xFS Simulation Environment

Doug Doucette

1.0 Introduction

This is the design document for the simulation environment (scaffolding) for xFS.

2.0 Goals and Requirements

- Allow user mode debugging of all xFS project file system components. Excludes volume manager (xlv) from the set of things we want to debug, at least for now.
- Allow integration and testing of all file system components (again, except xlv).
- Changes to file system components to work in the simulation environment must be minimal and limited to **#ifdefs**.
- Allow multi-user testing.
- Present a transparent interface to applications, except for linking differently.
- Run on unmodified 5.x machines (at least). [No kernel changes.]
- Use disk space identified by pathname and range of addresses for the data and log spaces. Allow either raw disk or ordinary files to be used.

3.0 Design Alternatives

There are three major design alternatives we could use to meet some or all of the goals and requirements listed above. Roughly, they are:

1. #**ifdef** the utilities to avoid name conflicts, and link all the utilities together with the file system code. Each user “process” gets its own sproc, and all the “kernel” data is shared. The “shell” is just some simple command lookup code.
2. Utilities link with a library which replaces all the file name and file descriptor calls; the library intercepts calls into the simulated filesystem and passes calls to real filesystems onto the kernel. The intercepted calls turn into messages to a set of server sproc’s which implement the file system.
3. The simulation registers itself as an NFS server for the simulated mount point, and runs as a user mode NFS server. No wrapper library is needed.

Any of these could be implemented. Each has differing implementation cost and a different set of problems and benefits.

Alternative 1 requires the most in the way of source changes to the utilities and test programs, since they must be changed to avoid overlapping names with each other and with the kernel file system code. Fork calls must be intercepted, and turned into sproc calls; exec calls must be intercepted and turned into function calls to other “programs” linked in. The real shells may be too complex to make work this way, but probably a simple shell can be written as the top level command interpreter; on the other hand, this means that scripting would be difficult, so maybe it is a requirement to make **sh** work. There is no protection for programs from each other, nor for the kernel from the programs. Memory allocation, if done with malloc, would be problematic since programs don’t normally release their memory before exiting; otherwise the malloc implementation needs to be replaced with some code that can find the “process” memory and release it.

Alternative 2 simulates the actual user/kernel split well. There is an implementation cost in making the message-passing system call library work. Not all system calls can be implemented in a straightforward way to reference the simulated file system: exec, for instance (and mmap), are difficult. We can use a real shell program. Utility changes will be minimal (turning off multiprocessing). The set of pathname functionality that can easily be implemented is a little restricted, but not too bad - the problems arise through operations such as .. through the root of the simulation.

In both alternatives 1 and 2, kernel code surrounding the filesystem must be simulated. This includes things like vnode support, memory allocation, and so on.

Alternative 3 changes this by turning the simulation into an NFS server. There are absolutely no changes required to the utilities, not even relinking. The scaffolding is a little more difficult, since a user-mode NFS server must be written. Operations which can’t be done through NFS, or which NFS clients turn into something else, won’t be seen by the simulation. For example, memory mapped files (and exec) will work, but the NFS operations are just reads and writes. For another example, there’s no way to simulate new semantics such as extended attributes, without extending the NFS protocol to support them. (Note that we would in all likelihood wish to do that eventually, for the real product, in any case.)

4.0 Design Overview

The fundamental idea behind the simulated xFS filesystem is that applications can use files in the simulated filesystem as though they were normal, real, kernel-implemented files.

User programs will link with a library replacing certain system calls (all the filesystem related ones) with wrappers. The wrappers will intercept calls aimed at files in the simulated filesystem. Each such call will result in a message being sent to a “monitor” process associated one-to-one with the user process. The monitor process is the bulk of the filesystem simulation.

The monitor contains all the xFS filesystem code as well as “scaffolding” code which replaces both the kernel services that the filesystem uses (page cache, memory allocation, locking) and the mechanisms to get from system calls to the filesystem code in a real kernel (syscall, vnodes, etc). The underlying disk driver accesses are replaced by writes to the disk space being used for the filesystem. The monitor processes for a given filesystem simulation share state in shared memory (and on disk) and communicate with each other as necessary.

The monitor process is created and destroyed by the library code linked with the user program. The local configuration - in particular, the mount point, disk space, etc. - is taken from environment variables by the library code, so that users of the simulation can share a single filesystem if they desire, and can have a private filesystem otherwise.

5.0 Design Details

How do we identify opens (pathnames) that are being simulated?

Assume that each filesystem simulation is started by a mkfs followed by a mount. The mount identifies the disk space(s) and the pathname to the simulation. Since creation of a new file type requires kernel changes, we will make the mount point be a symbolic link to a nonexistent name. Then we can catch pathname uses that fail, and look at them to see if the name matches a known mount point, and simulate the pathname use (open, chdir, etc.).

Actually since the current directory of a process is part of the process' real state, all pathname references that are relative to it must be checked as well. Issue: do we deal with symlinks into the simulation?

5.1 User Library Design

As mentioned above, we can make pathname references run the normal call then intercept the failure cases. We then maintain a "file table" for each file being simulated, so we can intercept system calls with file descriptor arguments. In order to keep the problem simple, we need to actually have a real open file descriptor behind each simulated file; the file descriptor doesn't need to refer to anything useful.

The user programs communicate with the monitor programs by some convenient mechanism; I suggest that pipes or System V messages or shared memory are adequate. I have no interest in dealing with sockets, and I don't think it's required that the simulation support a multi-machine environment either. The messages will need to encapsulate all the information that would be transmitted in a system call, including things like pathnames. The monitor side can be assumed to have as much state as the kernel side of a process, since that's what it's simulating.

5.2 Kernel System Call Side Design

The top level is the message receipt and breakout into individual routines, analogous to the syscall() code. This then gets us down into versions of the routines such as read(), ioctl(), etc. These can be rewritten for the simulation, in general; if it is easier in some cases to use the real code then we can do that. We will need to simulate the **u.** structure and process structures to support such code.

The next level takes us through vnode structures and the file system switch to the xFS code. The vnode support will need to be present intact; only xFS files will be supported, though.

5.3 Kernel Support Side Design

- Vnodes
- Memory Allocation
- Locking
- Coordination with other monitors: control data is in shared memory

What we need to do here is list the files that will be present intact (besides the actual filesystem code), and the routines that will be faked and stubbed to support them. The way this is done is to take the EFS code, and examine all the undefined symbols. For each such symbol, we either add the file that supplies it intact (thus generating more undefined symbols), or stub out the routine or variable. In rare cases we may want to add **#ifdefs** to the file. We repeat this procedure until it converges.

6.0 Implementation Schedule

- Implement and then test with EFS first. This should be done before the first integration and testing of xFS code could be done.