puffs - Pass-to-Userspace Framework File System

Antti Kantee <pooka@cs.hut.fi>
Helsinki University of Technology

ABSTRACT

Fault tolerant and secure operating systems are a worthwhile goal. A known method for accomplishing fault tolerance and security is isolation. This means running separate operating system services in separate protection domains so that they cannot interfere with each other, and can communicate only via well-defined messaging interfaces. Isolation and message passing brings inherent overhead when compared to services doing communication by accessing each others memory directly. To address this, the ultimate goal would be to be able to run the kernel subsystems in separate domains during development and testing, but have a drop-in availability to make them run in kernel mode for performance critical application scenarios. Still today, most operating systems are written purely with C and some assembly using the monolithic kernel approach, where all operating system code runs within a single protection domain. A single error in any subsystem can bring the whole operating system down.

This work presents *puffs* - the Pass-to-Userspace Framework File System - shipped with the NetBSD Operating System. It is a framework for implementing file systems outside of the kernel in a separate protection domain in a user process. The implementation is discussed in-depth for a kernel programmer audience. The benefits in implementation simplicity and increased security and fault tolerance are argued to outweigh the measured overhead when compared with a classic in-kernel file system. A concrete result of the work is a completely BSD-licensed sshfs implementation.

Keywords: userspace file systems, robust and secure operating systems, message-passing subsystems, BSD-licensed sshfs

1. Introduction

"Microkernels have won", is a famous quote from the Tanenbaum - Torvalds debate from the early 90's. Microkernel operating systems are associated with running the operating system services, such as file systems and networking protocols, in separate domains, and component communication via message passing through channels instead of direct memory references. This is known as isolation and provides an increase in system security and reliability in case of a misbehaving component [1]; at worst the component can corrupt only itself instead of the entire system. However, most contemporary operating systems still run all services inside a single protection domain with the popular argument being an advantage in performance. Even if we were to disregard research which states that the performance difference is irrelevant [2], we might be willing to make a tradeoff for a more robust system.

A separate argument is that we do not need to see issues only in black-and-white. An operating system's core can be monolithic with the associated tradeoffs, but offer the interfaces to implement some services in separate domains. An HTTP server or an NFS server can be implemented either as part of the monolithic kernel or as a separate user process, even though they both have their "correct" locations of implementation.

There is obviously room for both a microkernel and a monolithic kernel approach within the same operating system. Another relevant argument is the use of inline assembly in an operating system: almost everyone agrees that it is wrong, yet not using it makes the system less performant. Clearly, performance is not everything.

This work presents *puffs*, the Pass-To-Userspace Framework File System for NetBSD. *puffs* provides an interface similar to the kernel virtual file system interface, vfs [3], to a user process. *puffs* attaches itself to the kernel vfs layer. It passes requests it receives from the vfs interface in the kernel to userspace, waits for a result and provides the caller with the result. Applications and the rest of the kernel outside of the vfs module cannot distinguish a file system implemented on top of *puffs* from a file system implemented purely in the kernel. For the userspace implementation a library, libpuffs, is provided. libpuffs not only provides a programming interface to implement the file system on, but also includes convenience routines commonly required for implementing file systems.

puffs is envisioned to be a step in moving towards a more flexible NetBSD operating system. It clearly adds a microkernel touch with the associated implications for isolation and robustness, but also provides an environment in which programming a file system is much easier than compared to the same task done in the kernel. And instead of just creating a userspace file system framework, the lessons learned from doing so will be turned upside down and the whole system will also be improved to better facilitate creating functionality such as *puffs*. The latter part, however, is out of the scope of this paper.

Related Work

There are several other packages available for building file systems in userspace. When this project was begun in the summer of 2005, the only option available for BSD was nnpfs, which is supplied as part of the Arla [4] AFS implementation. Arla is a portable implementation of AFS. It relies on a small kernel module, nnpfs, which attaches to the host operating system's kernel and provides an interface for the actual userspace AFS implementation to talk to. A huge drawback was that at the time it only supported caching on a file level. Since, it has developed block level caching and some documentation on how to write file systems on top of it [5].

The best known userspace file system framework is FUSE, Filesystem in USErspace [6]. It supports already hundreds of file systems written against it. On a technical level, *puffs* is fairly similar to FUSE, since they both export similar virtual file system interfaces to userspace. However, the are differences already currently in, for example, pathname handling and concurrency control. The differences are expected to grow as the *puffs* project reaches future goals. Even so, providing a source compatible interface with FUSE is an important goal to leverage all the existing file systems (see Chapter 5). In the summer of 2005 FUSE was available only for Linux, but has since been ported to FreeBSD in the Fuse4BSD [7] project. A derivate project of the FreeBSD porting effort, MacFUSE [8], recently added support for Mac OS X. A downside from the BSD point-of-view is that userspace library for FUSE is available only under LGPL and that file systems written on top of it have a tendency of being GPL-licensed.

Apart from frameworks merely exporting the Unix-style vfs/vnode interface to userspace for file system implementation, there are systems which completely redesign the whole concept. Plan 9 is Bell Labs' operating system where the adage "everything is a file" really holds: there are no special system calls for services like there are on Unix-style operating systems, where, for example, opening a network connection requires a special type of system call. Plan 9 was also designed to be a distributed operating system, so all the file operations are encoded in such a way that a remote machine can decode them. As a roughly equivalent counterpart to the Unix virtual file system, Plan 9 provides the 9P [9] transport protocol, which is used by clients to communicate with file servers. 9P has been adapted to for example Linux [10], but the greater problem with 9P is that it is relatively different from the (Net)BSD vfs interface and it makes some assumptions about file systems in general not valid on Unix [10]. Therefore, it was not directly considered for the userspace library interface.

DragonFly BSD has started putting forth effort in creating a VFS transport protocol, which, like 9P, would be suitable for distributed environments in which the server can exist on a different network node than the client [11]. It is also usable for implementing a file system in userspace, but is a huge undertaking and restructures much of the kernel file system code.

