QuickTime® and a
are needed to

# Building File Systems with

QuickTime® and a
decompressor
are needed to see thi

Xavid Pretzer

SIPB IAP 2009

# What is FUSE?

- Stands for “File system in USErspace”

# What's a File System?

- A file system maps file paths (e.g., /etc/hostname) to file contents and metadata
- Metadata includes modification times, permissions, etc.
- File systems are 'mounted' over a particular directory

# What is Userspace?

- Your operating system has (at least) two modes: kernel (trusted) and user
- Kernelspace code has real ultimate power and can only be modified by root
- Base system software like filesystems are traditionally kernel modules and not changeable by normal users

# FUSE

- Makes it easy to write new filesystems
  - without knowing how the kernel works
  - without breaking unrelated things
  - more quickly/easily than traditional file systems built as a kernel module
- Makes it safe for sysadmins to let users they don't trust use custom file systems

# Other Key Features

- Cross-platform: Linux/BSD/OS X
- Wide language support: natively in C, with bindings in C++, Java, C#, Haskell, TCL, Python, Perl, Shell Script, SWIG, OCaml, Pliant, Ruby, Lua, Erlang, PHP
  - (My examples use Python)
- Low-level interface for more efficient file systems

# What do people do with FUSE?

- Hardware-based: ext2, iso, ZFS…
- Network-based: NFS, smb, SSH…
- Nontradtional: Gmail, MySQL…
- Loopback: compression, conversion, encryption, virus scanning, versioning…
- Synthetic: search results, application interaction, dynamic conf files…

# Using FUSE Filesystems

- To mount:
  - ./hello.py ~/somedir
- To unmount:
  - fusermount -u ~/somedir

# How FUSE Works

- Application makes a file-related syscall
- Kernel figures out that the file is in a mounted FUSE filesystem
- The FUSE kernel module forwards the request to your userspace FUSE app
- Your app tells FUSE how to reply

# Writing FUSE Filesystems


QuickTime® and a
decompressor
are needed to see this picture.

# Writing a FUSE Filesystem

- Write an ordinary application that defines certain functions/methods that FUSE will call to handle operations
- ~35 possible operations
- Many operations have useful defaults
  - Useful filesystems can define only ~4
  - Full-featured ones will need to define most

# Defining FUSE Operations

- In C, you define functions and put pointers to them on a struct
- In python-fuse, operations are methods on a subclass of fuse.Fuse
- You can set your Fuse subclass's file_class attribute to a class that implements the file operations, or implement them on your Fuse subclass

# FUSE Operations

- Directory Operations
- File Operations
- Metadata Operations
- Some other stuff

# Directory Operations

- readdir(path): yield directory entries for each file in the directory
- mkdir(path, mode): create a directory
- rmdir(path): delete an empty directory

# Basic File Operations

- mknod(path, mode, dev): create a file (or device)
- unlink(path): delete a file
- rename(old, new): move and/or rename a file

# Reading and Writing Files

- open(path, flags): open a file
- read(path, length, offset, fh)
- write(path, buf, offset, fh)
- truncate(path, len, fh): cut off at length
- flush(path, fh): one handle is closed
- release(path, fh): file handle is completely closed (no errors)

# Metadata Operations

- getattr(path): read metadata
- chmod(path, mode): alter permissions
- chown(path, uid, gid): alter ownership

# Meta Operations

- fsinit(self): initialize filesystem state after being mounted
  - start threads, for example

# Other Operations

- statfs(path)
- fsdestroy()
- create(path, flags, mode)
- utimens(path, times)
- readlink(path)
- symlink(target, name)
- link(target, name)
- fsync(path, fdatasync, fh)
- …

# Metadata Format

- self.st_size: size in bytes
- st_mode: type and permissions
- self.st_uid: owner id
- self.st_gid: group id
- self.st_atime: access time (often fudged)
- self.st_mtime: modification time
- self.st_ctime: metadata change time
- self.st_ino: doesn't matter too much
- self.st_dev: 0 for normal files/directories
- self.st_nlink: 2 for dirs, 1 for files (generally)

# FUSE Context

- GetContext() within a Fuse object returns a dict with:
  - uid: accessing user's user ID
  - gid: accessing user's group ID
  - pid: accessing process's ID
- Useful for nonstandard permission models and other user-specific behavior

# Errors in FUSE

- Don’t have access to the user’s terminal (if any), and can only send predefined codes from the errno module
  - Return -the error code to indicate failure
- Can log arbitrary messages to a log file for debugging

# Useful Errors

- errno.ENOSYS: Function not implemented
  errno.EROFS: Read-only file system
- errno.EPERM: Operation not permitted
- errno.EACCES: Permission denied
- errno.ENOENT: No such file or directory
- errno.EIO: I/O error
- errno.EEXIST: File exists
- errno.ENOTDIR: Not a directory
- errno.EISDIR: Is a directory
- errno.ENOTEMPTY: Directory not empty

