BSD Portals for LINUX 2.0

A. David McNab[†]
NAS Technical Report NAS-99-008

NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000
mcnab@nas.nasa.gov

Abstract

Portals, an experimental feature of 4.4BSD, extend the filesystem namespace by exporting certain `open(2)` requests to a user-space daemon. A portal daemon is mounted into the file namespace as if it were a standard filesystem. When the kernel resolves a pathname and encounters a portal mount point, the remainder of the path is passed to the portal daemon. Depending on the portal "pathname" and the daemon's configuration, some type of `open(2)` is performed. The resulting file descriptor is passed back to the kernel which eventually returns it to the user, to whom it appears that a "normal" `open(2)` has occurred. A proxy *portalfs* filesystem is responsible for kernel interaction with the daemon. The overall effect is that the portal daemon performs an `open(2)` on behalf of the kernel, possibly hiding substantial complexity from the calling process. One particularly useful application is implementing a *connection service* that allows simple scripts to open network sockets. This paper describes the implementation of portals for LINUX 2.0.

[†]David McNab is an employee of MRJ Technology Solutions, Inc.

1/ INTRODUCTION

One of the fundamental design commitments of UNIX is that all I/O devices can be represented as simple files in a hierarchical namespace. For example, disk devices can be addressed as long arrays of disk blocks through a device mapped into the filesystem. As networking became more integrated into the UNIX environment, it was natural to seek to extend this model to the network.

Programmatically, sockets are represented as file descriptors just as regular files are. However they require considerably different and more complex handling. For example to open a TCP socket to a particular port, one must first create the socket, then build a structure describing the relevant network address, then `connect(2)` the socket to the address, and finally begin exchanging data. In a flexible programming language with full access to the UNIX API this is not terribly complicated, and in most cases it can be automated by writing a more abstract `tcp_open()` function that hides the details. But other programming environments do not have access to the full UNIX API. For example the standard UNIX shells and tools like `awk(1)` can open and close files but present no interface to the necessary socket system calls.

Thus it is desirable, at the very least on an experimental basis, to explore the possibility of mapping the set of possible network connections into the filesystem. This would allow any process capable of the simplest of UNIX file I/O operations to open a network connection and read or write to it. It also renders the use of detail-hiding helper functions, as described above, unnecessary, since a programmer can simply open a standard filesystem pathname to gain access to the network (although in some cases this will not provide the necessary flexibility). Finally there is a certain aesthetic appeal in demonstrating that the elegant UNIX model can be extended to support a type of I/O for which it was not explicitly designed.

In 4.4BSD, an experimental feature called *portals* was introduced to meet these types of requirements. The work described here is a transfer of the portal concept to the LINUX 2.0 environment.

2/ OVERVIEW OF PORTALS

In 1985 Presotto and Ritchie described a *connection server*[4]: a service process that accepts requests to open connections of some sort, whether they be to a network resource or instructions to dial a modem and connect to a service provide, and then processes them on behalf of its clients. This has the desirable effect of hiding the details of the opening procedure from the client.

After a connection server does its work, it must have some mechanism by which it can transfer the open connection to the client. The infrastructure to support this was first introduced in 4.2BSD and is now implemented in almost all UNIX systems. The BSD UNIX `sendmsg(2)` and `recvmsg(2)` system calls can be used to pass open file descriptors across UNIX domain sockets. Thus a connection server can receive a request from a client, open the appropriate connection, then use `sendmsg()` to pass back the file descriptor. The client accepts it with `recvmsg()`, and the file descriptor can then be used to access the underlying connection. Conceptually, the server's file descriptor is a handle for the kernel data structures representing the connection. When the handle is passed to another process, the kernel structures remain unchanged but the recipient gains a reference to them.

4.4BSD portals use essentially the same mechanism, although the problem is complicated because in order to map the connection service into the filespace a kernel filesystem implementation is required. A *portal daemon* provides the connection service. This is a user-space background process that accepts connection requests, does the appropriate work, and returns an open file descriptor using `sendmsg()`. The kernel generates connection requests based on the pathname used to access a portal-space "file".

