Kernel File Systems as Userspace Programs

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

ABSTRACT

All things should be as a simple as possible. Kernel development is far from simple, and while kernel interfaces and implementation details exhibit a high level of complexity, it does not mean that development should be difficult from all angles. Making it possible to do kernel component development as a regular userspace program enables developers both to use standard tools for the job as well as leverage the generally easier environment.

This paper presents the design and implementation for mounting and running kernel file systems as userspace servers on NetBSD. Existing file systems can be used by means of recompilation - no modification or file system specific shim code is necessary. From an application perspective, the file system is run below the virtual file system layer. The end result is file system code running in userspace without applications being able to tell the difference from the same file system code running in the kernel. Contrasting to a typical kernel development approach where, testing in userspace is done with a specialized application possibly requiring code manipulation, this approach brings dramatic differences. It is possible to use existing applications to test real-world file system call patterns. Of course handcrafted unit tests are also still possible.

The design of the architecture is discussed. The implementation is explained pointing out approaches which resulted in dead ends. Potential extensions of the idea to other operating systems, other kernel subsystems and even other unforeseen possibilities are presented. Finally, experiences with the current implementation are shared and its shortcomings are contemplated while outlining future work.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.3 [**Operating Systems**]: File Systems Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Reliability, Verification

Keywords

kernel development, file systems, cross-platform kernel components

1. INTRODUCTION

"*How hard can it be?*" This is a typical question raised by a novice file system developer. After all, in principle file systems are very simple entities: they merely organize data in a namespace and produce once stored data based on a query key consisting of a file name and offsets. However, in real life, file systems are anything but simple beasts. Their tight integration with the virtual memory subsystem, requirement to be able to recover from crashes, and high level of caching and optimizations make them very complex. And as is well-known, programming in the kernel environment is always a difficult task.

File systems are typically implemented in userspace as a regular program library for initial testing and only later moved to the kernel once the parts dealing with the on-disk structures have been developed and tested. This approach does not facilitate *mounting* the file system during development for running normal applications against it. Rather, specialized test programs are used to make direct calls into file system code. For example, the Fast File System (FFS) [24] was implemented in this way. McKusick felt that the implementation of FFS would have been greatly helped if he had had a testbed with which to develop a file system with kernel interfaces as a userspace server [21]. To take a recent example, Sun's ZFS was also first implemented as a userspace program [5]. The ChunkFS [17] prototype is fully mountable, but it is implemented on top of FUSE [1] and userspace interfaces instead of the kernel file system interface.

This work aims to simplify the task of file system development. It presents the design and implementation of a framework for mounting and running kernel file systems in userspace with complete application transparency. The implementation is done on the NetBSD [3] operating system, which is a free, 4.4BSD derived OS running on over 50 platforms and used in the industry especially in embedded systems. The kernel file system code runs as a userspace file server behind the virtual file system (VFS) layer [20]. For userspace file system integration, the *puffs* [19] userspace file systems framework is used. While the concepts are believed

to be applicable to all Unix-type operating systems, when discussing the implementation this paper refers to NetBSD terminology.

These existing NetBSD kernel file systems have been tested to be mountable in userspace without source modifications: cd9660, EFS, Ext2fs, FFS, HFS, LFS (read-only), MSDOSFS, NTFS, tmpfs, and UDF. The only major real file system not currently supported is NFS, as it requires a lot of additional support from the kernel networking subsystem.

This paper defines the term "Runnable Userspace Meta Program" (*rump*) to mean a kernel component which is run as a userspace program. Specifically, a "*rump file system*" means kernel file system code running in userspace.

Some of the benefits of a userspace kernel file system development framework are:

- **no separate development cycle**: There is no need for a separate userspace development cycle before writing kernel code.
- **real call patterns**: Userspace testing can be trivially done with application-originated call patterns.
- **no bit-rot**: There is no maintenance cost for separate userspace code because it does not exist.
- **short test cycle**: The code-recompile-test cycle time is very short and a crash results in a core dump and inaccessible files, not a kernel panic and total application failures.
- **userspace tools**: Userspace dynamic analysis tools such as Valgrind [26] can be used on kernel code. A debugger can be used as with any other userspace program.
- **complete isolation**: Changing interface behavior for e.g. fault injection [18] purposes can be done without worrying about bringing the whole system down.

The idea is not to to turn NetBSD into a microkernel, but rather to use microkernel ideas and techniques on an existing system to bring real-world impact. The purpose, after all, is to make kernel code easier to develop, test, analyze, and instrument and therefore better. The interfaces used by the developer are still the regular kernel interfaces and existing file systems can be improved or new file systems developed using existing code as a base. [15]

To better integrate the work into the NetBSD operating system, the main kernel was modified where necessary. These modifications were done to make *rump*s easier to *compile*, not to run them. Modifications had to be architected properly so that they could be integrated into the official kernel sources and could not therefore have any adverse effects. File system modification was consciously avoided except for bug fixes. The work described in this paper has already been integrated to the official NetBSD source tree.

This is in contrast to a typical research approach where the developed component is completely detached from the system upon which it was built. A detached approach, while providing a drop-in ability, can mandate non-optimal technical solutions; it is not possible to fix problems in the right places, but rather they have to be worked around in the component being developed. As the work described in this paper has already been integrated, there is no argument for not having modified the main sources.

This paper makes the following contributions. It shows that it is possible and desirable to run existing kernel file systems as userspace file servers on a real-world BSD-derived operating system. It then points out that the resulting userspace servers are independent of the operating system and encourages other systems to implement similar features to promote wider inter-OS component re-use.

The remainder of the paper is organized as follows. Chapter 2 discusses related work. Chapter 3 presents the architecture and component division. Chapter 4 goes into details on the implementation. Chapter 5 presents experiences with the implementation, including rough performance measurements. Chapter 6 goes over future work and envisions uses beyond file systems. Chapter 7 concludes.

2. RELATED WORK

The Alpine [11] network protocol development infrastructure shares many of the same goals. It provides a networking stack development environment for running kernel code in userspace. However, it is implemented before the system call layer by overriding certain libc symbols and is run in the same process context as the application. This approach both makes it unsuitable for statically linked programs and creates difficulties with shared global resources such as the `read()`/`write` calls used for I/O beyond network one. Furthermore, from a file system perspective, this kind of approach shuts out kernel-initiated file system access, e.g. NFS servers.

Rialto [9] is an operating system with a unified interface both for userspace and the kernel making it possible to run most code in either environment. However, this system was designed from ground-up that way. Interesting ideas include the definition of both internal and external linkage for an interface. While the ideas are inspiring, *rump*s are not about rewriting the entire operating system.

Mach is capable of running Unix as a user process [14]. Lites [16] is a Mach server which can run a wide variety of binaries from different Unix flavors at the same time. It is is based on the 4.4BSD Lite code base. Debugging and developing 4.4BSD file systems under Mach/Lites is possible by using two Lites servers: one for running the debugger and one running the file system being developed, including applications using the file system. The debugging tool must be able to talk from server to another. For example, gdb uses Mach system calls. An additional difference to this work is that if the Lites server being debugged crashes, applications inside it will be terminated.