QuickTime® and a decompressor are needed to see thi

QuickTime® and a decompressor are needed to

# Examples

# Example: hello.py

- Minimal synthetic file system
- Holds a single immutable file with a pre-defined message
- Could easily be adapted to run arbitrary code to generate the file contents
- Uses 4 operations
  - readdir, open, read, getattr

# readdir

```
fuse.fuse_python_api = (0, 2)

hello_path = '/hello'
hello_str = 'Hello World!\n'

class HelloFS(Fuse):


    def readdir(self, path, offset):
        for r in  '.', '..', hello_path[1:]:
            yield fuse.Direntry(r)
```

# open

```
hello_path = '/hello'
hello_str = 'Hello World!\n'

class HelloFS(Fuse):

    # ...

    def open(self, path, flags):
        if path != hello_path:
            return -errno.ENOENT
        accmode = os.O_RDONLY | os.O_WRONLY \
                  | os.O_RDWR
        if (flags & accmode) != os.O_RDONLY:
            return -errno.EACCES
```

# read

```
def read(self, path, size, offset):
    if path != hello_path:
        return -errno.ENOENT
    slen = len(hello_str)
    if offset < slen:
        if offset + size > slen:
            size = slen - offset
        buf = hello_str[offset:offset+size]
    else:
        buf = ''
    return buf
```

# Helper Stat subclass

```
class MyStat(fuse.Stat):
    def __init__(self):
        self.st_mode = 0
        self.st_ino = 0
        self.st_dev = 0
        self.st_nlink = 0
        self.st_uid = 0
        self.st_gid = 0
        self.st_size = 0
        self.st_atime = 0
        self.st_mtime = 0
        self.st_ctime = 0
```

# getattr

```
def getattr(self, path):
    st = MyStat()
    if path == '/':
        st.st_mode = stat.S_IFDIR | 0755
        st.st_nlink = 2
    elif path == hello_path:
        st.st_mode = stat.S_IFREG | 0444
        st.st_nlink = 1
        st.st_size = len(hello_str)
    else:
        return -errno.ENOENT
    return st
```

# Boilerplate Main

```
def main():
    usage="\nUserspace hello example\n\n" \
          + Fuse.fusage
    server = HelloFS(version="%prog "
                             + fuse.__version__,
                     usage=usage,
                     dash_s_do='setsingle')

    server.parse(errex=1)
    server.main()

if __name__ == '__main__':
    main()
```

# Example: xmp.py

- Mirrors a local file hierarchy
- Simple to implement using functions in the os module
- Shows how many operations work
- Usage:

```
./xmp.py --o root=/mit/sipb/ /tmp/mntdir
```

# __init__ and fsinit

```
fuse.fuse_python_api = (0, 2)

# We use a custom file class and fsinit

feature_assert('stateful_files', 'has_init')

class Xmp(Fuse):
    def __init__(self, *args, **kw):
        Fuse.__init__(self, *args, **kw)
        self.root = '/'
        self.file_class = self.XmpFile

    def fsinit(self):
        os.chdir(self.root)
```

# Main with Options

```
def main():
    server = Xmp(version="%prog " +
        fuse.__version__, usage=Fuse.fusage)

    server.parser.add_option(
        mountopt="root", metavar="PATH",
        default='/',
        help="mirror PATH [def: %default]")
    server.parse(values=server, errex=1)

    if server.fuse_args.mount_expected():
        os.chdir(server.root)
    server.main()
```

# Operations on Fuse Subclass

```
def getattr(self, path):
    return os.lstat("." + path)

def readdir(self, path, offset):
    for e in os.listdir("." + path):
        yield fuse.Direntry(e)

def truncate(self, path, len):
    f = open("." + path, "a")
    f.truncate(len)
    f.close()

# ...
```

# Operations on File class

```
class XmpFile(object):
    # Called for 'open'
    def __init__(self, path, flags, *mode):
        self.file = os.fdopen(
            os.open("." + path, flags, *mode),
            flag2mode(flags))
        self.fd = self.file.fileno()
    def read(self, length, offset):
        self.file.seek(offset)
        return self.file.read(length)
    def write(self, buf, offset):
        self.file.seek(offset)
        self.file.write(buf)
        return len(buf)
    # ...
```

# Examples

- For full details, see xmp.py
  - http://stuff.mit.edu/iap/2009/fuse/examples/
  - /mit/sipb-iap/www/2009/fuse/examples/
- Also look at pyhesiodfs.py there
  - Used on Debathena machines for /mit/
  - Very simple, yet useful and used widely

# Try it Out

- `ssh iap-fuse.xvm.mit.edu`
- Play with the examples:
  - http://stuff.mit.edu/iap/2009/fuse/examples/
  - /mit/sipb-iap/www/2009/fuse/examples/
- Ask me questions
- Write your own! Some fun ideas are at:
  - http://stuff.mit.edu/iap/2009/fuse/practice.html

# fuse_lowlevel.h

- C only
- Uses numeric ‘ino’ identifiers instead of always passing full paths
- Less ‘friendly’ interface (more similar to kernel interface) allows FUSE to add less overhead