The mapping of "connection space" into the file namespace is performed by a filesystem called *portalfs*. Conceptually, portalfs is simple. The point at which the portal namespace is mounted indicates a border between the "normal" file namespace and the connection namespace. As the kernel processes `open()` requests it gradually resolves the pathname passed as an argument of `open()`. If a portal mount point is encountered, the resolution process stops. The portion of the pathname after the mount point is passed to the portal daemon. The daemon in-

terprets the path as a network connection specification, compares it with its configuration file to see whether the type of connection is supported, and then executes the `open()` on behalf of the kernel. The open file descriptor is passed back using `sendmsg()`, and the kernel effectively calls `recvmsg()` to extract the file descriptor. All of the communication between kernel and daemon process takes place over UNIX domain sockets, using a simple protocol. portalfs is responsible for accepting the unresolved portion of the pathname from the kernel, brokering the exchange with the daemon process, and returning the new file descriptor to the process that called `open()`.

Consider an example. We mount a portal daemon into the file namespace at `/p`, and a user subsequently opens `/p/tcp/foo.com/1795`. The kernel begins translating the pathname, but when it encounters `/p` it determines that a portal mount point has been crossed. The remainder of the pathname, `tcp/foo.com/1966`, is passed by portalfs to the portal daemon. This is interpreted as a request to open a TCP connection to host `foo.com`, accessing network port 1966. The daemon builds an address structure and performs the necessary `socket()` and `connect()` calls to set up the connection, then sends back the open file descriptor. Eventually the descriptor is passed back to the client process, which can now use it to access port 1966 on `foo.com`.

The implementation of 4.4BSD portals is described in detail by Pendry and Stevens[2]. The source code is also available via the freely available BSD implementations NetBSD and FreeBSD, as well as others.

3/ DESIGN AND IMPLEMENTATION

The primary design goal was to provide portal service and support the 4.4BSD portal daemon without substantial modification. The LINUX implementation of portals should appear to the portal daemon to be functionally the same as the BSD implementation. This allows us to take advantage of previous work in developing portal daemons.

A secondary goal was to limit changes to the LINUX kernel. Naturally the portal filesystem itself, previously unimplemented in LINUX, is new code, but the ideal was to avoid making any other changes to the OS. It turns out that some minor modifications were necessary, but they consist of a handful of additional lines of code in two kernel modules.

Another secondary goal was to ensure that the portal code did not introduce any additional instability to the LINUX kernel, nor that it could cause user processes to "lock up", in other words to sleep uninterruptibly while accessing portals.

3.1/ THE PORTAL DAEMON

The 4.4BSD portal daemon is implemented as child process of the command that mounts a portal filesystem. This program, `mount_portal(8)`, typically runs as part of the boot process. Its first action is to open a socket that will be used to listen for kernel requests. This *listen socket* is recorded as one of the portalfs-specific arguments to `mount(2)`, which is called to graft portalfs into the file namespace. If the mount succeeds, the program spawns a copy of itself which becomes the portal daemon. This daemon runs in the background for the remainder of the portal's lifetime, accepting incoming requests and spawning a copy of itself to process each of them. After spawning the daemon the parent exits, so that the `mount` command behaves synchronously as one would expect.

The `mount_portal` program used for the LINUX port was taken from release 1.3.1 of NetBSD. It supports two types of portals: *tcp* and *fs*. The *tcp* type maps pathnames of the form `tcp/<host>/<port>` into network connections, where `<host>` can be a hostname or an IP address and `<port>` is a port number. The *fs*-style portal simply re-maps pathnames of the form `fs/<path>` to `<path>`. It is intended to be used to support controlled egress from a `chroot(8)` environment. An extended version of the portal daemon could support filesystem extensions, for example access control lists. The NetBSD version of the daemon did not support the *tcplisten*-type portals described by Stevens and Pendry, hence it is not possible to implement TCP servers using this code[1].