The main reason for writing a framework from scratch is that the ultimate goal of the work is not to develop a userspace file system framework, but rather to improve the flexibility and robustness of the operating system itself. While taking a more flexible route such as that of 9P may eventually prove to be the right thing to do, it is easier to take n small steps in reaching a goal and keep the system functional all the time. Currently, especially the kernel side of *puffs* is very lightweight and tries to be a good kernel citizen in not modifying the rest of the kernel. The ultimate goal is to gradually change this in creating a more secure and reliable operating system.

Paper Contents

Chapter 2 discusses the architecture and implementation of *puffs* on an in-depth technical level. Chapter 3 presents a few file systems built on top of *puffs*. It discusses experiences in developing them. Chapter 4 presents performance measurements and analyses the measured results. Chapter 5 contains work being done currently and outlines some future visions for development. Finally, Chapter 6 provides conclusions.

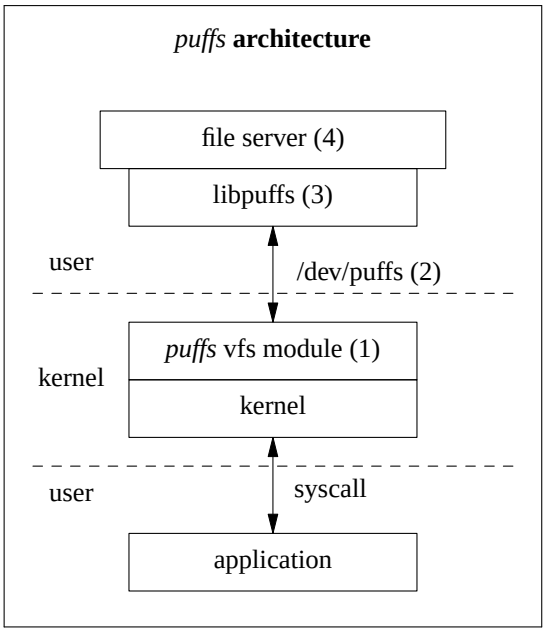
2. *puffs* Architecture

puffs is made up of four separate components (see figure):

1. VFS attachment, including virtual memory subsystem and page cache integration. This part interfaces with the kernel and makes sure that the kernel correctness is enforced. (Chapter 2.1.)
2. Messaging interface, which transports requests to and from the file system server. (Chapter 2.2.)
3. A user level adaption library, libpuffs, which handles the details of the kernel communication and provides supporting routines. (Chapter 2.3.)
4. The file system implementations themselves. (Chapter 3)

2.1. Virtual File System Attachment

Creating a new file system in the kernel is done by attaching it to the kernel's virtual file system (vfs) [3] interface. As long as the file system abides by the vfs layer's call protocols, it is free to provide the kind of file hierarchy and data content it wishes.



The vfs layer is made up of two separate interfaces: the actual virtual file system interface and the vnode interface. The former deals with calls involving file system level operations, such as mount and unmount, while the latter always involves an operation on a file; the vnode or virtual node is an abstract, i.e. virtual, representation of a file.

Vnodes are treated as reference counted objects by the kernel. Once the reference count for a vnode drops to zero, it is moved to the freelist and said to enter an *inactive* state. However, the file system in-memory data structures may still hold weak pointers to the vnode at this point and some vnode operations may prompt the file system to attempt to rescue the vnode from the freelist. Once a vnode is irreversibly freed and recycled for other use, it is said to be *reclaimed*. At this point a file system must invalidate all pointers to the vnode and in-memory file system specific data structures relating to the vnode are also freed [12].

A very central routine for every file system is the *lookup* routine in the vnode interface. This routine takes in a pathname component and produces a vnode. It must return the same vnode for the duration of the vnode's lifetime, or else the kernel could access the same file through multiple different interfaces destroying consistency guarantees. *puffs* uses cookie values to map node information between the kernel and the file server. The file server selects a cookie value and

communicates it to the kernel upon node creation[1]. The kernel checks that it was not handed a duplicate, creates a new vnode and stores the cookie value in the private portion of the newly created vnode. This cookie value is passed to the file server for all subsequent operations on the kernel vnode. A *cookie* → *vnode* mapping is also stored in a hash list so that *lookup* can later determine if it should create a new vnode or if it should return the an existing one.

The cookie shared by the file server and kernel is of type `void *`. While this is not enough to cover all file system nodes on a 32bit architecture, it should be recalled that the cookie value is used only to locate an in-memory file system data structure and is valid only from node creation to the reclaim operation and that this cycle is controlled by the kernel. Most file servers will simply use the address of the in-memory data structure as the cookie value and do mapping from the cookie to the file server node structure with a simple pointer typecast. Even further, this address will be that of a generic libpuffs node, `struct puffs_node`, and the file system's private data structure can be found from the private data pointer in `struct puffs_node`. This is not required, but as we will later see when discussing the user library, the generic node provides some additional convenience features.

For interfacing between the kernel and the file server, the vfs layer acts as a translator between the in-kernel representation for vfs parameters and a serialized representation for the file server. This part is discussed further in Chapter 2.2. Additionally, the vnode portion of the vfs attachment implements the file system side of the vnode locking protocol.

The vfs layer also acts as a semantic police between the kernel and the user fs server. It makes sure that the file server does not return anything which the rest of the kernel cannot handle and would result in incorrect operation, data corruption or a crash.

