Develop your own filesystem with FUSE

No kernel programming required

Sumit Singh

October 14, 2014
(First published February 28, 2006)

With Filesystem in Userspace (FUSE), you can develop a user space filesystem framework without understanding filesystem internals or learning kernel module programming. Follow this simple, step-by-step guide to install, customize, and enable FUSE and AFS, so you can create your own fully functional filesystem in user space in Linux®.

A filesystem is a method for storing and organizing computer files and directories and the data they contain, making it easy to find and access them. If you are using a computer, you are most likely using more than one kind of filesystem. A filesystem can provided extended capabilities. It can be written as a wrapper over an underlying filesystem to manage its data and provide an enhanced, feature-rich filesystem (such as cvsfs-fuse, which provides a filesystem interface for CVS, or a Wayback filesystem, which provides a backup mechanism to keep old copies of data).

Before the advent of user space filesystems, filesystem development was the job of the kernel developer. Creating filesystems required knowledge of kernel programming and the kernel technologies (like vfs). And debugging required C and C++ expertise. But other developers needed to manipulate a filesystem -- to add personalized features (such as adding history or forward-caching) and enhancements.

Introducing FUSE

FUSE lets you develop a fully functional filesystem that has a simple API library, can be accessed by non-privileged users, and provides a secure implementation. And, to top it all off, FUSE has a proven track record of stability.

With FUSE, you can develop a filesystem as executable binaries that are linked to the FUSE libraries -- in other words, this filesystem framework doesn't require you to learn filesystem internals or kernel module programming.

When it comes to filesystems, the user space filesystem is not a new design. A sampling of commercial and academic implementations of user space filesystems include:

- LUFS is a hybrid user space filesystem framework that supports an indefinite number of filesystems transparently for any application. It consists of a kernel module and a user space daemon. Basically it delegates most of the VFS calls to a specialized daemon that handles them.
- UserFS allows user processes to be mounted as a normal filesystem. This proof-of-concept prototype provides ftpfs, which allows anonymous FTP with a filesystem interface.
- The Ufo Project is a global filesystem for Solaris that allows users to treat remote files exactly as if they were local.
- OpenAFS is an open source version of the Andrew FileSystem.
- CIFS is the Common Internet FileSystem.

Unlike these commercial and academic examples, FUSE brings the capabilities of this filesystem design to Linux. Because FUSE uses executables (instead of, say, shared objects as LUFS uses), it makes debugging and developing easier. FUSE works with both kernels (2.4.x and 2.6.x) and now supports Java™ binding, so you aren't limited to programming the filesystem in C and C++. (See Related topics for more userland filesystems that use FUSE.)

To create a filesystem in FUSE, you need to install a FUSE kernel module and then use the FUSE library and API set to create your filesystem.

Unpack FUSE

To develop a filesystem, first download the FUSE source code (look on GitHub) and unpack the package: `tar -zxvf fuse-2.2.tar.gz`. This creates a FUSE directory with the source code. The contents of the fuse-2.2 directory are:

- **./doc** contains FUSE-related documentation. At this point, there is only one file, how-fuse-works.
- **./kernel** contains the FUSE kernel module source code (which, of course, you don't need to know for developing a filesystem with FUSE).
- **./include** contains the FUSE API headers, which you need to create a filesystem. 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 filesystem.
- **./util** has the source code for the FUSE utility library.
- **./example**, of course, contains samples for your reference, like the fusexmp.null and hello filesystems.

Build and install FUSE

1. Run the configure script from the fuse-2.2 directory: `./configure`. This creates the required makefiles, etc.
2. Run `./make` to build the libraries, binaries, and kernel module. Check the kernel directory for the file ./kernel/fuse.ko -- this is the kernel module file. Also check the lib directory for fuse.o, mount.o, and helper.o.
3. Run `./make install` to complete the installation of FUSE.