The `mount_portal` program itself was well written for 4.4BSD and was relatively easy to port. The most sub-

[1] Of course since the LINUX implementation of portalfs is compatible with the BSD portal daemon protocol, an extant daemon that supports *tcplisten* portals can be incorporated relatively easily, requiring only a port of user-level code.

stantial change was required because old-style regular expression library routines were used to process the configuration file. LINUX did not support these routines well, so the program was modified to make POSIX-compliant calls. Other modifications were primarily to remove dependency on 4.4BSD's mount infrastructure, which provides utility functions for processing arguments, and minor changes to function calls arguments or header file inclusion.

There was what appeared to be a minor bug in the portal daemon's handling of *fs*-style portals. Files were always opened with mode `O_RDWR|O_CREAT`. This prevented reads from files to which the user did not have write permission; an unnecessary restriction. Fortunately, the portal credentials structure, which is passed from the kernel to the daemon and primarily consists of information about the *userid* and *groupid* of the process accessing the portal, also includes a field for the `open()` mode. By changing the `open()` call in the portal daemon to use the mode value from this field it was possible to correctly open read-only files. However because of restrictions in the mode information available to the kernel when it sends the portal request, all write-mode opens have the `O_CREAT` flag set.

A significant concern during design was whether the UNIX domain file descriptor passing code in the LINUX kernel worked properly. This is an obscure portion of the BSD networking system and the author considered it possible that the code was either not fully implemented or not correct in LINUX. To allay these concerns, a user-space portal test client was developed. This consisted of approximately 200 lines of C code intended to simulate the kernel's component of the interaction between portalfs and the portal daemon. Essentially it constructed a portal open specification based on command line arguments, sent the request to the portal daemon, accepted a file descriptor in return, and then read a small chunk of data from the file descriptor. Using this code it was possible to test and debug the portal daemon before the kernel code that would call it was developed. It turned out that the LINUX kernel performed the file descriptor passing correctly and the portal daemon behaved perfectly without additional modifications. This provided a certain peace of mind that was invaluable during kernel debugging.

3.2/ THE PORTAL FILESYSTEM

Any type of syntactic port of the 4.4BSD portalfs kernel code was impractical. Despite providing roughly similar functionality in many areas, the LINUX and BSD kernels are very different. Thus the "port" of portalfs actually consisted of establishing a thorough understanding of the BSD code and then deciding how to reconstruct similar behavior for LINUX.

The LINUX file system infrastructure does not provide vnodes, used in BSD to provide an abstraction of filestore-specific inodes. Vnodes store function vectors that describe the filesystem specific implementation of abstract operations. In LINUX, the `inode` structure acts as a vnode, in that it is the generic in-memory representation of a filesystem inode, but it is also used directly by some filesystems. A generic *Virtual Filesystem*, or VFS, provides skeletal filesystem functionality. Three sets of function calls can be provided by filesystem implementors to override or supplement the VFS calls[5]. The end result is similar to the vnode interface but less flexible and rigorous.

Superblock operations manage the filesystem superblock, providing facilities such as `statfs()`, and are responsible for correctly mounting and unmounting the filesystem. *Inode operations* are used to support generic filesystem object services, such as `link()` and `unlink()`, `create()`, `lookup()`, `mkdir()` and `rmdir()`, and so forth. Operations that affect the kernel's `file` structure, which is essentially the internal representation of an open file descriptor, are segregated as *file operations*. These consist of `lseek()`, `read()`, `write()`, `open()`, etc.

File system development in LINUX largely consists of deciding which of the members of these three sets of operations must be explicitly implemented for the new filesystem, in contrast with those for which the generic VFS functionality is sufficient. In the case of portalfs, most of the superblock operations are specially written, but only one each of the file and inode operations are required.