Short circuiting Non-implemented Operations

All user file system servers do not implement all of the possible operations; open and close are examples of operations commonly not implemented at all on the vnode level. Therefore, unless mounted with the debug flag `PUFFS_KFLAG_ALLOPS`, operations unsupported by the file server will be short circuited in the kernel. To avoid littering operations with a check for a supported operation, the default vnode operations vector, *puffs_vnodeop_p*, defines some operations to be implemented by *puffs_checkop*(). This performs a table lookup to check if the operation is supported. If the operation is supported, the routine makes a `VOCALL`() for the operation from the vector *puffs_msgop_p* to communicate with the file server. Otherwise it returns immediately. To make this approach feasible, the script generating the vnode interface was modified to produce symbolic names for the operations, e.g. `VOP_READDIR_DESCOFFSET`, where they were previously generated only as numeric values. It should be noted that all operations cannot be directed to *puffs_checkop*(), since e.g. the reclaim operation must do in-kernel bookkeeping regardless of if the file server supports the operation in question. These operations use the macro `EXISTSOP`() to check if they need to contact the file server or is in-kernel maintenance enough.

puffs vnode op vector

```
{&vop_lookup_desc, puffs_lookup },
{&vop_create_desc, puffs_checkop },
{&vop_mknod_desc, puffs_checkop },
{&vop_open_desc, puffs_checkop },
 ...
{&vop_reclaim_desc, puffs_reclaim },
{&vop_lock_desc, puffs_lock },
{&vop_unlock_desc, puffs_unlock },
```

Kernel Caching

Caching relatively frequently required information in the kernel helps reduce roundtrips to the fs server, since operations can be short circuited already inside the kernel and cached data provided to the caller. Caching is normal behavior even for in-kernel file systems, as disk I/O is very slow compared to memory access.

The file system cache is divided into three separate caches: the page cache, the buffer cache and the name cache. The page cache [13] is a feature of the virtual memory subsystem and caches file contents. This avoids reading the contents of frequently used files from the backing storage. The buffer cache in turn [12,14] operates on disk blocks and is meant for file system metadata. The

[1] A node can be created by the following operations: *lookup*, *create*, *mknod*, *mkdir* and *symlink*. The first one just creates the node, while the final four create the backing file and the node.

name cache [12,15] is used to cache the results of the lookup from pathname to file system node to avoid the slow path of the frequent *VOP_LOOKUP*() operation.

To avoid doing expensive reads from the file server each time data is accessed, *puffs* utilizes the page cache like any other file system would. Additionally, it provides the file server with an interface to either flush or invalidate the page cache contents for a certain file for a given page range. These facilities can be used by file servers which use backends with distributed access. Since *puffs* does not operate on a block device in the kernel, it does not use the buffer cache at all. However, caching metadata is advantageous [16] even if it is not backed up by a block device. Support for caching metadata in the kernel is planned in the near future. Finally, *puffs* uses the name cache as any other file system would, but additionally provides the file server with a method to invalidate the name cache either on a per-file basis, per-directory basis or for the entire file system.

2.2. User-Kernel Messaging Interface

Messaging between the kernel and file server is done through a character device. Each file server opens `/dev/puffs` at mount time and the communication between the file server and kernel is done through the device. The only exception is mounting the file system, for which the initial stage is done by the file server by calling the *mount*() system call. Immediately when the device descriptor is closed the file system is forcibly unmounted in the kernel, as the file server is considered dead. This is an easy way to unmount a misbehaving file system, although normally `umount` should be preferred to make sure that all caches are flushed.

VFS and Vnode Operations

All vfs and vnode operations are initiated in the kernel, usually as the result of a process doing a system call involving a file in the file system. Most operations follow a query-response format. This means that when a kernel interface is called, the operation is serialized and queued for transport to the file server. The calling kernel context is then put to sleep until a response arrives (or the file system is forcibly unmounted). However, some operations do not require a response from the file server. Examples of such operations are the vnode reclaim operation and fsync not called with the flag `FSYNC_WAIT`. These operations are enqueued on the transport queue after which the caller of the operation continues executing. *puffs* calls these non-blocking type operations Fire-And-Forget (FAF) operations.

Before messages can be enqueued, they must be transformed to a format suitable for transport to userspace. The current solution is to represent parameters of the operation as structure members. Some members can be assigned directly, but others such as `struct componentname` must be translated because of pointers and other members the userland does not have direct access to. Currently all this modifying is done manually for each operation, but it is hoped that this could be avoided in the future with an operation description language.

Transport

As mentioned above, the format of messages exchanged between the kernel and file server is defined by structures. Every request structure is subclassed from `struct puffs_req`, which in C means that every structure describing a message contains the aforementioned structure as its first member. This member describes the operation enough so that it can be transported and decoded.

puffs_req members

```
struct puffs_req {
    uint64_t preq_id;

    union u {
        struct {
            uint8_t opclass;
            uint8_t optype;
            void   *cookie;
        } out;
        struct {
            int     rv;
            void   *buf;
        } in;
    } u;
    size_t  preq_buflen;
    uint8_t preq_buf[0]
      __aligned(ALIGNBYTES+1);
};
```

The messaging is designed so that each request can be handled by in-place modification of the buffer. For most operations the request

structures contain fields which should be filled, but the operations *read* and *readdir* may return much more data so it is not sensible to include this space in the structure. Conversely, *write* does not need to return all the data passed to userspace.

puffs_vnreq_read/_write

```
struct puffs_vnreq_readwrite {
    struct puffs_req  pvn_pr;

    struct puffs_cred pvnr_cred;
    off_t             pvnr_offset;
    size_t            pvnr_resid;
    int               pvnr_ioflag;

    uint8_t           pvnr_data[0];
};
```

When querying for requests from the kernel, the file server provides a pointer to a flat buffer along with the size of the buffer. The kernel places requests in this buffer either until the next operation would not fit in the buffer or the queue of waiting operations is empty. To facilitate in-place modification for operations which require more space in the response than in the query (read, readdir), the kernel leaves a gap which can fit the maximal response.

This solution, however, is suboptimal. It was designed before the continuation framework (see Chapter 2.3) and does not take into account that the whole flat buffer is not available every time a query is made. The currently implemented workaround is to *memcpy*() the requests from the buffer into storage allocated separately for the processing of each operation. To fix this, the query operation will eventually be modified to use a set of buffers instead of one big buffer.