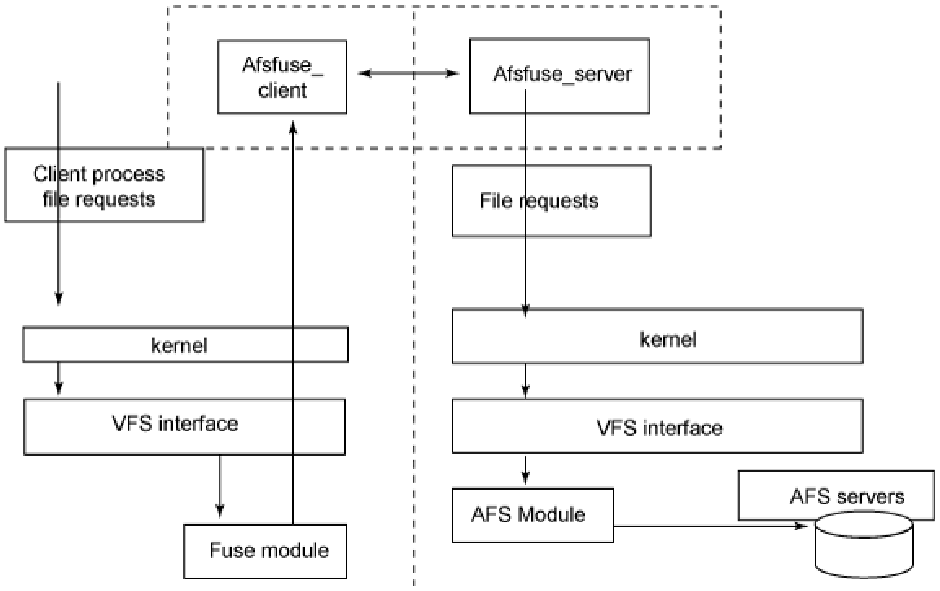
Alternative: Skip this step if you want to install the module in the kernel yourself using `insmod`. For example: `/usr/local/sbin/insmod ./kernel/fuse.ko` or `/sbin/insmod ./kernel/fuse.ko`. Remember to install the required module with root privileges.

You can do the above steps in just one step if you want. From the fuse-2.2 directory, run `./configure; make; make install;`.
Important: When making FUSE, you need to have the kernel headers or source code in place. To keep things simple, make sure that the kernel source code is in the /usr/src/ directory.

Customize the filesystem

Now let's create a filesystem so you can access your AFS space on a Linux box with latest kernel, using an older Linux kernel. You will have two processes: one server process running on a older Linux kernel, and the FUSE client process running on a Linux box with the latest kernel. Whenever a request comes to your FUSE client process, it contacts the remote server process. For communication purposes, this filesystem uses RX RPC code that is part of AFS, so you need to build OpenAFS. (Figure 1 gives an overview of this AFS filesystem.)

Figure 1. Overview of AFS-FUSE filesystem



Build OpenAFS

1. Download the OpenAFS Linux sources and untar the source.
   In the directory where you have untarred the source, run `./make ./configure --enable-transarc-paths`. If `./configure` cannot understand the sysname for build, then use the `--with-afs-sysname` option with the appropriate sysname.
   For a build on the Linux 2.4 kernel, use following command: `./configure --enable-transarc-paths --with-afs-sysname=i386_linux24`.
2. Run `./make`, and then run `./make dest`. Check for any errors during build.
   If the build went okay, then your AFS source tree is ready to work with. At this stage, you need to prepare a development directory named afsfuse. In that directory, create two more directories:

- The **include** directory will contain the include headers from OpenAFS and FUSE.
- The **lib** directory will contain the libraries from OpenAFS and FUSE.

3. Copy the headers and libraries.
   First copy the AFS headers from the OpenAFS directory by copying the directories and files from the dest\i386_linux24\include into the include directory. Then copy the FUSE headers from the fuse-2.2 directories into this directory. Repeat the same steps for libraries into the lib directory.
4. Create the structure of the application.
   You need two sets of files for two sets of processes. Name the client process files using the naming scheme **afsfuse_client.***; name the server process files **afsfuse_server.***.

   You will thus have an afsfuse_client.c file that will contain the FUSE process code, an afsfuse_server.c file that will contain the server code for the process running on a remote box, a makefile, and an rxgen file for creating the RPC header (like afsfuse.xg).

   The afsfuse_client.c file will create afsfuse_client process code that will be called by the FUSE filesystem to create your filesystem (use the example fuse-2.2/example/fusexmp.c to create this file).

Define necessary functions

To create a filesystem with FUSE, you need to declare a structure variable of type `fuse_operations` and pass it on to the `fuse_main` function. The `fuse_operations` structure carries a pointer to functions that will be called when the appropriate action is required. Listing 1 shows the `fuse_operations` structure.