Superblock Operations

Five custom superblock operations are required to support portalfs. These primarily support mounting and unmount-

ing. `Portal_read_inode()` "reads" a portal inode—in fact there is a single inode to read, representing the root directory, and it is synthesized and stored in memory rather than being read from filestore. `Portal_write_inode()` consists of marking the inode clean, which should be a no-op but is included as a safety measure. `Portal_put_inode()` "releases" an inode that is no longer needed. For portalfs this consists of deallocating storage used to record the connection that is being opened. `Portal_put_super()` is called when the filesystem is unmounted and any superblock-related cleanup needs to be done. For portalfs this consists of releasing some kernel memory used to store data needed to find the portal daemon, specifically a socket pointer and a file reference. VFS mandates that portalfs also reset the superblock's affiliated device to zero, indicating "none". Finally, `portal_statfs()` synthesizes some reasonable-looking values for a `stat` structure, so that programs like `df(8)` will display portal filesystem data reasonably.

Inode Operations

The `open(2)` system call invokes two VFS operations: `lookup()` and `open()`. The former is called repeatedly to resolve the directory components of the pathname. The latter is called once, when the "end" of the pathname is reached. For the portal, filesystem `open()` is the workhorse, and as a file operation it is described in the next section.

VFS `lookup()` calls a filesystem-specific `lookup()` to do any special processing. In the case of `portal_lookup()`, this consists of recording the remaining pathname so that the file operation `open()` can subsequently retrieve it and send it to the portal daemon. This is done by allocating an inode and storing the pathname in it.

The generic LINUX inode contains a union structure that provides a place to store filesystem-specific data. One design option was to create a portal data structure and add it to the list in the union. However this required modifying the header file that specifies the `inode` structure. Fortunately, to provide support for run-time configurable kernel modules, LINUX designers included a generic `void*` pointer in the union. `portal_lookup()` allocates a chunk of kernel memory sufficient to store the remaining pathname, copies the path into it, and then sets the generic pointer to the address of the string. If the inode is freed using the superblock `put_inode()` operation, this storage is deallocated.

Unfortunately the actual implementation was not this simple. In 4.4BSD, the `lookup()` vnode operation takes as arguments the remainder of the pathname to be resolved and the address of a pointer to the pathname component to be resolved next. Thus the 4.4BSD portal code can extract the entire remainder of the pathname, record it, move the progress pointer to the end of the string, and return. This allows the filesystem to choose how much of the path to resolve and to interpret it in whatever way is useful.

In LINUX, `lookup()` takes as arguments a directory and a filename to look up in that directory, and it returns the resulting inode. In other words the LINUX VFS assumes that all pathnames are a sequence of directories, possibly ending with a file, and that they can be resolved one component at a time. For portalfs, this is a disastrous assumption, and in general it is needless and irksome. For example it presents a problem for developers of distributed filesystems, where a significant optimization might be to resolve an entire remote pathname in one operation.

Two design options presented themselves. The first is to allow the LINUX VFS to behave as it wants to, repeatedly calling `portal_lookup()` for each component of the "pathname". This adds significant complexity: portalfs now needs to keep track of the sequence of `lookup()` calls and build the full pathname as it goes. The alternative is to insert a snippet of code to the VFS implementation of `open()` so that if a portal mount point is crossed, the entire remainder of the pathname is passed as a single argument to `lookup()`. The latter implementation was chosen, despite the fact that it required a modification to the extant LINUX code—albeit a four line change. This choice was made primarily in the interest of minimizing complexity and is discussed further in the *commentary* section.

After this change to the kernel's `fs/namei.c` routine was made, the implementation of `portal_lookup()` was straightforward.

File Operations

The majority of the work required to implement portalfs comes in the `open()` file operation, implemented as `portal_open()`. By the time the kernel calls this routine, the portal "pathname"—that is, connection specification—has been recorded as auxiliary data in an inode allocated for this purpose. This inode is passed into `portal_open()`, along with a pointer to a pre-allocated `file` structure that will be used for the file being opened.

In the abstract, the work done by `portal_open()` is straightforward: extract the connection specification, send it to the daemon, wait for a response that includes a file descriptor, and then return that file descriptor. Note that this last step requires some skullduggery, described later.