Responses from the user to the kernel use a scatter-gather type buffering scheme. This facilitates both operations which return less or more data than what was passed to them by the kernel and also operations which do not require a response at all. To minimize cross-boundary copy setup costs, the ioctl argument structure contains the address information of the first response. The `puffs_req` in the first response buffer contains the information for the second response buffer and so forth. This way only one copyin is needed per buffer instead of one for the header describing how much to copy from where and one for the buffer itself.

Snapshots

puffs supports building a snapshotting file system. What this means is that it supports the necessary functionality to suspend the file system temporarily into a state in which the file system server code can take a snapshot of the file system's state. Denying all access to the file system for the duration of taking the snapshot is easy: the file system server needs only to stop processing requests from the kernel. This is because, unlike in the kernel, all requests come through a single interface: the request queue. However, the problem is flushing all cached data from the kernel so that the file system is in a consistent state and disallowing new requests from entering the request queue while the kernel is flushing the information.

NetBSD provides file system suspension routines [17] for implementing suspending and snapshotting a file system within the kernel. These helper routines are designed to block any callers trying to modify the file system after suspension has begun and before all the cached information has been flushed. Once all caches have been flushed, the file system enters a suspended state where all writes are blocked. After a snapshot has been taken, normal operation is resumed and blocked writers are allowed to continue. Note that using these synchronization routines is left up to the file system, since generic routines cannot know where the file system will do writes to backing storage and where not.

puffs utilizes these routines much in the same fashion as an in-kernel file systems would. A file server can issue a suspend request to the kernel module. This causes the kernel vfs module to block all new access to the file system and flush all cached data. The kernel uses four different operations to notify the file server about the progress in suspending the file system. First, `PUFFS_SUSPEND_START` is inserted at the end of the operations queue to signal that only flushing operations will be coming from this point on. Second, when all the caches have been flushed, `PUFFS_SUSPEND_SUSPENDED` is issued to signal that the kernel is now quiescent. Note that at this point the file system server must still take care that it has completed all operations blocked with the continuation functionality or running in other threads and can only then proceed to take a clean snapshot. Finally, the kernel issues an explicit `PUFFS_SUSPEND_RESUME`, even though it always follows the suspend notification. In case of an error while attempting to suspend,

the kernel issues PUFFS_SUSPEND_ERROR. This also signals that the file system continues normal operation from the next request onwards.

2.3. User Level Library

The main purpose of the user library, libpuffs, is to take care of all details irrelevant for the file system implementation such as memory management for kernel operation fetch buffers and decoding the fetched operations.

The library offers essentially two modes of operation. The file server can either give total control to the library by calling *puffs_mainloop*(), or invoke the library only during points it chooses to with the *puffs_req* family of functions. The former is suited for file systems which handle all operations without blocking while the latter is meant for file systems which need to listen multiple sources of input for asynchronous I/O purposes. Currently, the library does not support a programming model where the library issues a separate worker thread to handle each request.

Interface

The current *puffs* library interface closely resembles the in-kernel virtual file system interface. The file server registers callbacks to the library for operations and these callbacks get executed when a request related to the callback arrives from the kernel.

For file system operations, only three operations from vfsops are exported: sync, statvfs and unmount. The sync callback is meant to signal the file server to synchronize its state to backing storage, statvfs is meant to return statistics about the file system, and unmount tells the file server that the kernel has requested to unmount the file system. The user server can still fail an unmount request which was not issued with MNT_FORCE. The kernel will respect this.

The operations dealing with file system nodes are greater in number, but some operations are missing when compared to the kernel vnode interface. For example, the kernel uses *VOP_GETPAGES*() and *VOP_PUTPAGES*() for integration with the virtual memory subsystem[2] and as a backend for *VOP_READ*() and *VOP_WRITE*() on most file systems. However, since *puffs* userspace file servers do not integrate into the kernel virtual memory subsystem, they do not need *VOP_GETPAGES*() and *VOP_PUTPAGES*() and can simply make do with read and write.

The parameters for the node operations follow in-kernel vnode operations fairly closely. Operations are given an opaque library call context pointer, pcc, and the operation cookie, opc, which the file server can use to find its internal data structure. The meaning of the operation cookie depends on each operation, but it is either the directory which the operation affects or the node itself if the operation is not a directory operation. For example, in the signature of rmdir, the operation cookie is the cookie of the directory from which the file is supposed to be removed from, targ is the cookie of the node to be removed and pcn describes the directory entry to remove from the directory.

puffs_node_rmdir

```
int
node_rmdir(struct puffs_cc *pcc,
    void *opc, void *targ,
    const struct puffs_cn *pcn);
```

Full descriptions of each operation and involved parameters can be found from the *puffs* manual pages [18].

Filenames and Paths

The kernel vnode layer has only minimal involvement with file names. Most importantly, the vnode does not contain a pathname. This has several benefits. First, it avoids confusion with hardlinks where there are several pathnames referring to a single file. Second, it makes directory rename a cheap operation, since the pathnames of all nodes under the given directory do not need to be modified. Only operations which require a pathname component are passed one. Examples are lookup, create and rmdir. The latter two require the pathname component to know what is the name of the directory entry they should modify.

However, most file system backends operate on paths and filenames. Examples include the sftp backend used by psshfs and the puffs null layer (discussed further in Chapter 3.1). To

[2] In NetBSD, file system read and write are commonly implemented as *uiomove*() on a kernel memory window. getpages is used to bring file data into memory while putpages is used to flush it to storage. This is how the file data is cached into the page cache and written from it.