Operating systems running in userspace, such as User Mode Linux [8], make it possible to run the entire operating system as a userspace process. The main aims in this are providing better debugging & development support and isolation between instances. However, this approach does not provide

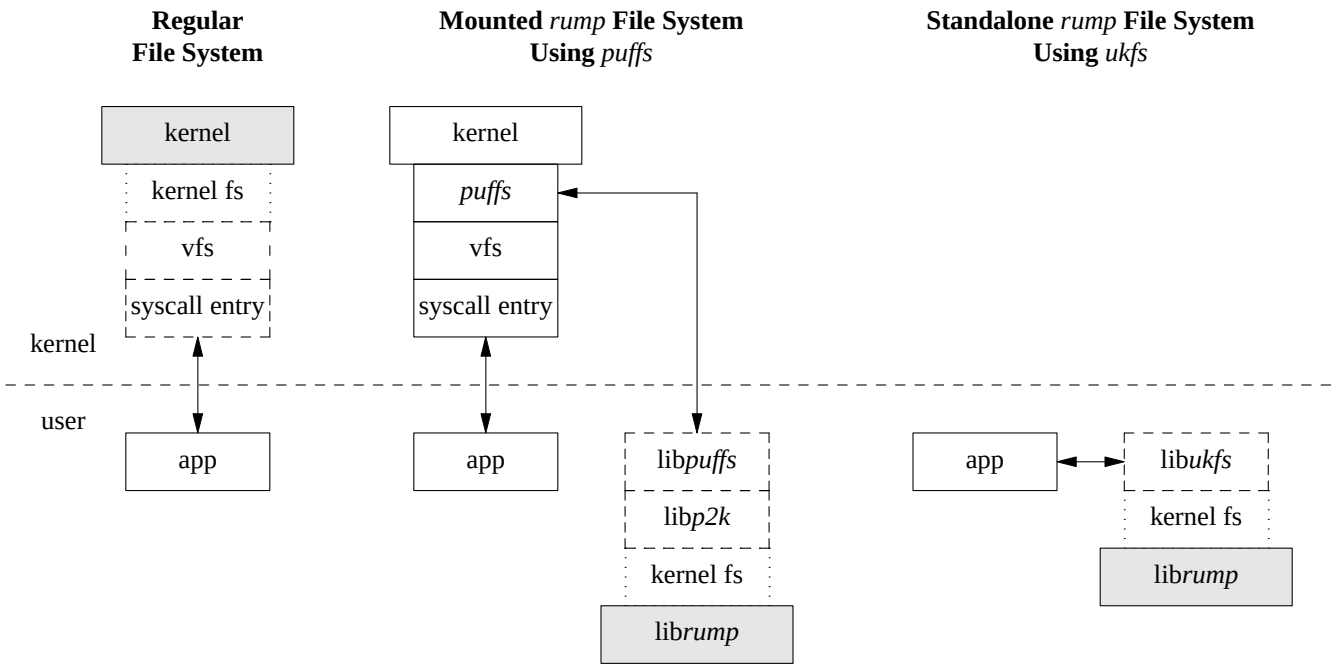


Figure 1: *rump* File System Architecture

isolation between the component under development and the core of the operating system - rather, they both run in the same process. This results in complexity in, for example, using fault injection and dynamic analysis tools. Additionally, this approach does not permit writing userspace test programs which directly call various interfaces under test. In a sense this is very similar to the Lites case.

Sun's ZFS [4] file system ships with a userspace testing library, libzpool. In addition to some kernel interface emulation routines, it consists of the Data Management Unit and Storage Pool Allocator components of ZFS compiled from the kernel sources. The ztest userspace program plugs directly into these interfaces for running tests. This approach has several shortcomings when compared to *rump*s. First, it does not test the entire file system code architecture, for example the VFS interface layer. The effort for getting the VFS interface (in ZFS terms known as the *ZFS POSIX Layer* or *ZPL*) right was specifically listed as the hardest part in porting ZFS to FreeBSD [7]. Second, it does not facilitate userspace testing with real applications. And third, the test program is specific to ZFS. While this does specific micro tests where needed, it destroys generality. *rump*s also allow for micro tests, but attempt to keep them to a minimum.

3. ARCHITECTURE

Before going on to the details of the architecture, it is important to note the difference between the namespaces defined by the C headers for kernel and for user code. Any given module must be *compiled* in either the kernel or user namespace. However, after compilation the modules may be *linked* together (assuming that the application binary interface provided by both namespaces is the same).

Code cannot use both namespaces simultaneously due to collisions. For example, `printf`() has a minor variation between both namespaces and will cause compilation to fail. This is historic payload we must deal with when operating on a real-world system with Unix heritage.

To solve the problem, we identify components which require the kernel namespace and components which require the user namespace and compile them as separate compilation units. We let the linker handle unifying them. Additionally, we provide a small set of "bridge" symbols which can be used from one namespace to call a symbol in the other. In a way they are analogous to system calls, except that they do not trap or switch execution mode.

For example, when a read is normally issued to retrieve file contents, a program in user namespace calls the `read`() system call. The system call traps into the kernel and the kernel (in kernel namespace, naturally) executes a `VOP_READ`() vnode operation to access the file system. However, in our case, we need to effectively call `VOP_READ`() from a userspace program. This symbol is not exposed in user namespace (nor can it easily be due to reasons given in Chapter 4.11). The problem is solved by adding a prefix to symbols and prototypes from the kernel counterparts and exposing the prefixed name to user namespace. In the above example, the `RUMP_VOP_READ`() bridge symbol is called. It does does exactly the same as `VOP_READ`(), but has a name suitable for exporting to userspace.

Overview

The architecture of the framework is presented in Figure 1 using three different cases to illuminate the situation. Analogous parts between the three are signaled. The differences are briefly discussed below before moving to further dissect

the components and architecture.

Regular File System

"Regular File System" shows the architecture relevant to running a file system in the kernel. It is presented for comparison.

Mounted rump File System Using puffs

From the application perspective, a mounted *rump file system* looks and behaves like the same file system code running in the kernel. *puffs* [19] (short overview in Chapter 3.3) is used to attach the file system to the kernel virtual file system.

Standalone rump File System Using ukfs

A standalone *rump file system* represents are more typical kernel test harness in that it uses a special interface for making direct calls to the file system code in userspace. Common tasks such as mounting and creating files can be done through the file system independent user-kernel file system library, *ukfs* (Chapter 3.6). This scheme allows also to build test programs which make direct calls into the file system past the virtual file system abstraction.

3.1 The File System Code Itself

Kernel file systems are obviously kernel code. The compilation process itself, though, is like building a normal userspace library. The end result is a library of kernel file system code ready to be linked into a binary.

3.2 librump

The linkage to a file system is handled from the above by the virtual file system layer. In a monolithic kernel, the bottom end consists of any kernel service the file system wishes to call. To facilitate running *rump file system*s, we must emulate the bottom end environment since it is not present. This is where librump comes into play. It emulates enough of the kernel environment for the file system code to be able to run.

3.2.1 Internal Division

librump is conceptually kernel code: it emulates the kernel proper for the file system code. However, to successfully emulate the kernel in userspace, it must be able to make some calls to libc such as system calls. The parts which make the libc calls cannot be compiled in the kernel namespace for reasons discussed in the beginning of this chapter.

We split librump into two portions: *rumpkern* and *rumpuser*. The kernel portion contains routines implementing the bottom end kernel interfaces while the user portion provides interfaces in user namespace. *rumpuser* is used only by the kernel part of librump and a means to defeat the problem mentioned in the previous paragraph. Figure 2 lists some example routines provided by the user portion. It is easy to see their similarity to system calls. The *error* parameter present in the routines signals the *errno* value, if any, for the call. In a addition to system calls, *rumpuser* provides a way to call libc's `malloc`().

To form the *rump* library, the user and kernel parts, which were compiled in separate namespaces, are archived together.

Figure 2: Examples of *rumpuser* interfaces

```
int rumpuser_open(const char *filename, int flags,
      int *error);

int rumpuser_gettimeofday(struct timeval *tv,
      int *error);

ssize_t rumpuser_pread(int fd, void *buf,
          size_t bufsize, off_t offset,
          int *error);
```

Table 1: *rump* library size analysis

Component	# of lines
rumpuser	101
rumpkern	1850
std kern	5040
FFS	14442

3.2.2 Development Approach

As maintainability is an issue, we want to write as little code for the emulation library as possible - code which does not exist cannot get out-of-sync with the kernel. This is done by identifying modules from the kernel sources that can be compiled into librump without modification.