In practice, the implementation of `portal_open()` is fairly complicated because a substantial amount of networking code had to be duplicated. As discussed in full detail in the *commentary* section, the LINUX implementation of UNIX domain sockets makes assumptions that limit its usefulness when called from elsewhere in the kernel. For example, it is assumed that certain data structures are stored in the user process's virtual memory context, hence they must be copied using special functions that fail when called on kernel-allocated memory.

The result is that in order to make the necessary `connect()`, `sendmsg()`, and `recvmsg()` calls, special versions of these functions had to be re-implemented inside portalfs. In practice this consisted of cutting out the UNIX domain implementations from the networking portion of the kernel and pasting them into the portalfs code. Much of the complexity of the full original calls was unneeded, since portalfs has very specific needs, and this code was excised. Where required, calling parameters were changed and assumptions about the location and type of memory being copied were changed. Nonetheless an undesirable duplication of code occurred, and if the standard implementations of UNIX domain `sendmsg()` and `recvmsg()` are ever changed the portal-specific versions may require modification. The portal versions are defined as statically scoped functions accessible only to `portal_open()`.

The final function of `portal_open()` is in returning the file descriptor acquired from the portal daemon to the user. In fact something slightly different happens. When `portal_open` is called, a new, empty file table entry has already been allocated for the calling process. Likewise, a file descriptor has been allocated to point to that file structure. This is the file descriptor that will be returned to the user. Hence `portal_open()` must copy the contents of the file entry corresponding to the portal file descriptor into the newly allocated file entry set up by `open()`. Then the descriptor and file entry originally returned from the daemon can be deallocated.

In implementing `portal_open()` the need for a second modification to the external kernel arose. In LINUX, networking protocols have affiliated vectors of functions similar to those used in the VFS. For example each protocol implements a *create()* method to implement the protocol-specific details of the `socket(2)` system call. Since `portal_open()` needs to allocate a UNIX domain socket, it was necessary to call `unix_create()`, the UNIX domain protocol-specific initialization method. However these functions are defined with static scope and accessible only to the implementation of sockets. Thus once again the design decision was whether to make a change to the kernel at large or to duplicate a chunk of kernel networking code inside the portalfs implementation. In this case, unlike those of `sendmsg()` and `recvmsg()`, the extant kernel function worked correctly when called from the portal code. Hence it seemed more natural to modify the networking code to export the protocol-specific functions than to duplicate code. This was done by adding a three line function to `net/socket.c` that takes as an argument a protocol specification and returns a pointer to a structure containing the operations specific to that protocol.

3.3/ IMPLEMENTATION SUMMARY

Excluding the implementation of portalfs itself, only two functional changes were made to the LINUX kernel. Four lines were added to `fs/namei.c` to short-circuit pathname resolution in the special case where a portal mount point is crossed, and three lines were added to `net/socket.c` to export the protocol specific method vector. There were also minor changes to the build infrastructure to support the portal changes, for example adding a portal filesystem type and adding a kernel configuration variable to specify whether or not portals are to be

included in a build.

portalfs itself was implemented in three C files, one for each type of operation. `super.c` implements the superblock operations and is 310 lines long. `inode.c` implements `portal_lookup()` and consists of 131 lines of code. `file.c` implements the more complicated `portal_open()` file operation and is 542 lines long. Overall approximately 1000 lines of kernel code (comments included) were required to implement portalfs.

The administrative `mount_portal` command and portal daemon were ported from 4.4BSD and required on the order of thirty lines of changes, mostly dealing with regular expression processing.

4/ RESULTS

The primary design goal was satisfied. The BSD portal daemon was ported without major changes and works correctly as the connection service for the LINUX portal implementation. It is now possible to mount a portal filesystem and open network connections in LINUX in exactly the same way as takes place in 4.4BSD UNIX. The *fs*-type portals also work correctly, with the exception that the user's *umask* is ignored. This could be corrected but would require modifying the portal credentials structure to include *umask* data. Created files do have the correct user and group ownership, and the correct access controls are enforced.