Listing 1. Necessary functions in fuse_operation structure

```
struct fuse_operations {
    int (*getattr) (const char *, struct stat *);
    int (*readlink) (const char *, char *, size_t);
    int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
    int (*mknod) (const char *, mode_t, dev_t);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*symlink) (const char *, const char *);
    int (*rename) (const char *, const char *);
    int (*link) (const char *, const char *);
    int (*chmod) (const char *, mode_t);
    int (*chown) (const char *, uid_t, gid_t);
    int (*truncate) (const char *, off_t);
    int (*utime) (const char *, struct utimbuf *);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t, off_t,struct fuse_file_info *);
    int (*statfs) (const char *, struct statfs *);
    int (*flush) (const char *, struct fuse_file_info *);
    int (*release) (const char *, struct fuse_file_info *);
    int (*fsync) (const char *, int, struct fuse_file_info *);
    int (*setxattr) (const char *, const char *, const char *, size_t, int);
    int (*getxattr) (const char *, const char *, char *, size_t);
    int (*listxattr) (const char *, char *, size_t);
    int (*removexattr) (const char *, const char *);
};
```

None of these operations are absolutely essential, but many are needed for a filesystem to work properly. You can implement a full-featured filesystem with the special-purpose methods `.flush`,

`.release`, or `.fsync`. (This article doesn't cover any `xattr` functions.) The functions shown in Listing 1 are as follows:

- `getattr: int (*getattr) (const char *, struct stat *);`
  This is similar to `stat()`. The `st_dev` and `st_blksize` fields are ignored. The `st_ino` field is ignored unless the `use_ino` mount option is given.
- `readlink: int (*readlink) (const char *, char *, size_t);`
  This reads the target of a symbolic link. The buffer should be filled with a null-terminated string. The buffer size argument includes the space for the terminating null character. If the linkname is too long to fit in the buffer, it should be truncated. The return value should be "0" for success.
- `getdir: int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);`
  This reads the contents of a directory. This operation is the `opendir()`, `readdir()`, ..., `closedir()` sequence in one call. For each directory entry, the `filldir()` function should be called.
- `mknod: int (*mknod) (const char *, mode_t, dev_t);`
  This creates a file node. There is no `create()` operation; `mknod()` will be called for creation of all non-directory, non-symlink nodes.
- `mkdir: int (*mkdir) (const char *, mode_t);`
  `rmdir: int (*rmdir) (const char *);`
  These create and remove a directory, respectively.
- `unlink: int (*unlink) (const char *);`
  `rename: int (*rename) (const char *, const char *);`
  These remove and rename a file, respectively.
- `symlink: int (*symlink) (const char *, const char *);`
  This creates a symbolic link.
- `link: int (*link) (const char *, const char *);`
  This creates a hard link to a file.
- `chmod: int (*chmod) (const char *, mode_t);`
  `chown: int (*chown) (const char *, uid_t, gid_t);`
  `truncate: int (*truncate) (const char *, off_t);`
  `utime: int (*utime) (const char *, struct utimbuf *);`
  These change the permission bits, owner and group, size, and access/modification times of a file, respectively.
- `open: int (*open) (const char *, struct fuse_file_info *);`
  This is the file open operation. No creation or truncation flags (`O_CREAT`, `O_EXCL`, `O_TRUNC`) will be passed to `open()`. This should check if the operation is permitted for the given flags. Optionally, `open()` may also return an arbitrary filehandle in the `fuse_file_info` structure, which will be passed to all file operations.
- `read: int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);`
  This reads data from an open file. `read()` should return exactly the number of bytes requested, except on EOF or error; otherwise, the rest of the data will be substituted with zeroes. An exception to this is when the `direct_io` mount option is specified, in which case the return value of the `read()` system call will reflect the return value of this operation.

- `write: int (*write) (const char *, const char *, size_t, off_t, struct fuse_file_info *);`
  This writes data to an open file. `write()` should return exactly the number of bytes requested except on error. An exception to this is when the `direct_io` mount option is specified (as in the `read()` operation).
- `statfs: int (*statfs) (const char *, struct statfs *);`
  This gets filesystem statistics. The `f_type` and `f_fsid` fields are ignored.
- `flush: int (*flush) (const char *, struct fuse_file_info *);`
  This represents flush-cached data. It is not equivalent to `fsync()` -- it's not a request to sync dirty data. `flush()` is called on each `close()` of a file descriptor, so if a filesystem wants to return write errors in `close()` and the file has cached dirty data, this is a good place to write back data and return any errors. Since many applications ignore `close()` errors, this is not always useful.

Note: The `flush()` method may be called more than once for each `open()`. This happens if more than one file descriptor refers to an opened file due to `dup()`, `dup2()`, or `fork()` calls. It is not possible to determine if a flush is final, so each flush should be treated equally. Multiple write-flush sequences are relatively rare, so this shouldn't be a problem.

