Introduction to Pseudo and Stackable File Systems under BSD

Allan Fields
Afields Research / AFRSL

http://afields.ca
bsd@afields.ca

Outline

- Introduction
  - User-space and Kernel-space
  - What is the VFS?
  - What are Vnodes?
  - Vnode Operations
  - Mount points & the mount structure
  - Root Vnode
- Traditional File Systems
- Namespace

- Pseudo File Systems
  - Portal File Systems

- Stackable File Systems
  - What is Vnode Stacking?
  - How does it work?
  - Applications of Vnode Stacking

- Templated Base Filesystems
  - FiST and fistgen
  - Supported Platforms
  - The FiST language
  - Pros & Cons of template filesystems
  - Developing a template fs in FiST

- User Filesystems
- "Next Generation" Filesystems
  - Extending Filesystem Semantics
  - Adapting Filesystem Namespaces

- Conclusions
- Q&As

Introduction

- The BSD file system was first developed at University of California at Berkeley
- Significant portions of Free/Net/OpenBSD file system based on 4.4BSD code
- BSD Operating Systems default to UFS file system with FFS filestore
- Recent changes to UFS in FreeBSD include:
  - Snapshots: ability to save state of mounted file system at point in time while continuing use
  - UFS2 inode format - supports: Extended Attributes; 64-bit fields
  - Access Control Lists using EAs

User-space and Kernel-space

- BSD Operating Systems use virtual memory - hardware protection of address space
- User space: User process in the BSD Operating Systems run in user-space or user land
- Each process has own address space; process structure, user+kernel stack, heap; text and data pages
- Kernel space (kernel land): the kernel has own address space
- System Calls (syscalls) interface to kernel from userland processes

What is the VFS?

- VFS is kernel layer, provides object oriented interface to file systems
- The vnode interface provides unified set of routines called by kernel
- Abstracts interfacing details from underlying file systems
- Supports multiple file system types
- VFS provides a common set of routines for working with file systems: mounting, sync, quota, etc.
- Standard/default setof vfsops

What are Vnodes?

- File system in Unix allows uniform access to multiple objects types
- Vnode: generalized file system object - struct vnode
- Vnodes can represent different types of objects
  - VREG: regular files
  - VDIR: directories
  - VBLK: block devices
  - VCHR: character devices
  - VLNK: link devices
  - VSOCK: sockets
  - VFIFO: named pipe

Vnode Operations

- Vnode Operations (vnops) are basic file system primitives for operating on vnodes
- vnode operation vector: structure contains function pointers to routines
- VOP_* Macros: provide object oriented calling of vop on vnode
- Default vnode ops: Standard routines defined by system; vnode operations not defined by file system fall back on defaults

The Big Picture

- User processes performing I/O or lookups call standard library functions such as:
  - `read(), write(), readdir()`
- This initiates syscall into kernel
- Kernel then references vnode and calls appropriate vnode operation

Namespace

- Namespaces composed of symbolic identifiers, usually representative of objects
- Namespaces have two primary elements:
  - Names or identifiers:
    - point to underlying objects
    - usually unique at any given scope
  - Spaces:
    - flat / single-level namespaces
    - hierarchical directories (trees)
- BSD filesy stem namespace is hierarchical, path based; Element of namespace is path component
- Lookup recursively resolves vnode referenced by path

Pseudo File Systems

- Pseudo File Systems behave like normal file systems with some primary differences
- Allow creation of arbitrary file hierarchies accessible by standard user binaries
- Provide uniform access to objects with name and data components, convenient way to represent hierarchical information
- Can expose non-filesystem objects through VFS/vnode layers
- Names may not correspond 1:1 to files
- May not have backing storage at all
- Concerned with “top-half”: namespace, file system semantics

Pseudo File Systems Include..

- procfs
  - Provides a file system mount with processes
  - Process status and control through standard file entries
- devfs
  - Provides dynamicaly generated tree w/ device nodes
  - Old approach was manually created device entries in user-space
  - Interface directly to kernel device drivers
  - Now standard in FreeBSD; use in jails
- portalfs
  - Establish TCP connections through FS
  - Socket semantics in file system
  - Pipes and naïve tools

Practical Benefits

- Dynamically generated hierarchies can accurately represent changing structures, entities w/o need to maintain corresponding file entries
- Use existing tools, standar library file functions: open(), read(), write(), etc..
- Exploits *nix file system closure: accesible from shell

- FreeBSD PseudoFS
  - Generic code for constructing pseudo file system trees
  - Interfaces VFS and provides API for building

Stackable File Systems

- Stackable file systems provide modular object-oriented approach to building file system
- Mounting stackable file systems on top of each other allows build complex behaviour

Applications of Vnode Stacking

- I/O Request Transformation
  - Replication
  - Caching
  - Fail-over
- Transformation of Lookup Requests
  - Overlays/Union File Systems
- Security, Logging and Auditing
- User and attribute mapping
- Encryption, Compression

Template Based File Systems

- Template Based File Systems are file systems which reuse templates to generate file system code
- Designed for portability between Operating Systems with differing vnode interfaces
- Templates take care of vnode interface details.