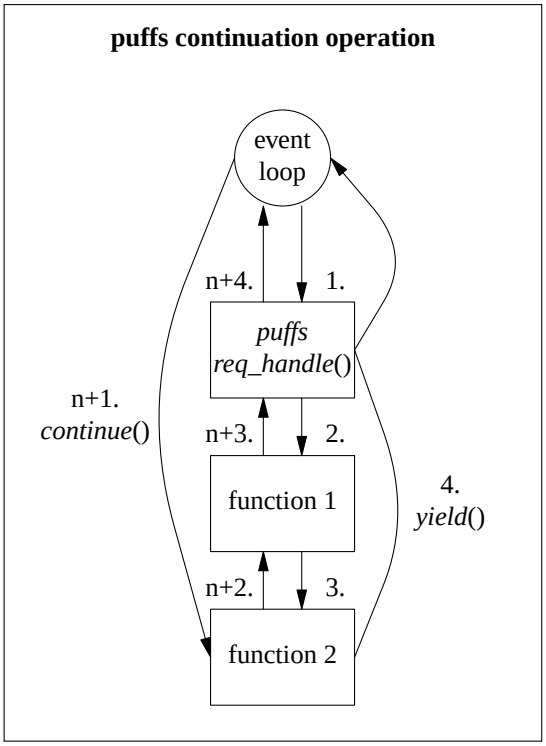
facilitate easier implementation of these file systems, *puffs* provides the mount flag `PUFFS_FLAG_BUILDPATH` to include full pathnames[3] in componentnames passed to interface functions as well as store the full path in `struct puffs_node` for use by the file server. In addition to providing automatic support for building pathnames, *puffs* also provides hooks for file systems to register their own routines for pathname building in case a file system happens to support an alternative pathname scheme. An example of this is sysctlfs (Chapter 3.1), which uses sysctl MIB names as the pathnames stored in `struct puffs_node`s. This alternate scheme helps keep pathnames in the same place as other file systems, but it requires some extra effort from the file system: the file system must itself complete the path in routines such as lookup after it figures out its internal representation for the pathname component; file systems based on "regular" pathnames do not require this extra burden.

The advantage of having pathnames as an optional feature provided by the framework is that file servers implemented more in the style of classical file system do not need to concern themselves unnecessarily with the hassle of dealing with pathnames, and yet backends which require pathnames have then readily available. The framework also handles directory renames and modifies the pathnames of all child nodes of a renamed directory.

Continuations

libpuffs operates purely as a single threaded program. The question between the preference for an event loop or multiple threads is mostly an open question and the conscious decision was to in no way bias the implementation in such a fashion that threading with all its uncertainties [19] would be required to create a working file system which does not block while waiting for operations to complete.

The *puffs* solution is to provide a continuation framework in the library. Multitasking with continuations is like multitasking with cooperative threads: the program must explicitly indicate scheduling points. In a file system these scheduling points are usually very clear and similar to the kernel: a yield happens when the file system has issued an I/O operation and starts waiting for the result. Conversely, a continue is issued once the result has been produced. This also bears resemblance to how the in-kernel file systems operate (*ltsleep*()/*wakeup*() and the buffer cache operations *biowait*()/*biodone*()) and should provide a much better standing point for running unmodified kernel file systems under *puffs* than relying on thread scheduling.



The programming interface is extremely simple. The library provides an opaque cookie, `struct puffs_cc *pcc`, with each interface operation. The file system can put itself to sleep by calling *puffs_cc_yield*() with the cookie as the argument and resume execution from the yield point with *puffs_cc_continue*(). Before yielding, the file system must of course store the `pcc` in its internal data structures so that it knows where to continue from once the correct outside event arrives. This is further demonstrated in the above figure and also Chapter 3.1, where the *puffs* ssh file system is discussed.

However, since the worker thread model is useful for example in situations where the file system must call third party code and does not have a chance to influence scheduling points, support for it will likely be added at some stage. Also, a file system can be argued to be an "embarrassingly parallel" application, where most operations, depending slightly on the backend, can run completely independently of each other.

[3] "full" as in "starting from the mount point"

3. Results and Experiences

puffs has been imported to the NetBSD source tree. It will be featured in the upcoming NetBSD 4.0 release as an unsupported experimental subsystem. Example file systems are shipped in source form to make it clear no binary compatibility is going to be provided for NetBSD 4.0. Full support is planned for NetBSD 5.0.

3.1. Example File Systems

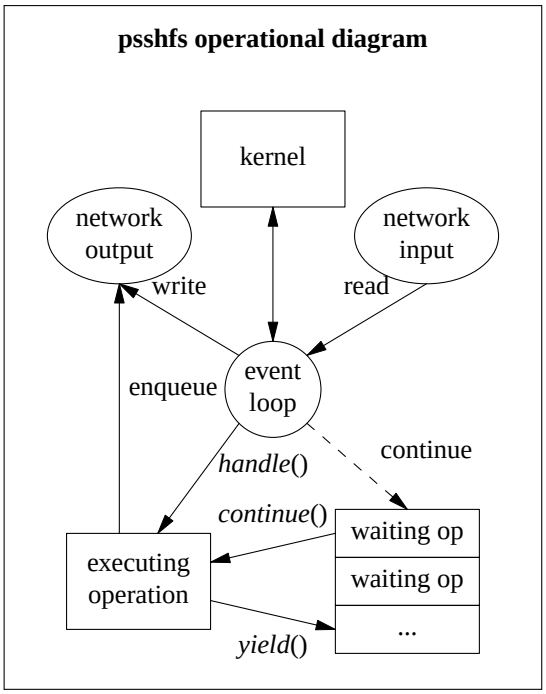
psshfs - puffs sshfs

One desired feature commonly associated with userspace file systems is sshfs. It gives the ability to mount a remote file system through the sftp ssh subprotocol [20]. The most widely known sshfs implementation is FUSE sshfs. It was originally available only for Linux, but is currently available also for FreeBSD and Mac OS X. However, since all the other projects use (L)GPL licensed original FUSE code, with *puffs* NetBSD is only operating system to provide a completely BSD-licensed sshfs solution out-of-the-box.