We took advantage of our ability to modify the kernel sources to avoid unnecessary difficulties in direct inclusion. Some parts had great potential, but could not be included unmodified. They contained elements which depended on too many other elements from elsewhere in the kernel and their dependencies would have ended up pulling in large chunks of the kernel.

An example is the file `vfs_subr.c` which contains virtual file system subroutines. Some of these routines could be used without modification in userspace. However, routines dealing with the vnode life cycle were not directly usable. The file was split into two parts, `vfs_subr.c` and `vfs_subr2.c`, the former containing parts tightly tied to the kernel and the latter containing parts suitable for use in librump. It should be noted that splitting the file provides a clear separation and good readability as opposed to using `#ifdef`'s.

The decision of what to include unmodified, what to modify and what to implement from scratch is left to the implementor. However, the trend during development has been to increase the amount of code compiled directly from the kernel sources. It is likely that the amount of specifically-written code will diminish even more over time once solutions to the remaining problems become clearer and unified paths suitable both for *rump*s and the kernel are revealed.

See Table 1 for a rough idea on how much code had to be written ("rumpuser" and "rumpkern") and how much could be compiled directly from the kernel sources. The count is measured without comments or empty lines. The size of the Fast File System is included for comparison to put us on the map about the code size of a kernel file system.

3.3 puffs

puffs [19], or Pass-to-Userspace Framework File System, is the NetBSD framework for building userspace file servers. File systems such as sshfs and 9P have been implemented on top of it. In addition a compatibility layer to Linux's FUSE [1] is provided.

puffs is made up of two components: the kernel virtual file system attachment and a userlevel library, libpuffs. The kernel *puffs* virtual file system attachment handles requests from the kernel. When a request is received, it is transported to userspace, where it is processed by libpuffs and handed to the file server implementation. After the userspace file server has processed the request, the result is returned back to the kernel.

The interface for implementing a userspace file system is defined by libpuffs. This interface is heavily influenced by the kernel VFS interface. However, as the libpuffs interface is designed for userspace file systems, it does not use the same arguments as the kernel VFS interface which uses kernel private structures. Therefore, the interface provided by libpuffs is not directly applicable to kernel file systems.

Files are identified by cookie values. Every file server operation which creates a new in-memory node structure returns a cookie to the kernel. The kernel stores the cookie value in the vnode it creates and uses it to refer to that particular file in future operations. Once a userspace file server has issued a cookie, it must keep the data structure behind it alive until the kernel tells it can be released.

3.4 libp2k

The *p2k* library is a request translator between the libpuffs interface and the kernel virtual file system interface. It registers itself a file system to the libpuffs. As *p2k* uses userspace services, the most prominent case being the *puffs* library, it is compiled in the user namespace and can call kernel routines only through our bridge namespace.

To give an example of the interface differences mentioned in Chapter 3.3, we discuss reading a file. The kernel `VOP_READ`() interface takes a `struct uio` pointer, which defines where data should be read from and where it should be copied to. This includes an address space pointer which indicates what address space (kernel or any given process) contains the data pointed to by the uio I/O vectors. Because userspace can only copy to/from its own address space and because the uio structure is not exposed to userspace, it is not used in the *puffs* interface. Instead, the interface passes relevant parameters individually.

The above is further illustrated by Figure 3, which includes the full *p2k* read routine. The cookie, `opc`, received from *puffs* is passed to the vnode operation. We see the uio structure created by `rump_uio_setup`() before calling the vnode operation and freed after the call while saving the results. We also notice the *puffs* credit type being converted to the *kauth_cred_t* type (further discussed in Chapter 4.7) used in the kernel. This is done by the *p2k* library's own `cred_create`() routine. The `VLS`() and `VUL`() macros deal with the VFS locking protocol.

Figure 3: `p2k_node_read`() Implementation

```
int
p2k_node_read(struct puffs_cc *pcc, void *opc,
        uint8_t *buf, off_t offset, size_t *resid,
        const struct puffs_cred *pcr, int ioflag)
{
        kauth_cred_t cred;
        struct uio *uio;
        int rv;

        cred = cred_create(pcr);
        uio = rump_uio_setup(buf, *resid,
            offset, RUMPUIO_READ);
        VLS(opc);
        rv = RUMP_VOP_READ(opc, uio, ioflag, cred);
        VUL(opc);
        *resid = rump_uio_free(uio);
        cred_destroy(cred);

        return rv;
}
```

We also immediately notice that the library is using kernel types even though earlier it was stated that it is code compiled in the user namespace. This is possible because librump headers provide forward declarations for all necessary kernel data types. We can therefore have opaque pointers to those types in userspace code using librump.

3.5 mount

All kernel file systems depend on a userspace counterpart, *mount_fs*. This program parses command line arguments, creates a file system specific argument structure and calls the `mount`() system call to mount the file system.

Similarly we must have a comparable component if we wish to mount a *rump file system*. Only we do not call the mount system call, we instead do a function call to the kernel virtual file system routine `VFS_MOUNT`().

When the mount program is linked together with libpuffs (user), libp2k (user) and librump (kern + user) and the file system library (kern), we have an executable capable of mounting the file system and servicing requests from the kernel VFS layer.

3.6 libukfs

The *ukfs*, or user-kernel file system, approach is to provide a programming interface which completely skips the system call approach and provides direct access to file systems in userspace. While this implies that normal applications cannot use the file system directly, it means that applications specifically written for the purpose can interact with the file system without kernel involvement. This opens up possibilities beyond testing, one of which is discussed later in Chapter 6.1.

Some example operations provided by the *ukfs* interface are presented in Figure 4. The interface is a high-level interface

Figure 4: Examples of *ukfs* interfaces

```
struct ukfs *ukfs_mount(const char *vfstype,
               const char *devpath,
               const char *mntpath, int mntflags,
               void *arg, size_t arglen);

int ukfs_ll_namei(struct vnode *rvp,
      const char **pnp, u_long op,
      struct vnode **dvpp, struct vnode **vpp);

int ukfs_getdents(struct ukfs *u, const char *dir,
      off_t off, uint8_t *buf, size_t bufsize);

int ukfs_rmdir(struct ukfs *u, const char *dir);

ssize_t ukfs_read(struct ukfs *u, const char *file,
          off_t off, uint8_t *buf, size_t bufsize);
```

and the similarity to the system call interface is apparent. It should be noted, though, that this is provided only for convenience. Should a specific need, e.g. fine-grained testing, require calling a more low-level routine directly, it is certainly permissible to do so.

A notable exception in Figure 4 to the system call interface is the routine `ukfs_ll_namei`(). The `namei`() routine implements the pathname-to-vnode translation. When wanting to execute operations on a vnode represented by a pathname, we must first locate the vnode. For the benefit of applications wishing to call vnode operations directly, this lowlevel routine is exposed.

4. IMPLEMENTATION

This chapter describes the implementation on NetBSD. In select places it goes into great detail and debates different solutions. This is meant to provoke ideas in a reader wanting to implement a similar system.

4.1 Initial Work

The initial development goal of *rump* was to run the Fast File System [24]. The first task was to make FFS compile in userspace. This was a very trivial task and merely included creating a Makefile and setting the header directories right.

The next task of being able to link the resulting library into a runnable binary resulted in a much more labour intensive problem with 231 unsatisfied symbols[1]. These symbols represent all of the kernel's services the FFS code uses.

To make good progress initially, some of these were addressed by adding `#if 0` into the file system code. It was obvious that modifying code is not in line with the goal of not modifying code and had to be redone properly. However, initial shortcuts help to show how to later do the implementation properly and are the recommended approach.

