *path, const char *name, const char *value,
445              size_t size, int flags)
446  {
447      int res = lsetxattr(path, name, value, size, flags);
448      if (res == -1)
449          return -errno;
450      return 0;
451  }
452
453  static int xmp_getxattr(const char *path, const char *name, char *value,
454              size_t size)
455  {
456      int res = lgetxattr(path, name, value, size);
457      if (res == -1)
458          return -errno;
459      return res;
460  }
```

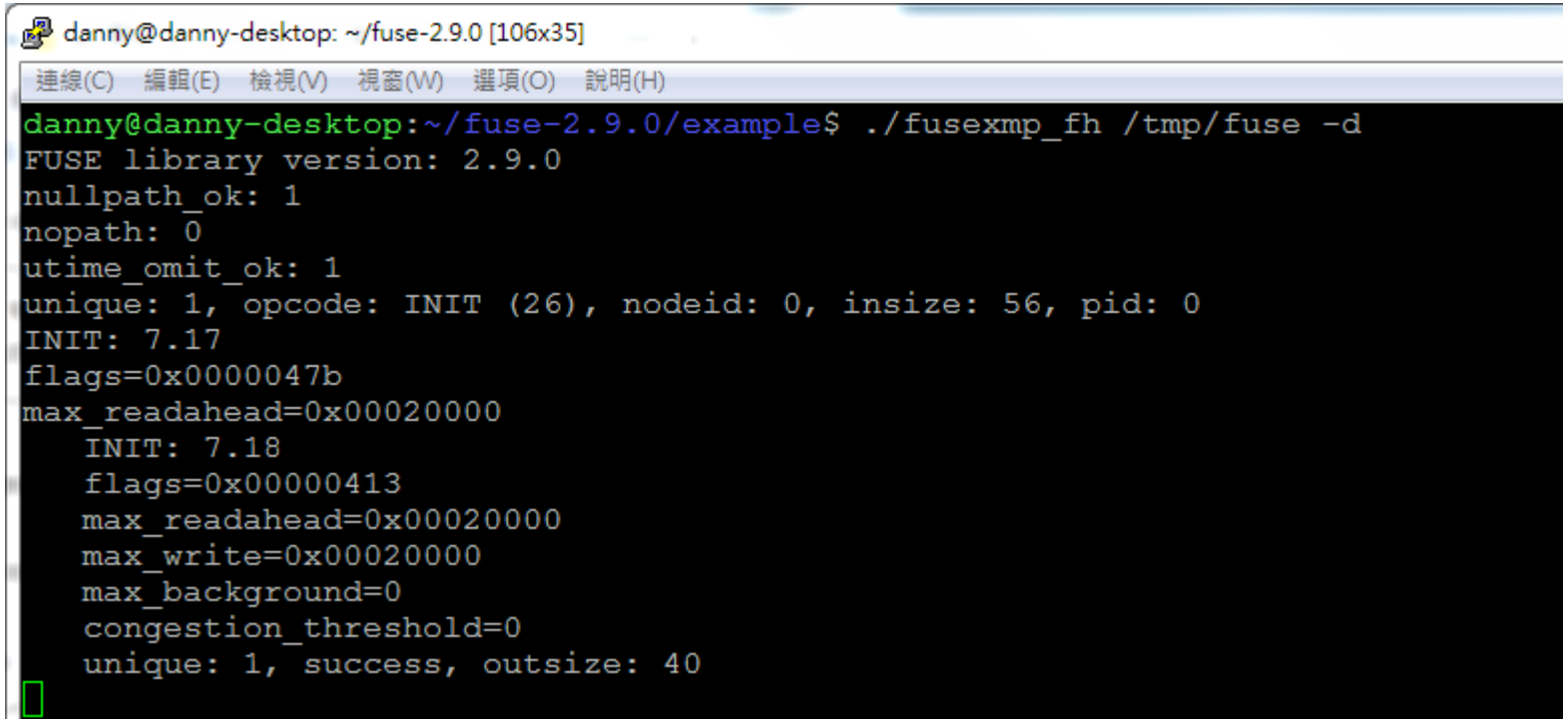
xmp_listattr(), xmp_removexatttr()

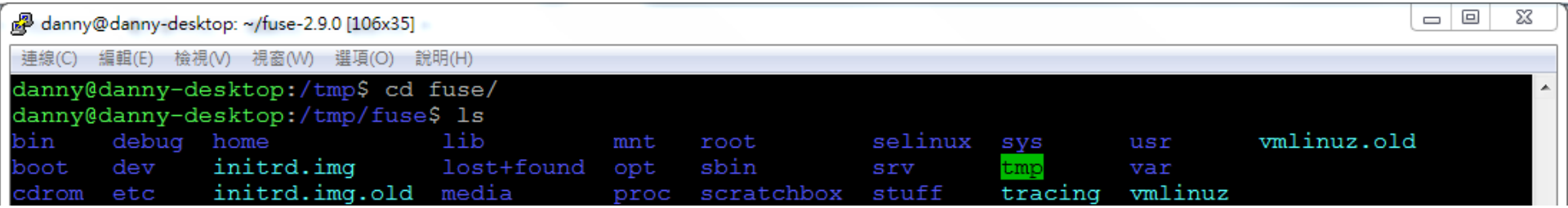
```
462  static int xmp_listxattr(const char *path, char *list, size_t size)
463  {
464      int res = llistxattr(path, list, size);
465      if (res == -1)
466          return -errno;
467      return res;
468  }
469
470  static int xmp_removexattr(const char *path, const char *name)
471  {
472      int res = lremovexattr(path, name);
473      if (res == -1)
474          return -errno;
475      return 0;
476  }
477  #endif /* HAVE_SETXATTR */
```

xmp_lock(), xmp_flock()

```
479  static int xmp_lock(const char *path, struct fuse_file_info *fi, int cmd,
480              struct flock *lock)
481  {
482      (void) path;
483
484      return ulockmgr_op(fi->fh, cmd, lock, &fi->lock_owner,
485                 sizeof(fi->lock_owner));
486  }
487
488  static int xmp_flock(const char *path, struct fuse_file_info *fi, int op)
489  {
490      int res;
491      (void) path;
492
493      res = flock(fi->fh, op);
494      if (res == -1)
495          return -errno;
496
497      return 0;
498  }
```

Example2: fusexmp_fh.c 執行





The End

Thank you for your listening