- `release: int (*release) (const char *, struct fuse_file_info *);`
  This releases 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. For every `open()` call, there will be exactly one `release()` call with the same flags and file descriptor. It is possible to have a file opened more than once, in which case only the last release will count and no more reads/writes will happen on the file. The return value of release is ignored.
- `fsync: int (*fsync) (const char *, int, struct fuse_file_info *);`
  This synchronizes file contents. If the `datasync` parameter is non-zero, then only the user data should be flushed, not the meta data.
- `setxattr: int (*setxattr) (const char *, const char *, const char *, size_t, int);`
  `getxattr: int (*getxattr) (const char *, const char *, char *, size_t);`
  `listxattr: int (*listxattr) (const char *, char *, size_t);`
  `removexattr: int (*removexattr) (const char *, const char *);`
  These set, get, list, and remove extended attributes, respectively.

The resulting filesystem

Your filesystem will look something like this:

```
afsfuse_client   <--RX[RPC]-->   afsfuse_server
```

The afsfuse_client will forward the filesystem calls passed to it to the afsfuse_server residing on a different machine. The afsfuse_server will service all the requests sent to it by the client and return the results to the client. It does all the jobs necessary. The mechanism used for RPC is RX. There is no caching involved for either data or metadata.

Define the RX RPC layer

Before going further, you need to define your RX RPC layer. To do that, create an .xg file for rxgen, to describe your proxy and stub code that will be linked with your afsfuse_client.c and afsfuse_server.c. Listing 2 shows how to create an afsfuse.xg file with the following contents:

Listing 2. Creating an afsfuse.xg file

```
#define MYMAXPATH 512
%#include <rx/rx.h>
%#include </rx_null.h >
%#define SAMPLE_SERVER_PORT 5000
%#define SAMPLE_SERVICE_PORT 0
/* i.e. user server's port */
%#define SAMPLE_SERVICE_ID 4 /* Maximum number of requests that will be handled by this
                                service simultaneously */
/* This number will also be guaranteed to execute in parallel if no services' requests
   are being processed */
%#define SAMPLE_MAX 2 /* Minimum number of requests that are guaranteed to be handled
                         immediately */
%#define SAMPLE_MIN 1 /* Index of the "null" security class in the sample service. This
                         must be 0 (there are N classes, numbered from 0. In this case,
                         N is 1) */
%#define SAMPLE_NULL 0 /********************** fuse4_file_info taken from fuse.h the
                        rxgen does not understands fuse.h  mystat taken from man 2
                        mystat these are required again rxgen does not understand the
                        struct paras and will bump.. **********************/
struct my_file_info { /** Open flags. Available in open() and release() */
                     int flags; /** File handle. May be filled in by filesystem in
                                   open(). Available in all other file operations */
                     unsigned int fh; /** In case of a write operation indicates if
                                         this was caused by a writepage */
                     int writepage;
                    };
struct mystatfs {
                    afs_uint32 f_type; /* type of filesystem (see below) */
                    afs_uint32 f_bsize; /* optimal transfer block size */
                    afs_uint32 f_blocks; /* total data blocks in file system */
                    afs_uint32 f_bfree; /* free blocks in fs */
                    afs_uint32 f_bavail; /* free blocks avail to non-superuser */
                    afs_uint32 f_files; /* total file nodes in file system */
                    afs_uint32 f_ffree; /* free file nodes in fs */
                    afs_uint32 f_fsid1; /* file system id */
                    afs_uint32 f_fsid2; /* file system id */
                    afs_uint32 f_namelen; /* maximum length of filenames */
                    afs_uint32 f_spare[6]; /* spare for later */
                };
struct mystat {
                    afs_uint32 st_dev; /* device */
                    afs_uint32 st_ino; /* inode */
                    afs_uint32 st_mode; /* protection */
                    afs_uint32 st_nlink; /* number of hard links */
                    afs_uint32 st_uid; /* user ID of owner */
                    afs_uint32 st_gid;/* group ID of owner */
                    afs_uint32 st_rdev; /* device type (if inode device) */
                    afs_uint32 st_size; /* total size, in bytes */
                    afs_uint32 st_blksize; /* blocksize for filesystem I/O */
                    afs_uint32 st_blocks; /* number of blocks allocated */
                    afs_uint32 st_atim; /* time of last access */
                    afs_uint32 st_mtim; /* time of last modification */
                    afs_uint32 st_ctim; /* time of last change */
                };
struct my_dirhandle{
                    afs_uint32 type;
                    afs_uint32 inode;
                    char name[MYMAXPATH];
                };
typedef my_dirhandle bulkmydirhandles<>;
 /*********************phase 1 functions *********************************************/
rxc_getattr(IN string mypath<MYMAXPATH>, IN int dummy) split = 1;
rxc_getdirWrapper(IN string path<MYMAXPATH>, OUT bulkmydirhandles *handles) = 2;
rxc_read(IN string path<MYMAXPATH>;, IN afs_uint32 size, IN afs_uint32 offset,
         IN struct my_file_info *fi) split = 3;
rxc_open(IN string path<MYMAXPATH>, IN int flags, OUT u_int *hd) = 4;
rxc_write(IN string path<MYMAXPATH>,IN afs_uint32 size, IN afs_uint32 offset,
          IN struct my_file_info *fi) split = 5;
rxc_chmod(IN string path<MYMAXPATH>, IN afs_uint32 mode) = 6;
rxc_chown(IN string path<MYMAXPATH>, IN afs_uint32 uid, IN afs_uint32 gid) = 7;
rxc_utime(IN string path<MYMAXPATH>, IN afs_uint32 at,IN afs_uint32 mt) = 8;
rxc_mknod(IN string path<MYMAXPATH>, afs_uint32 mode, afs_uint32 rdev) = 9 ;
rxc_mkdir(IN string path<MYMAXPATH>, IN afs_uint32 mode) = 10;
rxc_unlink(IN string path<MYMAXPATH>) = 11 ;
rxc_rmdir(IN string path<MYMAXPATH>) = 12;
rxc_rename(IN string from<MYMAXPATH>, IN string to<MYMAXPATH>) = 13;
rxc_truncate(IN string path<MYMAXPATH>, IN afs_uint32 size) = 14;
rxc_release(IN string path<MYMAXPATH>, IN struct my_file_info *fi) = 15;
rxc_readlink(IN string path<MYMAXPATH>, IN afs_uint32 size,OUT string
             data<MYMAXPATH>) = 16;
rxc_symlink(IN string from<MYMAXPATH>, IN string to<MYMAXPATH>) = 17;
rxc_link(IN string from<MYMAXPATH>, IN string to<MYMAXPATH>) = 18;
rxc_statfs(IN string path<MYMAXPATH>, OUT struct mystatfs *stbuf) = 19;
rxc_fsync(IN string path <MYMAXPAT>, IN int isdatasync, IN struct my_file_info
          *fi) = 20 ;
rxc_flush(IN string path <MYMAXPATH>, IN struct my_file_info *fi) = 21 ;
```