[1]McKusick reported that trying to compile the original Bell Labs' File System in userspace when starting work on FFS generated about 50 unresolved symbols [21]. File systems have become much more complex over the last 25 years.

Unresolved symbols which are not relevant in userspace, such as ones manipulating the system interrupt priority level, were initially resolved using preprocessor macros. Later, using the preprocessor was discovered to be undesirable, as there is no globally included file in which to place the defines, and using the `cc` command line does not scale. All macro definitions were eventually converted to function stubs.

While a tool for automatically creating stubs sounds like an attractive improvement, there are multiple benefits from manually going over the symbols. First, it forces the developer to think about the interfaces being added. Second, a human does a good job of grouping the functions resulting in a maintainable codebase. Third, a human can decide if some of the symbols should be compiled directly from kernel sources.

4.2 Synchronization and Sleeping

The NetBSD kernel has multiple ways of synchronizing resources. These can roughly be divided into methods involving the system interrupt priority level, process context sleeps, and locks.

Currently, *rump file system*s run all operations from start to finish in a single thread and as there are no interrupts in userspace, locking and synchronization is not required. While not mimicking kernel behavior closely, this is enough for fully functioning file systems.

Even though not implementing locking, some routines do track lock status. The virtual file system layer has a well-defined locking protocol which file systems must adhere to. libp2k contains runtime assertions to check that file systems conform to this locking protocol.

4.3 Virtual Memory Subsystem

As stated already in the introductory chapter, file systems are heavily involved with the virtual memory subsystem. We implement parts of NetBSD's UVM [6] virtual memory subsystem relevant to file systems.

The virtual memory subsystem provides lowlevel memory allocation and mapping routines. As we do not need to care about anything starting from page table management and ending with page residence tracking in userspace, we can allocate memory simply by calling `malloc`().

UVM uses pagers for moving data in and out memory. We are mostly interested in vnode pagers, although the *tmpfs* [29] file system uses anonymous memory pagers. We implement both types. Especially the anonymous memory pager implementation is much simpler than in the kernel and is mostly just `malloc`(). This is because we do not need to implement moving pages in and out of swap: our vm uses regular userspace memory and the kernel will handle the paging and swapping of our data when it chooses to. The vnode pager implementation, however, must be able to move data in and out of files.

We also implement support routines required for various purposes by file systems. Additionally, while not strictly vm functionality, `copyin`() and `copyout`() are implemented.

As *puffs* guarantees all data to always be in the same address space, these are implemented simply as `memcpy`().

4.4 Virtual File System

The kernel virtual file system layer, besides providing the interface individual file systems hook to [20], also provides vast amounts of subroutines used by file systems. This section describes the implementation and rationale of various parts which were reimplemented. The parts of the virtual file system functionality which are compiled as-in from the kernel source tree are: name caching, vnode operation interface, various subroutines and initialization code.

4.4.1 vnode Management

The kernel vnode management is characterized by two things: all file systems share a common vnode pool and data structures are allowed to keep weak references to vnodes for caching purposes. The lifetime of a vnode is controlled by its reference count and the least recently used vnode with a reference count of zero is recycled when a new one is needed.

Recall, in userspace the node life cycle is dictated by the kernel and the *puffs* interface. We use the address of the vnode structure created by the *rump file system* as the *puffs* cookie. As we cannot destroy the node before receiving the command from the *puffs* interface, it does not make sense to reference count vnodes in our userspace implementation. We simply destroy the vnode when *puffs* asks us to. Pooling does not make sense, as we run one file system per user address space and cannot share structures between them.

One exception to vnode release is when the file system itself wishes to recycle an existing vnode: `VOP_INACTIVE`() is called when a vnode's reference count drops to zero in the kernel. This is forwarded to our *rump file system* by *puffs*. If the file is no longer accessible through the file system namespace, i.e. all links have been deleted, the file system may call `vrecycle`() in `VOP_INACTIVE`() to immediately garbage collect the node. This is how Unix file removal semantics work - a removed file is really removed only when all active references to it are gone.

We implement `vrecycle`() to reclaim the vnode and signal in internal bookkeeping that this has been done by setting the vnode's private data pointer to an invalid value. We do not, however, free the node yet. When returning to the kernel from `VOP_INACTIVE`(), *p2k* checks if a node has been recycled and hints *puffs* (and therefore the kernel) it can reclaim the node. Only when the kernel reclaim operation is issued through *puffs*, do we free the node.

4.4.2 namei

As mentioned already earlier in Chapter 3.6, `namei`() is responsible for translating a pathname to a vnode. Most file systems call it to translate their storage device path to a vnode. In the kernel namei is implemented by a series of `VOP_LOOKUP`() calls to the pathname components.

We cannot do this, as we do not have a file system to call lookup on; mounting a *rump file system* can be thought of being like mounting the root file system. Instead, we note that a file system is really only interested in the vnode type. We execute the `stat`() system call on the path and decide what type of vnode to create, if any, based on the returned data.

4.4.3 Buffer Cache

The file system buffer cache [22] is used for caching file system metadata in modern BSD operating systems. It operates on a block basis. The file system can request a block from the buffer cache. If the block is in memory and unbusy, it is given to the caller. Otherwise, the caller is put to sleep and awaken either when another user releases the block or the block has been read from disk. If the block is modified, it can be written synchronously, asynchronously, or just marked dirty to be flushed eventually.

The first implementation of the buffer cache for librump was a very simple implementation which did not actually cache anything. It always read the block contents from the device when a block was requested and always wrote the contents of a dirty block to the device immediately when the block was released. This simple implementation was only 201 lines long as opposed to the kernel buffer cache's 1861 lines. However, it lacked the handling of all the kernel buffer cache flags and semantics.

The simple implementation was done first because large parts of the kernel and vnode handling were interlinked with the buffer cache. Later, due to pulling in other kernel code it became possible to compile in the real kernel buffer cache. What we were after was gaining the full buffer cache interface semantics, but naturally this also boosted performance.

4.5 Reading and Writing Files

4.5.1 Reading and Writing Inside the Kernel

Modern operating systems depend heavily on the virtual memory subsystem for caching file contents and NetBSD is no exception [28]. To understand how we can support I/O in userspace, we must first recall how I/O is performed in the kernel. Reading is used as the example; writing is similar, but differs in specifics.

The read operation receives two arguments: the vnode to read from and a `struct uio` pointer describing how much to read and from which offset in the file. To perform the read, the kernel maps a memory window for the desired offsets into the kernel virtual memory space using `ubc_alloc`(). After this, `uiomove`() is performed on the memory window to read the contents. If the data desired are already present in the page cache, they are simply transferred to the memory addresses designated by the uio control structure. However, the interesting case for us is when the data are not present.

If the memory window does not have a backing page mapped, a page fault is generated. The kernel page fault handler resolves the fault to a vnode pager[2], which in turn handles the fault by calling the file system's `VOP_GETPAGES`() method, which retrieves the appropriate data. The details of getpages, especially with respect to vm interaction, are complex, but almost all file systems are able to directly use

[2]As mentioned earlier, *tmpfs* uses anonymous memory pagers and differs slightly from the discussion.

a generic routine called genfs_getpages(). Ones which cannot, typically leverage it through a wrapper. genfs_getpages() in turn interacts with the actual file system storage through VOP_BMAP() and VOP_STRATEGY() to fetch the file contents.

4.5.2 Emulating Page Faults

As we do not have a page fault mechanism available in userspace, we cannot plug a fault handler to it. We could emulate page faults in the typical userspace fashion by returning a PROT_NONE-mapped region from ubc_alloc() and by trapping the segmentation violation signal. But as operating in signal handlers is always difficult to do correctly and brings no true benefit, the option was left uninvestigated.