One secondary design goal—minimizing changes to the LINUX kernel—was achieved. Only seven lines of code were added to the LINUX kernel. The other secondary goal, avoiding introducing new kernel instability, also seems to have be satisfied, although the portal implementation has not been heavily tested or even used by anyone but the designer. It is possible that there are resource leaks or corner-case errors yet to be discovered. The basic implementation does not cause any kernel faults.

Performance tests were minimal. To ensure that the cost of opening a portal socket is not prohibitive relative to the cost of making direct socket system calls, two programs were written. The first was a shell script that repeatedly read from the *daytime* service on the localhost using `cat < /p/tcp/localhost/daytime`, where `/p` was the portal mount point. The second was a short C program that did the same thing using socket system calls. The shell version completed 100 reads in 2.05 seconds, with user CPU consumption of 0.44 seconds and system CPU consumption of 0.41 seconds[2]. The C program performed the same number of reads in 0.52 seconds, consuming only 0.03 seconds of user CPU time and 0.07 seconds of system CPU time.

Thus initially accessing a network resource via a portal takes on the order of four times the elapsed time and consumes roughly an order of magnitude more CPU time than using sockets directly. These are very crude estimates intended only to give a rough idea of portal overhead. Of course these numbers are of dubious importance anyway, since they represent a one-time cost for opening the portal. All subsequent access occurs as if the user process had directly opened the file or socket directly, thus there is no overhead.

5/ COMMENTARY ON LINUX

LINUX is a young operating system, and direct comparison with the more mature BSD code base is perhaps unfair. There certainly appear to be parts of the LINUX kernel that would benefit from an infusion of the design expertise developed over the nearly twenty years of BSD's lifetime—for example the introduction of a vnode abstraction. If the LINUX development model scales well, architectural changes like this should take place evolutionarily as the current design encounters limitations.

However there is a different class of problem that became apparent during the portalfs work. This is not so much an issue of operating system design—which should work itself out if the LINUX development model is valid—but rather issues relating to dubious programming methodology. That is, these are engineering concerns that apply to any large software system, regardless of its application.

One example is a failure to rigorously adhere to the principles of functional programming, specifically a tendency to group several clearly orthogonal functions into a single large subroutine. This makes it impossible to use

[2]This excluded the overhead of a shell loop, which was measured independently and subtracted from the test results. It was not possible to account for the time consumed by the portal daemon on behalf of the user.

some a proper subset of those functions. An example was in the implementation of `portal_connect()`. During the portalfs development one requirement was to take a pair of `socket` structures and connect them. This is exactly the functionality provided by the UNIX domain socket protocol operation `unix_connect()`, except that `unix_connect()` takes a socket and an address structure as arguments instead of two sockets. In other words the conversion of the address to a socket structure, which is completely independent of the logic necessary to connect two sockets, is hardwired into the same function that does the connect. A similar and perhaps more vexing problem was the failure to separate the orthogonal functions of copying-in data from the user's address space and conducting socket functions using that data. This is the design error that forced the duplication the `sendmsg()` and `recvmsg()` code with minor changes pertaining only to the source of the data upon which they were operating.

A second problem is excessive and unnecessary use of internal linkage, i.e. the declaration of kernel functions as `static`. In some cases there is no conceivable use for a "helper" function and a static declaration is clearly in order. This is the case in the implementation of `portal_open()`, which uses its own statically declared versions of `sendmsg()` and `recvmsg()`. These are special purpose "hacks" that clearly should not be exported to the rest of the kernel. However there were several cases where useful data or functionality was declared static for no apparent reason. One can infer that there was a concern over namespace pollution, in which case perhaps a subsystem naming convention should be introduced, or perhaps the code author simply failed to consider the possibility that the function would be useful elsewhere. A prime example was in failing to provide any mechanism to access protocol specific operations outside of the socket implementation code.