When defining the RX RPC layer, note these points:

- You've defined `mystatfs`, `mystat`, and `my_file_info` as a wrapper over `statfs`, `stat`, and `fuse_file_info` structures. These will be transformed over wire using the generated XDR code. (XDR, External Data Representation, allows data to be wrapped in an architecture-independent manner so data can be transferred between heterogeneous computer systems.)
- You've defined nearly one function for every member of structure `fuse_operations` with nearly the same parameters since the job of the afsfuse_client is just to take the calls from the FUSE filesystem and pass them on to the afsfuse_server.
- You've hard-coded some values like `MYMAXPATH`, which you should obtain from the system -- hard-coding was done for the sake of simplicity.

Create client and stub files

Next compile the afsfuse.xg file with rxgen to create client and stub files. From the directory containing the source for your afsfuse_server and afsfuse_client, run the command `openafs-1.2.13/i386_linux24/dest/bin/rxgen afsfuse.xg`. This creates the following files:

- afsfuse.cs.c is the client stub code to be linked with the afsfuse_client.c.
- afsfuse.h is the header containing various definitions for your FUSE RX code.
- afsfuse.ss.c is the server stub code (proxy code) to be linked with your afsfuse_server code.
- afsfuse.xdr.c contains code to marshall the three structures you defined in your afsfuse.xg.

Now add some code to the afsfuse_client.c and afsfuse_server.c to do the real work. Most of the calls will look like this:

- `Our_call_in_afs_fuse_client()`. Marshall the parameters and prepare for RPC. Call the afsfuse_server application over RX [RPC]. Demarshall the parameters. Copy the values to the formal parameters passed to this function.
- `Our_call_in_afs_fuse_server()`. Demarshall the parameters. Call the local filesystem or AFS-specific functions. Marshall the parameters and prepare for RPC. Make the RX RPC call.