Instead, we base operation on the simple premise that we have no particular desire to follow the kernel closely in this regard. Faulting is a method for implementing file caching in the page cache - we do not wish to implement the page cache is userspace, nor could we do it this way due to the lack of a proper virtual memory subsystem. Additionally, we observe that all manipulation of the memory window in all file systems is done using the uiomove() routine. To exploit this fact, we return a special window address from ubc_alloc() and teach the librump uiomove() implementation about special window addresses. In case uiomove() encounters a special address, it performs file system operations instead of regular memory operations.

While this solution can be argued to be a hack, it is very simple and very efficient. However, it will fall down if someone implements a file system which does direct access on the memory window by using methods such as memcpy(). No such file systems exist to the author's knowledge. Notably though, some file systems contain an error branch to set the mapped window contents to zero using memset() if a file is enlarged but writing data to the enlarged portion fails. We must catch failures in our uiomove() magic handler.

4.5.3 Servicing Requests

Initially, the servicing routine called from our magic address detection in uiomove() did nothing more than transfer data to and from the file system using VOP_BMAP() and VOP_STRATEGY(). This was because it was regarded unnecessary to implement a full virtual memory object pager interface considering that we were not interfacing with the virtual memory subsystem. The genfs_getpages() routine mentioned earlier in Chapter 4.5.1 along with its write side counterpart genfs_putpages() were implemented simply as do-nothing stubs. However, there were multiple problems with this strategy.

The FFS file system uses so-called fragments to avoid allocating a full file system block as the last block in a small file[24]. When such a file is extended so that the data does not fit into fragments anymore, a full file system block is allocated. Data is moved from the fragment to the full block and the fragment is released. Essentially what happens when the FFS file system code is requested to write beyond its EOF is that it faults in the blocks from the end of the file using VOP_GETPAGES(), extends the file and writes the data.

Since we defined genfs_getpages(), which is used by FFS VOP_GETPAGES(), to be a stub operation, the faulting routine did not work. It should also be stressed that it is not enough to read/modify/write when writing the new data, since the last block has already been remapped earlier by the file system and doing so will result in garbage in the sections that are remapped from the fragment.

This was initially solved by wrapping the call to VOP_WRITE() in *p2k* to read the file into memory for the block-rounded range we were going to write to. The other part was teaching our magic uiomove() routine to write full file system blocks instead of just the offsets it was asked to write. This, while being a solution clearly geared toward file systems which need fragment remapping, worked.

However, this scheme assumed that the file system vnode's vm object pager was a vnode object - it is not possible to perform VOP_STRATEGY() on other types of objects. When testing tmpfs as a *rump file system*, it was noticed that tmpfs uses anonymous memory pagers and does not fit the assumption of a vnode pager.

To support file systems with any kind of pager, the abstraction in our magic uiomove() routine had to be brought up to the level of the virtual memory pager interface and pagers for both types of objects had to be implemented in librump - their in-kernel implementations depend too heavily on the kernel vm. Implementing the vnode pager also involved implementing the *genfs* get and put routines.

As an end result, the implementation is very much like the kernel counterpart, the exception being fault handling. Virtual memory subsystem interaction is also much simpler due to our userspace vm subsystem being simpler.

4.6 The Device Interface

All file systems which use mass media as a storage backend need to operate on a device to read and write data. In practice this is handled in the kernel by the file systems knowing the vnode of their device special file, and invoking VOP_STRATEGY() on the device vnode. In the kernel, device special files use a *specfs* vnode operations vector. The strategy call is handled by spec_strategy(), which is turn calls bdev_strategy() to transfer the data to/from the physical device.

4.6.1 *rump* specfs

All device special nodes created by the librump namei() routine have their vnode operations vector set to *rumpspec_vnodeops*. This operations vector implements necessary support for userspace routines.

The kernel accesses the device driver directly. As this is not possible from userspace, we need to make system calls to perform the same activities. The following is an overview of the routines implemented in *rumpspec_vnodeops*:

- **open**: call the open() syscall to open a file descriptor for the device file
- **close**: close previously opened device file descriptor
- **ioctl**: call ioctl() on the device file descriptor

- **strategy**: `read`() or `write`() the open device file descriptor. This makes transactions with the storage backend and is used both for data and metadata.

4.6.2 Block Device Emulation

The kernel requires that the node the file system is mounted from is a block device special node. Technically, there is no reason why the kernel could not mount file systems off of regular files, but that would require restructuring in various areas. This means that for the kernel to be able to mount regular files, a driver called *vnd* (vnode disk driver) must be used. It makes a regular file to look like a block device so that the kernel can use it as a file system device.

*rump file system*s have no such limitations. All our device operations are executed on a file descriptor and a `read`() is a read regardless of backing type. Therefore, we can allow file systems to be mounted off of regular files. We just teach our `namei`() to fake regular files as of type block special file, otherwise the file system code rejects the attempted mount.

Not having to use a *vnd* to mount a file system has the added benefit of not requiring root privileges to configure the *vnd*: mounting depends only on access to the media and permissions to execute the mount operation, regardless of if the media is a special device or a regular file.

4.7 Authentication and Credentials

NetBSD uses an abstract authorization framework, kauth [10], where credentials are represented by an opaque cookie of type *kauth_cred_t* and interpreted entirely within the policy implementation. As the framework takes extreme care to not leak the implementation out of the policies, implementing our own specific version of the interface is a straightforward task.

File systems require only a very limited subset of the kauth interfaces. Mostly they are interested in the user id and group id's of the kauth cookie to determine if a certain set of credentials has the authority to perform a particular operation on a file. As an additional example, FFS is interested in if the credentials belong to the equivalent of a super user ("root") when the file system's free space has dropped below reserved space [24].

librump has its own implementation for the kauth routines. A *kauth_cred_t* is created by the routine `rump_cred_create`(), which takes as its argument the uid and primary and supplementary gids. The return value is *kauth_cred_t* eligible to be passed to the kauth routines.

4.8 Machine Headers

NetBSD is a multi-architecture OS. Some structure definitions required for running the kernel are machine dependent. An example is definition of the light weight process currently running on the CPU. This definition can be overridden by the CPU architecture header so that machine dependent code can optimize the handling of the information. Figure 5 contains an illustration of this.

To avoid implementing symbols for all the possible machine values of *curlwp* and other similar entities, librump provides its own subset of machine dependent headers. These define symbols like *curlwp* to values uniform across all architectures. Since *rump*s do not run in the kernel, they do not need to, nor can they, be concerned with machine dependent values.

Figure 5: Definition of `curlwp`

Machine Independent:

```
#if !defined(curlwp)
#if defined(MULTIPROCESSOR)
#define curlwp          curcpu()->ci_curlwp
#else
extern struct lwp       *curlwp;
#endif /* MULTIPROCESSOR */
#endif /* ! curlwp */
```

Machine Dependent (mips):

```
#define curlwp          mips_curlwp
```

4.9 Symbol Collisions

Some symbols share the same name between the kernel and userspace but have different linkage. This means that while compilation and linking go fine, there will incorrect operation at runtime resulting in various degrees of problems.

A very good example of such a symbol is `malloc`() function call. The ISO C signature is `malloc(size)`, while the BSD kernel uses the signature `malloc(size, type, flags)`. *type* describes the subsystem the memory is allocated and *flags* gives arguments as how to perform allocation. A critical flag is `M_ZERO`, which tells the memory allocator to hand out zeroed memory. The equivalent userspace convention is calling `memset`() after `malloc`().

Initially, the kernel calls to `malloc`() were accidentally directly linked with the libc version resulting in the *flags* argument not being handled properly and hence memory not being zeroed. Very surprisingly, this bug was noticed only after a very long time when checking that the optional UFS directory hashing code worked.