- Template file systems aren't just for file system developers:
- System administrators have easy access to cross-platform file systems

FiST and fistgen

- FiST or the File System Translator: language for creating template based file systems
- Fistgen: Tool to generate file system code based on FiST code and file system templates
- Supports vnode stacking, fan-in and fan-out
- Enables:
  - portability of code
  - reduced development time and effort
- Minimal performance overhead
- Many have expressed interest using FiST under BSD

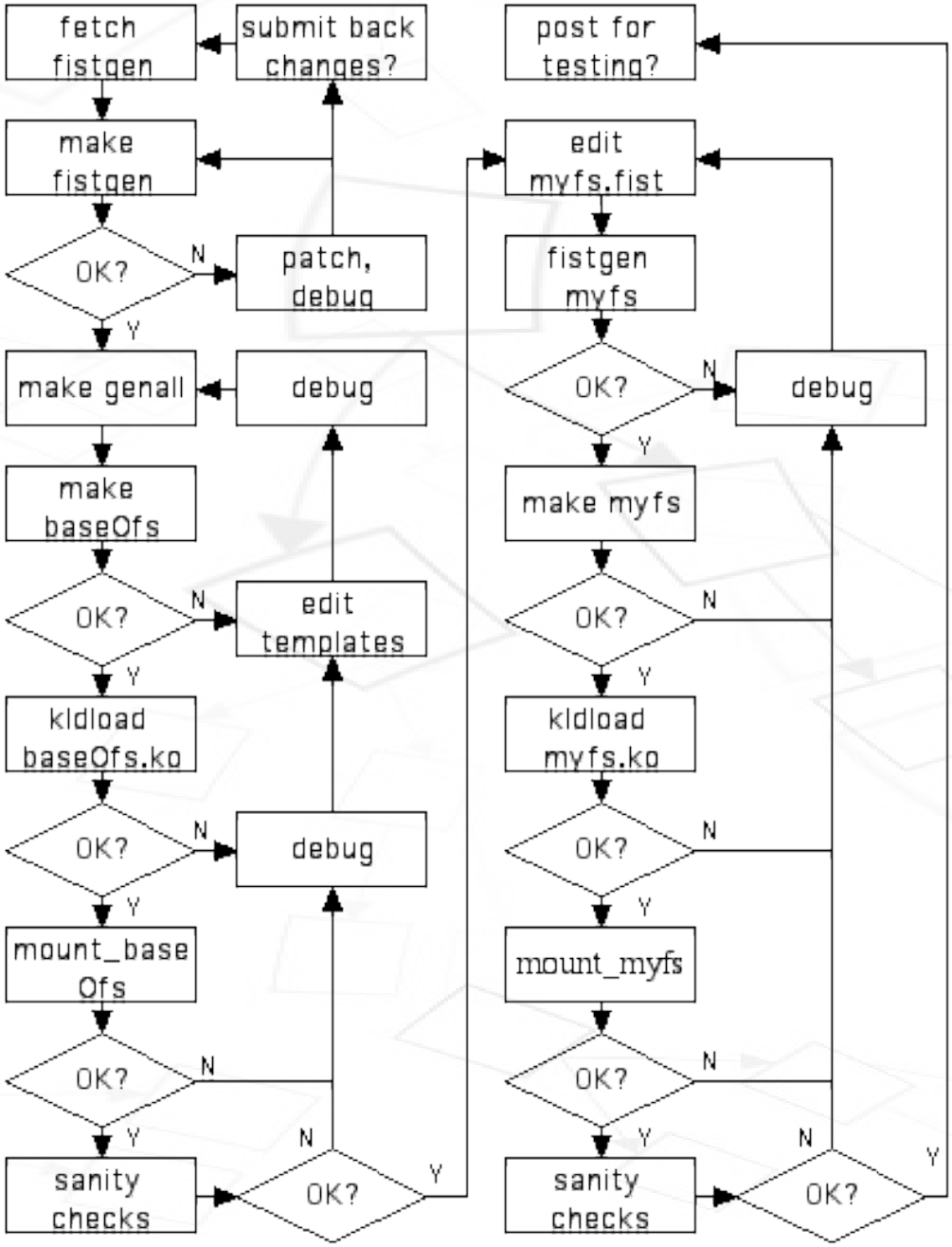
Supported Platforms

- fistgen provides templates for several platforms:
  - Linux
  - FreeBSD
  - Solaris
- Linux support is most mature, latest development efforts support Linux kernels
- Current state of FreeBSD support:
  - FreeBSD support vastly improved in recent releases
  - FreeBSD templates still require work to support wider range of releases

Work on fistgen FreeBSD port

- Problems w/ templates & different releases
- Keeping templates synchronized with changes to VFS, vnode interfaces (requires periodic maintenance)
- Linux-supported features needing poring:
  - SCAs: Size Change Algorithm support
  - Attach-mode mounts
- Fistgen has potential for true cross-platform file system on major free nixes
- Porting to additional platforms requires more templates to be built: Net/OpenBSD