The afsfuse_client.c call looks like this:

```
int afsfuse_readlink(const char *path, char *buf, size_t size){
    rx_connection *local& int ret& char *buffer = malloc (512)&
    memset(buffer,0,512)& memset(buf,0,size)& local = getconnection()&
    ret = rxc_rxc_readlink(local,path,512,&buffer) // rpc call
         relconnection(local)&
         strncpy(buf,buffer,512-1)&
        //<- demarshall the parametrs
        return ret&
    }
```

The afsfuse_server.c call looks like this:

Listing 3. afsfuse_server.c call

```
int rxc_rxc_readlink( struct rx_call *call, char * path, afs_uint32 size, char**data)
    { int ret& char lbuff[512] ={0}&
    translatePath(path,lbuff)& //<- make filesystem call
     *data = malloc(512)&
     res = readlink(lbuff, *data, 512-1)&
     if(res == -1) return -errno& (*data)[res] = '\0'& return 0&
    }
```

Similarly, you can add code in other functions to enhance your filesystem.

You still need to create a makefile to compile your code. Remember to include the following options while compiling code for the afsfuse_client: `-D_FILE_OFFSET_BITS=64` and `-DFUSE_USE_VERSION=22`.

Listing 4. Generating the makefile to compile client code

```
SRCDIR=./ LIBRX=${SRCDIR}lib/librx.a
LIBS=${LIBRX} ${SRCDIR}lib/liblwp.a
#CC = g++
CFLAGS=-g -I. -I${SRCDIR}include -I${SRCDIR}include/fuse/ -DDEBUG ${XCFLAGS}
    -D_FILE_OFFSET_BITS=64 -DFUSE_USE_VERSION=22
afsfuse_client: afsfuse_client.o afsfuse.xdr.o ${LIBS} bulk_io.o afsfuse.cs.o
    ${CC} ${CFLAGS} -o afsfuse_client afsfuse_client.o ${SRCDIR}lib/fuse/fuse.o
    ${SRCDIR}lib/fuse/mount.o ${SRCDIR}lib/fuse/helper.o
    ${SRCDIR}lib/fuse/fuse_mt.o bulk_io.o afsfuse.cs.o afsfuse.xdr.o ${LIBS}
afsfuse_server: afsfuse_server.o afsfuse.xdr.o afsfuse.ss.o bulk_io.o ${LIBS}
    ${CC} ${CFLAGS} -o afsfuse_server afsfuse_server.o bulk_io.o afsfuse.ss.o
    afsfuse.xdr.o ${LIBS}
#afsfuse_client.o: afsfuse.h
#afsfuse_server.o: afsfuse.h
bulk_io.o: ${CC} -c -g -I${SRCDIR}include bulk_io.c afsfuse.cs.c afsfuse.ss.c
    afsfuse.er.c afsfuse.h afsfuse.xdr.c: afsfuse.xg rxgen afsfuse.xg
afsfuse.xdr.o: afsfuse.xdr.c ${CC} -c -g -I{SRCDIR}include afsfuse.xdr.c
all: afsfuse_server afsfuse_client
clean: rm *.o rm afsfuse_client rm afsfuse_server
```

Remember, you still need to use librx.a and liblwp.a to link to RX and LWP code for RX. And fuse/fuse.o, fuse/helper.o, and fuse/mount.o are the FUSE libraries you need to link your code to.

Conclusion

In this article, you've learned how to install FUSE and OpenAFS and how to use them to create and customize your own user space filesystem, a fully functional, stable filesystem in Linux that doesn't require you to patch or recompile the active kernel -- you don't even have to be a kernel module programmer. You've seen details of the two key concepts in enabling a FUSE filesystem: how to install and configure the FUSE kernel module and how to leverage the power of the FUSE libraries and APIs.

Downloadable resources

Description	Name	Size
AFSFuse filesystem sample code	l-fuse.zip	9KB

Related topics

- For an excellent example of FUSE in action, see Google hack: Use Gmail as a Linux filesystem, which details and dissects Google Gmail (Richard Jones, Computerworld, September 2006).
- Read this introductory article on implementing FUSE.
- The "Common threads: Advanced filesystem implementor's guide" (developerWorks) is a series of articles that can help you become a filesystem expert.
- "Take charge of processor affinity" (developerWorks, September 2005) shows how to write better userspace applications.
- FUSE-J provides Java bindings for FUSE and comes with a "proof-of-concept" ZIP filesystem that seems to be pretty stable.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.