This problem, and similar ones, can be solved on the GNU toolchain by using the *–wrap* linker argument. This causes references to `symname` to be directed to `__wrap_symname`, while a reference to `__real_symname` will cause the actual symbol to be referenced. In our case, we implement the wrapper `__wrap_malloc`() and call `__real_malloc`() from there. If the wrapper detects it is called with the `M_ZERO` flag, it will zero out the memory.

To detect symbols which might collide, standard tools can be used. By running the `nm` symbol listing tool on both a kernel image and libc duplicates can be easily identified. They must then be analyzed to separate harmful collisions (`malloc`()) from harmless ones.

Currently NetBSD has 244 symbols with the same names between the kernel and userland. The majority of these are routines which are compiled from shared source between the kernel and libc, such as crypto routines. Others, such as

`strcpy`() and `printf`()[3], are harmless. Finally there is the malloc family, where the difference causes problems.

4.10 Special Cases

4.10.1 FFS Soft Updates

Soft updates [23] is a technique for eliminating a good number of synchronous metadata writes in file systems while keeping the on-disk result consistent. It operates by building dependencies of metadata updates for file system operations and flushing the blocks out in the correct order. If a block is flushed out-of-order, it is rolled back to in a consistent state with respect to on-disk metadata, written, rolled forward and marked dirty again.

The implementation of soft updates in BSD is peculiar in the sense that it depends heavily upon code called from the interrupt handler. The above mentioned roll-back and roll-forward operations are hooked into the buffer cache. When a block is being flushed out, it is checked for dependencies and necessary operations are performed. When the write completes, the soft update code is called from the interrupt handler to process any dependencies that need a roll-forward. As *rump file system*s have no interrupt context comparable to the kernel's, this does not present problems: the interrupt handler is simply relegated to a regular function call from our mock device driver's strategy routine.

The only encountered problem was with the handling of asynchronous writes when flushing all the dirty buffers belonging to a node. Since all of our asynchronous writes are immediately finished from the caller's perspective, a block with dependencies is immediately placed back on the dirty list. Due to implementation details, it can be placed only on the head of the list. And due to system interrupt priority level details, the queue must be always traversed from the beginning after issuing a write.

The effect is a livelock where the same block is picked up from the queue, written, put back on and picked up. Notably, this problem theoretically exists also in the kernel, but since a mass media write is orders of magnitude slower than queue traversal, buffers will be on the device queue waiting to be written instead of the dirty block queue. When the interrupts for successful disk writes finally arrive, the queue will already have been traversed completely.

As the problem exists in code shared between the kernel and *rump*s, the solution must be acceptable for inclusion in the kernel. The problem is easy to fix by just putting rolled-back blocks at the end of the dirty list instead of the head. However, doing this is numerous in implementation details and pervades a lot of code. The problem is also easy to work around specifically in *rump* by ignoring system priority level constraints and traversing the entire list from start to finish. As the author does not believe quick `#ifdef`'s to be a proper means of software development, soft dependencies are unavailable in the tree as of writing this.

[3]The linkage of `printf`() does differ, but only in that the kernel one is on type `void` instead of the ISO C specified `int`. As the kernel callers never check return values, returning one does not hurt.

4.10.2 Log-Structured File System (LFS)

The LFS implementation on NetBSD, derived from the 4.4BSD implementation [27], is unlike a regular file system in the sense that part of the file system code runs in userspace. The userspace part is the cleaner process, which reclaims unused parts of the disk log to free up space. According to the LFS paper, one reason for the userspace implementation was being able to experiment with different cleaning algorithms while the two others were for the kernel to be able to separate mechanism from policy and to be able to select a cleaning algorithm based on the workload.

In a few places LFS operates directly on the vnode freelists instead of calling `vrele`(), because it wants to avoid a call to `VOP_INACTIVE`() - calling `VOP_INACTIVE`() requires that `vrele`() locks the vnode. This is a good example of a questionable code architecture which *rump*s make stand out. Although the real problem with this can be argued to exist deeper, the immediate problem was solved by introducing a proper interface into the kernel to do what LFS wants and making LFS use it.

Currently it is possible to run a LFS *rump file system* in read-only mode. Write support in the future depends both on building a communication mechanism with the cleaner and being able to synchronize with it. It remains to be seen how this is going to be implemented, but the author's suggestion is to move the cleaner from the userspace into the kernel to reduce complexity. First, *rump*s make it possible to experiment with cleaning policies in userspace even if the cleaner is "in the kernel". Second, it is possible to separate policy from mechanism and use difference cleaning algorithms even if the two components share the implementation domain [27].

4.11 Foreign Platforms

Using NetBSD kernel file system code on a foreign platforms is a two-faceted question. First, the file system code and assorted libraries must be buildable as a userspace program on the foreign host. This part depends only on POSIX functionality required by *rumpuser*. It can be solved by legwork and cleaning interfaces to remove any assumptions about the host.

Second, if desiring to mount the *rump file system*, a mechanism similar to *puffs+p2k* must exist on the platform. This has not yet been implemented on a non-NetBSD platform, but the wide availability of the FUSE [1] userspace file systems interface, especially the low level version, makes it a good candidate for experimentation.

4.11.1 Building NetBSD *rump*s on Linux

The *rump* build is heavily dependent on the NetBSD build system and tools. However, NetBSD is fully cross-compilable out of the box and it is possible to build a cross toolchain for UNIX type systems by running one single command [25]. But as the cross-compile mechanism has been developed for producing NetBSD target binaries on other hosts, it is not directly applicable - we wish to produce host binaries using the NetBSD build framework. This is possible through appropriate commands. The variables `USETOOLS_BINUTILS` and `USETOOLS_GCC` must be set to `"no"`. This causes the host toolchain to be used instead of the NetBSD target toolchain.

Also, it is not possible to build libp2k on Linux, since it is NetBSD-specific, and must be skipped.

4.11.2 Namespaces

Earlier, in Chapter 3, we discussed the namespace separation between kernel and user programs. When compiling *rump*s on a foreign host, we are greeted with even stricter restrictions: we may not include any of the *rump* platform's kernel headers from user programs, as this will result in collisions with the host platform's kernel headers.

Instead, we must take the separation even further and provide all *rump* platform kernel symbols we wish to call from user code through librump headers. How do best do this depends on the case.

For example, the vnode type definitions are simply copied from the kernel sources to a local header during the build process, but are wrapped in a preprocessor check for systems such as NetBSD which already provide the symbols:

```
#ifndef __VTYPE_DEFINED
#define __VTYPE_DEFINED
enum vtype { VNON, VREG, VDIR, VBLK, VCHR,
             VLNK, VSOCK, VFIFO, VBAD };
#endif /* __VTYPE_DEFINED */
```

The vnode operations normally namespaced *VOP* are included as a duplicate copy in the automatically generated *RUMP_VOP* namespace. We must provide two copies to facilitate both between userspace callers and kernel callers.

Additionally, the interfaces provided by librump have to be split into ones which are internal to the library and might need kernel headers (*rump_private.h*), and public ones (*rump.h*). The latter header is legal for inclusion from user programs, and it includes the definitions we mentioned earlier, e.g. the *RUMP_VOP* method interfaces and forward declarations for some kernel types.

4.11.3 ABI Considerations

Linking NetBSD kernel code compiled with the NetBSD kernel headers to Linux a userspace program compiled with Linux user and kernel headers is technically not legal. There are no guarantees that that the application binary interfaces for both are identical and will therefore work when linked together. Luckily, in practice the differences are minimal.