While psshfs will be supported fully by the eventual release of NetBSD 5.0, NetBSD 4.0 ships with an experimental source-only simple sshfs, ssshfs, found under `share/examples/puffs/ssshfs` in the source tree. The difference between ssshfs and psshfs is that ssshfs was written as simple glue to OpenSSH code and cannot utilize puffs continuations. psshfs was written completely from scratch with multiple outstanding operations in mind.

The operational logic of psshfs is based on an event loop and puffs continuations. The loop is the following:

1. read and process all requests from the kernel. some of these may enqueue outgoing network traffic and *yield*().
2. read input from the network, locate continuations waiting for input, issue *continue*() for them. if a request blocks or finishes, continue from the next protocol unit received from the network. do this until all outstanding network traffic has been processed.
3. send traffic from the outgoing queue until all traffic has been sent or the socket buffer is full.
4. issue responses to the kernel for all operations which were completed during this cycle.



dtfs

dtfs was used for the final development of *puffs* before it was integrated into NetBSD. It is a fully functional file system, meaning that it can do all that e.g. ffs can. The author has run it on at least */tmp*, */usr/bin* and */dev* of his desktop system. For ease of development dtfs uses memory as the storage backend. However, it is possible to extend the file system for permanent storage by using a permanent storage backed memory allocator, such as one built on top of *mmap*() with `MAP_FILE`.

Development of dtfs was straightforward, as it does what the exported kernel virtual file system layer assumes a file system will do and it very closely resembles the operational logic of in-kernel file systems.

puffs nullfs

A nullfs [12] layer, also known in some contexts as a loopback file system [21], is provided by libpuffs. A null or loopback layer maps a directory hierarchy from one location to another. The puffs nullfs is conceptually similar to the in-kernel nullfs in that it acts as a simple passthrough mechanism and always relays unmodified calls the file system below it. However, since it is implemented in the user library instead of the kernel, it cannot simply push the request to the next layer. Instead, it uses

pathnames and system calls to issue requests to the new location.

The null layer in itself is not useful, especially since NetBSD already provides a fully functional alternative in the kernel. However, it can be used to implement various file systems which modify filenames or file content with very little effort for the backend. An example of a user of the null layer is rot13fs, which is less than 200 lines of code and even of those almost half are involved with setting up the file system and parsing command line options. rot13fs translates pathnames and file content to rot13 for any given directory hierarchy in the file system.

sysctlfs

sysctlfs was an experiment in writing a file system which provides the storage backend through other means than a traditional file system block device -like solution. It maps the sysctl namespace as a file system and supports querying (with e.g. *cat*) and changing the values of integer and string type sysctl nodes. Nodes of type "struct" are currently not supported. Traversing the sysctl namespace is possible with standard tools such as *find*(1) or *fts*(3). sysctlfs does not currently support dynamically adding or removing sysctl nodes. While support for the latter would be possible, the former is problematic, since the current file system interface exported to processes in the form of system calls does not provide any obvious way to specify all the information, such as node type, required to create a sysctl node. Non-obvious kludges such as abusing mknod are possible, though.

Development was mostly done during a single day. One of the features introduced to *puffs* because of sysctlfs was the ability to instruct the kernel vfs attachment to bypass cache for all operations. This is useful here because re-querying the information each time from sysctl(3) is not expensive and we want changes in both directions to show up as quickly as possible in the other namespace.

3.2. Experiences

The above clearly demonstrates that adapting a name hierarchy and associated data under the file system interface is possible with relative ease and in a very short time. It can be argued that the development time was cut down greatly due to the author’s intimate familiarity with the system. But it must also be pointed out that some time included in the development time was spent tracking down generic kernel bugs triggered by the corner-case vfs uses of userspace file systems and that some effort was used on framework development. Currently, the development of simple file systems should take only hours or days for someone with a reasonable familiarity in the problem scope.

3.3. Stability

One of the obvious goals is to "bullet-proof" the kernel from mistakes or malice in other protection domains. The author has long since developed file systems purely on his desktop machine instead of inside an emulator or test environment. This has resulted in a few crashes in cases where the userspace file server has been acting erroneously. There are no known cases of *puffs* leading to a system crash when the file system is operating properly and many people in fact already run psshfs on their systems. Incidents where a misbehaving file server manages to crash the system are being fixed as they are discovered and discoveries are further and further apart.

It is, however, still very easy to figure out a way to maliciously crash the system, such as introduce a loop. This is more of a convenience problem than a security problem, though, since mounting a file system still requires special privileges not available to regular users.

Simply using the system long enough and developing new file systems will iron out all fairly easy-to-detect bugs. However, to meet the final goal and accomplish complete certainty over the stability and security of the system, formal methods more developed than cursory analysis and careful C coding principles are required.

4. Performance

These performance measurements are meant to give a rough estimate of the amount of overhead that is caused by *puffs*. Naturally a userspace file system will always be slower than a kernel file system, but the question is if the difference is acceptable. Nevertheless, it is important to keep in mind that the implementation has not yet reached a performance tuning stage and what has been measured is code which was written to work instead of be optimal.

The measurements were done on 2GHz Pentium 4 laptop running NetBSD 4.99.9. Note that the slowness of disk I/O is exacerbated on a laptop.

The first measurement used was extracting a tarball which contains the author's kernel compilation directory hierarchy from memory to the target file system. The extracted size for this is 127MB and contains 2332 files. It will therefore reasonably exercise both the data and name hierarchy sides of a file system.

The files were extracted in two different fashions: a single extract and two extractions running concurrently. For non-random access media the latter will stress disk I/O even more.