FiST Dev. Flowchart



The FiST language

- FiST is intermediary language: contains declarations, rules and C code-blocks
- fistgen fills-out template, produces C code, sets up Makefile
- Supports pre-call, call & Post-call operation
- Hooks into vnode ops: before, during & after call to vnop, call any further vnops
- insert arbitrary blocks of code (doesn't have to be vnode specific)
- Define filter functions to {en,de}code data
- Read-only (readops), writeonly (writeops)

Pros of Template File Systems

- Portability: cross-platform support
- Ease of Use: potentially shallower developer learning curve
- Frees developer to focus on file system specific code, semantics
- Maintainability - easier to maintain:
  - 1 set of templates than 7-8 different file systems
  - 4 sets of templates than 4 ports of same file system
- Extensible approach to constructing file systems

Cons of Template File Systems

- Templates can break easily with changes to vnode interface (also true w/ non-template fs)
- Some templates aren't complete: FreeBSD templates (more developers needed)
- Slight overhead incurred (minimal impact)
- Coding Trade-off: easier to develop with, but loose some control over code (can be "tweaked" afterward, address some problems manually)
- Some loss of versatility: Not all things are possible in FiST (support could be added)
- Exporting FS over NFS can be problematic

Developing a Template fs in FiST

- Process for creating a file system with fistgen:
  - Fetch fistgen tarball
    ```
    fetch ftp://ftp.filesystems.org/fist/fistgen-0.0.n.tar.gz
    tar xzf fistgen-0.0.n.tar.gz
    ```
  - Build fistgen `cd fistgen-0.0.n; ./configure&&make`
  - Build test filesystems `make genall` # Test target only works on Linux
  - Create a .fist file `cp testfs/wrapfs.fist testfs.fist`
  - (Add code here) `$EDITOR testfs.fist`
  - Generate code `fistgen testfs.fist`
  - Run make `cd out/testfs; make`
  - Test kernel module `kldload ./testfs.ko`
  - Attempt mount `mkdir /mnt/test`
  - Sanity Checks `./mount_testfs /tmp /mnt/test`

```
mkdir /mnt/test/1; rmdir /mnt/test/1
touch /mnt/test/2; rm /mnt/test/2
dd if=/dev/random of=/mnt/test/3; dd if=/mnt/test/3 of=/dev/null
```

User File Systems

- User file systems such a cfsd: user-process
- NFS interface, kernel-stub
- Problems w/ user file systems under BSD:
- Slow: context-switch necessary to service I/O request
- Scalability limited: single-process user-file systems create bottlenecks
- Hurd: microkernel based, provides API for filesystem sevices in userspace
- DragonFlyBSD: proposing various changes: message passing, VFS; could make user file systems practical

"Next Generation" File Systems

- Latest file systems to hit the scene provide journaling or balanced-tree implementations
- Differing on-disk structure (filestore layout)
- Include enhanced semantics
- Mostly monolithic development model:
  - Separate/”full” file systems (unit-wise replacement for existing FS)
  - Not the Stacking Model
  - Implement both namespace and filestore components
- BSD focus remains on extending UFS and supporting additional file systems
- Linux has three major next-gen file systems

Next-gen includes..

- XFS (SGI) [Popular journaling File System]
  - Some BSD ports available or in progress
- Ext3fs
  - Next generation of Linux Extended File System
- ReiserFS (Namesys) [B+Tree Based]
  - currently Linux-only development; major distros
  - significant departure from traditional file systems: reiser4 creates new syscalls, significant changes in semantics
  - supports plug-ins
- BeFS (Be, Inc.)
  - BeOS file system, was clean reimplementation of traditional inode-based file system
  - Provided native [extended] attributes and indexing (“database” like features)
  - BeOS/BeFS being reimplemented: OpenBeOS

Adapting File System Namepsaces

- pseudo and stackable file systems provide ability to adapt namespaces to specific purposes
- Examples:
  - procfs (pseudo) - creates a namespace modeled on running processes
  - unionfs (stackable) - combines namespaces of two file systems; overlays
- VFS vs. other kernel APIs

Conclusions

- Many possibilities exist for pseudo and stackable file systems
- Pseudo file systems provide
- Wide-ranging applications for advanced file systems
- Going beyond a simple 1:1 filename:object mapping

- Accompanying paper: Introduction to Pseudo and Stackable File systems under BSD
- See:
  - More info: `http://afields.ca/bsdcan/2004`
  - FiST homepage: http://www.filesystems.org

Q&As

- Questions? Comments?

- Speaking of name spaces: "file system" vs. "filesystem"?
  - Liberals: filesystem is a new word!
  - Conservatives: file system (two words)
  - Traditionalists: Don't you mean: microcomputer disk filing system?
  - Pragmatists: Randomly placed use of both forms, doesn't matter
  - Foldoc: filesystem syn. file system