The only problem that was discovered when testing on i386 hardware was related to the `off_t`. On Linux you need specific compilation settings to make `off_t` 64bit, while it is always 64bit on NetBSD. This caused obvious malfunctioning and was fixed by using `#ifdef __linux__` to instruct the Linux compilation to use 64bit `off_t`.

5. EXPERIENCES AND USE

This chapter discusses experiences with the *rump* framework and associated file systems. The framework was added to the public NetBSD source tree a little over a month ago before writing this, so experience at this point is still limited. However, the initial feedback especially from people doing file system development has been extremely positive.

5.1 Use on Existing File Systems

The author has personally used *rump file system*s for debugging several file system problems. Cases have included improper evaluation order, lock protocol violation and invalid on-disk structure management. In all cases the environment performed like it was assumed to before developing it: with a short retry cycle, with working debugger support, and with a core dump instead of system crash in case of a "kernel panic".

5.2 Development of New File Systems

The *rump* framework was developed by taking existing kernel file systems and making them run in userspace. The userspace *rump* environment is currently more lax than the kernel and permits situations the kernel will not permit. This is not a problem for moving the kernel file system code to userspace for testing, but will impact developing file systems first in userspace.

Also, *rump file system*s might not duplicate all corner cases accurately with respect to the kernel. For example, Zhang & Ghose [30] list problems related to flushing resources as the implementation issues with using BSD VFS. Theoretically, the flushing behavior can be different if the file system code is running in userspace, and therefore some bugs might be left unnoticed. On the flip-side, the potentially different behavior might expose bugs otherwise very hard to detect when running in the kernel. The livelock issue we already covered in Chapter 4.10.1 was an example of this.

Currently there is no data available on how many problems will be met if developing a file system initially in userspace and then moving it to the kernel. As experience in that area is gained, discrepancies in behavior will be handled.

5.3 Maintenance

Although librump has been developed minimizing duplicated code, there are small amounts of it. Handling the problem of keeping the emulating code in sync with the kernel code can be argued to be a problem mostly in software engineering management.

Making changes to the kernel can cause two types of breakage in the *rump* framework: the type that the compiler will be able to detect and the type it will not be able to detect.

The first type is easy to battle: building the *rump* tree will show if faults have been introduced. NetBSD is cross compiled daily for all architectures by a build cluster [2]. The compilation of the *rump* tree has been enabled by default and breakage will result in build failures. Problems may be fixed by anyone in the community and people beyond the author have already dealt with build problems on esoteric architectures the author does not normally build for.

The second type involves changing the semantics of interfaces implemented separately by the kernel and librump. These problems will be detected only at runtime when the software does not function as expected. As such changes are relatively rare and in addition the NetBSD source changes receive widespread review from the kernel developers, the author included, this is not expected to be a problem. If it

Figure 6: kernel compilation

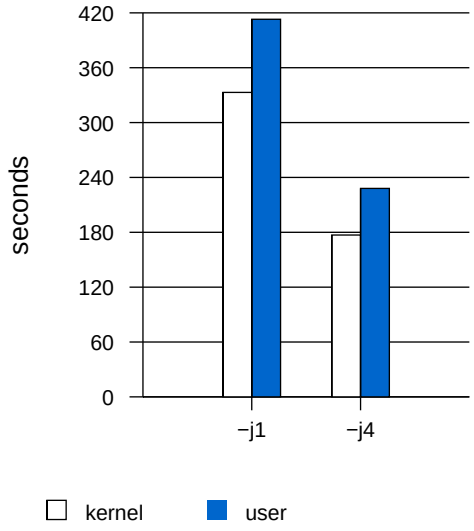
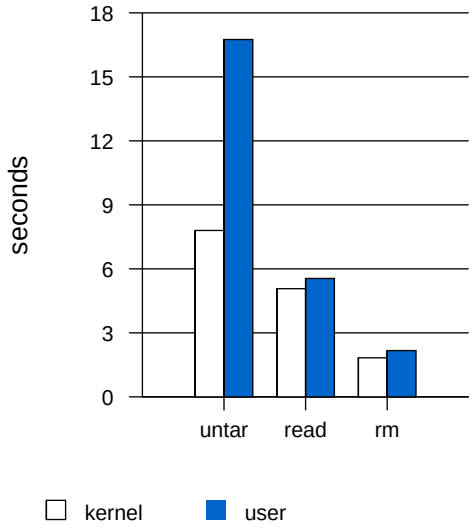


Figure 7: file operations



is discovered to be so, methods for detecting discrepancies in use [12] may be employed.

5.4 Performance

While the main motivation for this work is testing and development, speed never hurt anyone. The following measurements present the performance of real world tasks. Microbenchmarks are not presented for a better view of how much the difference between running the same code userspace or the kernel is. The purpose here is to show that it is not inconceivable to run file systems in userspace for example when hunting for a seldomly occurring bug in desktop use.

A test for kernel compilation times is presented in Figure 6. The "j1" case is with one concurrent compiler, while "j4" runs four compilers concurrently. The second set of tests in Figure 7 measures normal file operations: untarring a file, finding and reading all files in a directory, and removing the directory hierarchy. All tests are performed on a file system residing on a normal hard disk partition and using FFS.

The first test clearly shows that the current implementation is single threaded. The penalty for the userspace implementation is compounded in the "j4" case, where userspace is proportionally slower than in "j1".

It is difficult to make the measurements completely fair. The kernel has much tighter control over what it writes as synchronous writes and what is done asynchronously. In userspace, the choice is either to write asynchronously or flush *all* outstanding operations. Currently librump issues a flush in strategy if the current write is synchronous. This is evident especially in the write-heavy untar measurement. If done with purely asynchronous disk access, the duration is under 12 seconds. Other tests also confirm it is observable that I/O dominates - removal time is a third *faster* in userspace if using asynchronous metadata writes when the kernel version uses synchronous metadata writes.

6. EXTENDING, FUTURE WORK

After developing the system, it quickly became obvious that the scheme will be able to lend itself to more uses than just mounting a kernel file systems as a process.

6.1 File System Image Creation

A full distribution build of NetBSD requires to build file system images for boot media. Historically this was done by loopback mounting a file system image. However, this both required root privileges and build host kernel support for the file system being generated. Currently this is done with a tool called *makefs*, which takes a directory hierarchy and generates a file system image out of it [25]. The tool currently supports FFS and cd9660.

The FFS implementation is made up of a modified version of the buffer cache interface along with a modified copy of the kernel FFS code. These have been edited to fit the modified buffer cache interface and other userspace requirements.

By using the *ukfs* interface to implement similar functionality, the modified copy of the file system in userspace along with the buffer cache interface modification can be removed. Additionally, support for all kernel file systems which support writing can be added to *makefs*. This requires adding code equivalent to *newfs* for creating the empty file system, but the functionality for populating the newly created file system comes for free from the kernel implementation.

6.2 File System Migration

Migration of components is desirable for various load balancing and fault tolerance purposes. Migrating a kernel file system to another host is relatively simple if the kernel file system has been running in userspace. It involves recreating the current state on the migration target. As all file system operations go through libpuffs, the library can build a log of what needs to be replayed on the migration target to create the current file system state.

Migration between the kernel and userspace is more complex, as there is no functionality in the kernel to build the file system's current state. Neither is it easy to add one there - "kernel programming is difficult" is the premise for this entire work.

6.3 Other Subsystems

Extending the functionality to encompass other kernel subsystems beyond file systems has two aspects to it. First, it must be possible to link the code into a userspace program. In other words, enough support must be added to librump to make this possible. Second, the kernel subsystems must be modified so that they understand userspace components. Naturally, this is not needed if only wanting *ukfs*-style testing ability. Some initial thoughts on the subject and contrasts to file systems are presented next.