Four different setups were measured in two pairs: ffs and ffs through puffs nullfs; dtfs and tmpfs[4]. Technically this grouping gives a rough estimate about the overhead induced by *puffs*. It should be noted that the double test for the dtfs case is not entirely fair, as the machine used for testing only has 512MB of memory. The tree and the associated page cache does not fit into main memory twice. The tmpfs test does not have this problem, as it does not store the tree in memory and in the page cache.

tar extraction test

	tmpfs (s)	dtfs (s)	diff (%)
single	3.203	11.398	256%
double	5.536	22.350	303%
	ffs (s)	ffs+null (s)	diff (%)
single	47.677	53.826	12.9%
double	109.894	113.836	3.6%

Another type of test performed was the reading of a large file. It was done both directly off of ffs and through a *puffs* null layer backed by ffs and it was done both for an uncached file (uc) and a file in the page cache (c). Additionally, the null layer test was done so that the file was in the page cache of the backing ffs mount but not the cache of the null mount itself (bc). This means that the read travelled from the kernel to the user server, was mapped as a system call to ffs, and the data was found from the ffs file system's page cache, so no disk I/O was necessary.

4.1. Analysis of Results

The results for extraction show that *puffs* is clearly slower than an in-kernel file system. This is expected. But what is surprising is how little overhead is added. tmpfs is a high optimized in-kernel memory efficient file system. dtfs is a

[4] tmpfs is NetBSD's modern memory file system

read large file

	system (s)	wall (s)	cpu (%)
ffs (uc)	0.2	11.05	1.8
null (uc)	0.6	11.01	5.9
ffs (c)	0.2	0.21	100.0
null (c)	0.2	0.44	61.6
null (bc)	0.6	1.99	31.7

userspace file system written for testing purposes and not optimized at all. It uses *malloc*() as a storage backend and as a extreme detail it does not do block level allocation; rather it *realloc*()s the entire storage for a file when it grows.

tmpfs contains 4828 lines of code while dtfs is 1157 lines. The difference in code size is over four times as many lines of code for tmpfs. The difference in development effort probably was probably even greater than this, although of course there is no measurable evidence to back it up. Development cycles for fatal errors for a kernel file system are also considerably slower: even though loadable modules can be used to reduce the test cycle time to not require a complete reboot, this will not help if the file system under test crashes the kernel.

Even though tmpfs and dtfs are compared here, it is important to keep in mind that they in no way attempt to compete with each other.

A regular system call for a file operation requires the user-kernel privilege boundary to be crossed twice, while the puffs null scheme requires it to be crossed at least six times: system calls do not map 1:1 to vnode operations, but rather they usually require several vnode operations per system call. However, as the results show, the wall time penalty is very much hidden under the I/O time imposed by the media.

The large file read test mostly measures cache performance. The interaction of *puffs* with the page cache is less efficient than ffs. The reasons will be examined in the future. Also an interesting result is the direct read from disk, which was always slower than the read from disk via nullfs. This result cannot yet be fully explained. One possible explanation is that the utility *cat* used for testing issues *read*() system calls using the file system blocksize as the buffer size and this creates suboptimal interaction with ffs. When reading the file through the null layer the read-ahead code requests 64k (MAXPHYS) chunks and these are converted back to system calls at the null layer and ffs is accessed in 64k

chunks providing better interaction. This is, however, just a hypothesis.

The "backend cached" test (bc) gives yet another idea of overhead introduced by *puffs*. It shows that reading a file in backend cache is ten times as expensive in terms of wall time as reading it directly from an in-kernel file system's cache is. It shows a lot of time was spent waiting instead of keeping the CPU busy. This will be analyzed in-depth later.

5. Current and Future Work

Even though *puffs* is fully functional and included in the NetBSD source tree, work is far from complete. This chapter outlines the current and future work for reaching the ultimate goals of the project.

File System Layering

File system layering or stacking [12,22] is a technique which enables file system features to be stacked on top of each other. All layers in the stack have the ability to modify requests and the results. A common example of such a file system is the union file system [23], which layers the top layer in front of the bottom layer in such a fashion that all modifications are done on the top layer and shadow the file system in the bottom layer.

While rot13fs is a clear example of a layering file system implemented on top of the puffs null layer, libpuffs does not yet support any kind of layering. Making layering support an integral, easy-to-use, non-intrusive part of libpuffs a future goal.

Improving Caching

As mentioned in Chapter 2, kernel caching is already at a fairly good stage, although it could still use minor improvements. However, library support for generalized caching is missing. The goal is to implement caching support on such a level in libpuffs that most file systems could benefit from the caching logic by just supplying information about their backend's modification activity.

This type of library caching is useful for distributed file system where the file system backend can be modified through other routes than the kernel alone. In cases where the file system is accessed only through the local kernel, the file server does not need to take care about caches: the kernel will flush its caches correctly whenever it is required, for example when a file is removed.

Another use is more aggressive read-ahead than what the kernel issues. To give an example, when reading a file in bulk over psshfs, the kernel read-ahead code eventually starts issuing reads in large blocks. However, an aggressive caching subsystem could issue a read-ahead already for the next large block to avoid latency at a block boundary. It could also measure the backend latency and bandwidth figures and optimize its performance based on those.

Messaging Interface Description

Currently the message passing interface between the kernel and libpuffs is described with struct definitions in `puffs_msgif.h`. All request encoding and decoding is handled manually in code both in the kernel and libpuffs. This is both error-prone and requires manual labour in a number of places. First of all, multiple locations must be modified both in the kernel and in the library in case of an interface change. Second, since all semantic information is lost when the messages are written as C structures, it is difficult to facilitate a tool for automatically creating a skeleton file system based on the properties of the file system about to be written.

By representing the message passing interface by a higher level description with, for example XML, much of the code written manually can be autogenerated. Also, this would lend to skeleton file system creation and to building limited userspace file system testers based on the properties of the created file system skeletons.

Abolishing Vnode Locking

Currently the system holds vnode locks while doing a call to the file server. The intent is to release vnode locks and introduce locking to the userspace file system framework. This will open up several opportunities and will enable the file system itself to decide what kind of locking it requires; it knows its own requirements better than the kernel.

Self-Healing and Self-Recovery

In case a file server hangs due to a programming error, processes accessing the file system will hang until the file server either starts responding again or is killed. While the problem can always be solved by killing the file server, it requires the intervention from someone with the correct credentials. Detecting malfunctioning servers and automatically unmounting them

would introduce recovery and self-healing properties into the system. Remounting the file system automatically afterwards would minimize a break in service.