The LINUX kernel appears very much to have been written to service requests from user-space. Of course this is the primary function of an operating system—to provide a virtual machine abstraction for userspace applications. However it is often the case that a piece of code useful for application level software will also be useful to kernel code in another subsystem. Careful design of kernel subsystem abstractions and intrakernel programming interfaces is important for keeping an operating system flexible. This is one design element of the BSD-based kernels that makes them attractive to OS scholars and developers, and both of the methodological problem areas described above suggest than it is an area to which LINUX designers could pay more attention.

6/ RELATED WORK

The original BSD implementation of portals is described in detail by Stevens and Pendry[2] and briefly in the 4.4BSD book[3], the latter also describing the general BSD file system infrastructure.

Presotto and Ritchie describe the general idea of a connection server[4], and Stevens later provides an example of such a system using UNIX domain socket passing[6].

The Sprite operating system provide *pseudo-devices*, which map user programs into the file space[7].

Plan 9 extended the UNIX I/O model to the network by building networking into the kernel in a more fundamental way than BSD did with sockets. All Plan 9 I/O, including networking, is done through the file namespace. User processes have access to detailed network information either directly through the file system or through a connection server called *cs*, which is also mapped into the file namespace[8].

7/ FUTURE WORK

One avenue for future work is expanding the functionality of the portal daemon, for example adding *tcplisten* capabilities so that scripts can use portals to act as network servers. Another possibility would be modifying the *fs* server to implement access control lists, as suggested by Stevens and Pendry.

The `mount_portal(8)` program could be better integrated into the suite of LINUX *mount* programs, so that it conforms to LINUX standards.

The basic technique of using UNIX domain sockets inside the kernel to communicate with user-space processes presents some interesting possibilities. For example it may be possible to build a general purpose proxy filesystem. This would look to external processes to be a "normal" filesystem, but internally it would direct all requests for inodes, directory entries, and other filesystem data

to an external daemon. In other words a daemon process would act as the filestore portion of the filesystem. The daemon could synthesize data as needed, could perform transformations on extant data, or could retrieve the data from other hosts. This would support out-of-kernel filesystem development, although it introduces substantial safety concerns—the proxy file system would have to be careful to ensure that data structures returned from the daemon were consistent and would not introduce problems.

8/ CONCLUSION

The goals of this work were to support BSD portal processes in a stable LINUX environment with as few code modifications as possible. These were met. Only seven lines of kernel code were changed, and the LINUX implementation of portals works correctly with the ported but basically unmodified BSD portal daemon. Portals provide network access for scripts and programs that do not have access to the full UNIX networking API. The extent to which this is practically useful is an open question, but given an implementation with which to experiment the user community should provide the answer.

Bibliography

[1] Joy, Cooper, Fabry, Leffler, McKusick & Mosher, *4.2BSD System Manual*, *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Volume 2C, Computer Systems Research Group, Univ. of California, Berkeley, CA; 1983.

[2] R. Stevens & J. Pendry, "Portals in 4.4BSD," USENIX Conference Proceedings, pp. 1-10, January 1995.

[3] McKusick, Bostic, Karels, & Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, pp. 237-8 (portals) and general material, Copyright ©1996 by Addison-Wesley Publishing Company, Inc.

[4] D. L. Presotto & D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *Proceedings of the 1985 Summer USENIX Conference*, Portland OR; 1985.

[5] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, & D. Verworner, *Linux Kernel Internals, Second Edition*, Copyright ©1998 Addison Wesley Longman, ISBN 0-201-33143-8.

[6] W. R. Stevens, *UNIX Network Programming*, Copyright ©1990 by Prentice-Hall.

[7] B. B. Welch & J. K. Ousterhout, "Pseudo Devices: User-Level Extensions to the Sprite Filesystem," *Proceedings of the 1988 Summer USENIX Conference*, pp. 37-49, San Francisco CA (1988).

[8] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom, "Plan 9 from Bell Labs," *Plan 9: The Documents*, Copyright ©1995 by AT&T, pp. 1-22.