File systems are easy for running in userspace in the sense that they do not bind globally, but on a specific mountpoint. The same does not apply to, for example, networking protocols, where we have only protocol-level bindings for code. For instance, we are unlikely to want to route all TCP traffic through userspace. This can be solved through packet filters like in the Exokernel [13], with the difference that we specifically want to run the data via userspace. Also, a networking stack does not share the request-response characteristic of a file system, but rather is a processing entity.

Character device nodes are much more like file systems in the sense that they are issued requests and they produce responses. However, they are globally bound. A device node with the same major and minor numbers at point `/a` is exactly the same instance as the node with same major and minor numbers located at point `/b`. A minor amount of effort is required to dynamically register userspace device driver bindings.

6.4 Locking and Multithreading

The work presented in this paper operates in a single thread. As mentioned in Chapter 4.2, the current implementation runs all requests from start to finish in one go and does not implement real locks or the system interrupt priority level.

This means that locking and timing problems cannot be detected by means of dynamic analysis when running the program. This is rarely a problem when testing and extending existing file system, but can lead to problems if developing from scratch in userspace.

Luckily emulating kernel multitasking is fairly easy: we can use condition variables for `ltsleep`() / `wakeup`(). For interrupt priorities, the only "interrupt" we generate is when the device strategy routine finishes. We can use reader-writer locks for this. Raising the interrupt priority level takes a read lock while "delivering" an interrupt takes a write lock.

6.5 Integration

As *rump* technology is new, it is implemented in the least intrusive manner possible. When its potentials become clearer, it can be better integrated into the system.

Currently, *rump* is built as a completely separate subtree in the NetBSD kernel sources. No installation of binaries or libraries into the system is done and everything must either be run from the build tree or manually copied elsewhere. This is because interface provided by librump depends on the kernel internal interfaces at the time of compilation. This could be solved with a naming convention or library major number, but changing the name requires modifications to the library set lists, which include the names of all libraries installed on the system. While installing a library sounds like a simple enough problem, there are many details which need to be taken into account on a real-world system.

In Chapter 3.5 the user program necessary for mounting a file system was mentioned. The implementations for the regular case and the *rump* case are currently separate. However, their only real difference is in the mount function they call: the mount system call or *rump* routines. Integration would reduce code duplication, but depends on the build process integration.

Build integration opens up another possibility. The file system library created by the *rump* build and a loadable kernel module are mostly equivalent in content. By linking the loadable kernel module against *rump* instead of loading it into the kernel, it is possible to run the kernel module in userspace without recompilation. However, as kernel modules are built against real machine headers instead of faux headers, problems discussed in Chapter 4.8 need rethinking.

6.6 Other Platforms

As mentioned in Chapter 4.11, the framework lends itself to running NetBSD kernel file system code in userspace on Linux, and other Unix-type systems follow easily.

The interesting question is can similar techniques be applied to, for example, Linux to produce Linux kernel file systems which are out-of-the-box runnable in userspace on all Unix-type operating systems. If the platform additionally supports *p2k+puffs* functionality for the Linux virtual file system interface, it is possible to mount and use the wealth of file systems available in Linux. As a minor real-world induced side remark, the userspace approach also steers clear of GPL licensing complications.

A cursory analysis has been conducted on doing the equivalent of *rump*s for Linux. The same techniques appear to apply. This is not surprising, as Linux is a Unix-type operating system apart from minor differences and alternate terminology.

7. CONCLUSIONS

This paper presented *rump file system*s, unmodified kernel file systems mountable and runnable in userspace. The impact of the existence of such technology is twofold. File systems can be developed in userspace using familiar kernel interfaces. Additionally, file systems written for an operating system supporting such a scheme can be used on virtually any operating system.

The technical aspects of the architecture as well as lessons learned were discussed in-depth. This discussion is meant to encourage adapting the scheme. Although this work is not about optimal performance, speed was shown to be more

than acceptable for running regular applications on top of the system during the development phase.

Finally, the framework has already been integrated into the NetBSD operating system where it has received much interest from the developer community. This shows the work to have potential to bring a difference in the real world.

Availability

The code and architecture described in this paper is available for download, examination and use under the BSD license from the NetBSD source tree development branch (known as *-current* or *HEAD*) in `src/sys/rump`. See the NetBSD web site [3] for more information on fetching the code.

Acknowledgments

The work in this paper has been supported by Google Summer of Code and the Finnish Cultural Foundation.

The author wishes to thank Paweł Jakub Dawidek, André Dolenc, Johannes Helander, Kirk McKusick, Heikki Saikkonen, Chuck Silvers, Bill Stouder-Studenmund and David Young for ideas, conversations, inspiring questions, answers and support.

8. REFERENCES

[1] FUSE - Filesystem in Userspace. http://fuse.sourceforge.net/.

[2] NetBSD: Summary of daily snapshot builds. http://releng.NetBSD.org/cgi-bin/builds.cgi.

[3] The NetBSD Project. http://www.NetBSD.org/.

[4] ZFS source tour. http://www.opensolaris.org/os/community/zfs/source/.

[5] J. Bonwick. Zfs: The last word in filesystems, Oct 31, 2005. blog entry.

[6] C. D. Cranor. *Design and Implementation of the UVM Virtual Memory System*. PhD thesis, Washington University, 1998.

[7] P. J. Dawidek. Porting the ZFS file system to the FreeBSD operating system. In *Proc. of AsiaBSDCon 2007*, pages 97–103, 2007.

[8] J. Dike. A user-mode port of the Linux kernel. In *ALS'00: Proc. of the 4th conference on 4th Annual Linux Showcase & Conference*, 2000.

[9] R. Draves and S. Cutshall. Unifying the user and kernel environments. Technical Report MSR-TR-97-10, Microsoft Research, 1997.

[10] E. Efrat. Recent security enhancements in NetBSD. In *Proc. of 5th European BSD Conference*, pages 35–50, 2006.

[11] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level infrastructure for network protocol development. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, pages 171–184, 2001.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proc. of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, 2001.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of Symposium on Operating Systems Principles*, pages 251–266, 1995.

[14] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid. UNIX as an application program. In *Proc. of USENIX Summer*, pages 87–95, 1990.

[15] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *HOTOS'05: Proc. of the 10th conference on Hot Topics in Operating Systems*, pages 1–6, 2005.

[16] J. Helander. Unix under Mach: The Lites server. Master's thesis, Helsinki University of Technology, 1994.

[17] V. Henson, A. van de Ven, A. Gud, and Z. Brown. ChunkFS: using divide-and-conquer to improve file system reliability and repair. In *HOTDEP'06: Proc. of the 2nd conference on Hot Topics in System Dependability*, 2006.

[18] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[19] A. Kantee. puffs - Pass-to-Userspace Framework File System. In *Proc. of AsiaBSDCon 2007*, pages 29–42, 2007.

[20] S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun UNIX. In *Proc. of USENIX Summer*, pages 238–247, 1986.

[21] M. K. McKusick. Implementing FFS. private communication, 2007.

[22] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4BSD operating system*. Addison Wesley, 1996.

[23] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proc. of USENIX*, pages 1–17, 1999.

[24] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[25] L. Mewburn and M. Green. build.sh: Cross-building NetBSD. In *Proc. of BSDCon*, pages 47–56, 2003.

[26] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI*, pages 89–100, 2007.

[27] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proc. of USENIX Winter*, pages 307–326, 1993.

[28] C. Silvers. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *USENIX Annual Technical Conference, FREENIX Track*, pages 285–290, 2000.

[29] P. Snyder. tmpfs: A virtual memory file system. In *Proc. of Autumn 1990 EUUG Conference*, pages 241–248, 1990.

[30] Z. Zhang and K. Ghose. hFS: a hybrid file system prototype for improving small file and metadata performance. In *Proc. of EuroSys*, pages 175–187, 2007.