Compatibility

To leverage the huge number of userspace file systems already written and available, it makes sense to be interface compatible with some projects. The most important of these is FUSE, and a source code level compatibility layer to *puffs* for FUSE file systems, dubbed *refuse*, is being developed as a third party effort. As of writing this, the compatibility layer is able run simple FUSE file systems such as hellofs. Progress here has been fast.

Another interesting compatibility project is 9P support. Even though, as stated earlier, supporting it in the kernel would require a huge undertaking, emulating it on top of the *puffs* library interface may prove to be a manageable task. Currently though, the author knows of no such effort.

Longer Term Goals

A large theme is improving the vfs layer by identifying some of its properties through formal techniques [24] and using these to show that the *puffs* kernel side correctly shields the kernel from malicious and/or accidentally misbehaving user file system servers. It also allows for the development of the vfs subsystem into a more flexible and less fragile direction.

6. Conclusions

The Pass-to-Userspace Framework File System (*puffs*), a standard component of the NetBSD operating system, was presented in depth, including the kernel and user level architecture. *puffs* was shown to be capable of supporting multiple different kinds of file systems:

- psshfs - the puffs sshfs file system capable of mounting a remote location through the ssh sftp protocol
- dtfs - an in-memory general-purpose file system
- sysctlfs - a file system mapping the sysctl tree to a file system hierarchy
- nullfs - a file system providing any directory hierarchy in the system in another location

The ease of development of these file systems was observed to be good. Similarly, the development test cycle time and time for error recovery from crashes was observed to be very close to nil. The comparison is the typical times measured in minutes for kernel file systems. Additionally, *puffs* does not require any special tools or setup to develop, as is typical for kernel development. Rather, standard issue user program debuggers such as *gdb* can be attached to the file system and the live file system can be debugged on the same host as it is being developed on.

Performance of file systems built on top of *puffs* was shown to be acceptable. In cases where the storage backend has any significant I/O cost, i.e. practically anything but in-memory file systems, the wall time cost for *puffs* overhead was shown to be shadowed by the I/O cost. As expected, *puffs* was measured to introduce some additional CPU cost.

Finally, since *puffs* is entirely BSD licensed code, it provides a significant advantage to some parties over (L)GPL licensed competitors.

Acknowledgements

The *puffs* project was initially started under Google Summer of Code 2005 mentored by Bill Studenmund. Some later funding was provided by the Ulla Tuominen Foundation.

Professor Heikki Saikkonen helped review and finalize this paper.

References

1. Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos, “Can We Make Operating Systems Reliable and Secure?,” *IEEE Computer*, vol. 39, no. 5, pp. 44-51.
2. Brian N. Bershad, *The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems*, pp. 205-211, Workshop on Micro-Kernels and Other Kernel Architectures (1992).
3. S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, pp. 238-247, Summer Usenix Conference, Atlanta, GA (1986).
4. Assar Westerlund and Johan Danielsson, *Arla---a free AFS client*, pp. 149-152, Usenix Freenix Track (1998).

5. Kristaps Dzonsons, *nnpfs File-systems: an Introduction,* Proceedings of the 5th European BSD Conference (November 2006).

6. Miklos Szeredi, *Filesystem in USErspace, http://fuse.sourceforge.net/* (referenced February 1st 2007).

7. Csaba Henk, *Fuse for FreeBSD, http://fuse4bsd.creo.hu/* (referenced February 1st 2007).

8. Amit Singh, *A FUSE-Compliant File System Implementation Mechanism for Mac OS X, http://code.google.com/p/macfuse/* (referenced February 4th, 2007).

9. Bell Labs, "Plan 9 File Protocol, 9P," *Plan 9 Manual.*

10. Eric Van Hensbergen and Ron Minnich, *Grave Robbers from Outer Space: Using 9P2000 Under Linux,* pp. 83--94, USENIX 2005 Annual Technical Conference, FREENIX Track (2005).

11. *"The Clustering and Userland VFS transport protocol - summary"* (May 2006). DragonFly BSD Kernel mailing list thread title.

12. Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley (1996).

13. Chuck Silvers, *UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD,* pp. 285-290, Usenix Freenix Track (2000).

14. *buffercache(9) -- buffer cache interfaces* (October 2006). NetBSD Kernel Developer's Manual.

15. Marshall Kirk McKusick, Samuel J. Leffler, and Michael J. Karels, "Name Cacheing," *Measuring and Improving the Performance of Berkeley UNIX* (April 1991).

16. David C. Steere, James J. Kistler, and M. Satyanarayanan, *Efficient User-Level File Cache Management on the Sun Vnode Interface,* pp. 325-332, Summer Usenix Conference (1990).

17. Juergen Hannken-Illjes, *fstrans(9) -- file system suspension helper subsystem* (January 2007). NetBSD Kernel Developer's Manual.

18. *puffs -- Pass-to-Userspace Framework File System development interface* (February 2007). NetBSD Library Functions Manual.

19. Edward A. Lee, "The Problem with Threads," UCB/EECS-2006-1, EECS Department, University of California, Berkeley (2006).

20. J. Galbraith, T. Ylönen, and S. Lehtinen, *SSH File Transfer Protocol draft 03,* Internet-Draft (October 16, 2002).

21. Sun Microsystems, "lofs - loopback virtual file system," *SunOS Manual Pages, Chapter 7FS* (April 1996).

22. David S. H. Rosenthal, *Evolving the Vnode Interface,* pp. 107-118, Summer Usenix Conference (1990).

23. J. Pendry and M. McKusick, *Union Mounts in 4.4BSD-Lite,* pp. 25-33, New Orleans Usenix Conference (January 1995).

24. Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi, *Using Model Checking to Find Serious File System Errors,* pp. 273-288, OSDI (2004).