Rump File Systems: Kernel Code Reborn

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

Abstract

When kernel functionality is desired in userspace, the common approach is to reimplement it for userspace interfaces. We show that use of existing kernel file systems in userspace programs is possible without modifying the kernel file system code base. Two different operating modes are explored: 1) a transparent mode, in which the file system is mounted in the typical fashion by using the kernel code as a userspace server, and 2) a standalone mode, in which applications can use a kernel file system as a library. The first mode provides isolation from the trusted computing base and a secure way for mounting untrusted file systems on a monolithic kernel. The second mode is useful for file system utilities and applications, such as populating an image or viewing the contents without requiring host operating system kernel support. Additional uses for both modes include debugging, development and testing.

The design and implementation of the Runnable Userspace Meta Program file system (rump fs) framework for NetBSD is presented. Using rump, ten disk-based file systems, a memory file system, a network file system and a userspace framework file system have been tested to be functional. File system performance for an estimated typical workload is found to be ±5% of kernel performance. The prototype of a similar framework for Linux was also implemented and portability was verified: Linux file systems work on NetBSD and NetBSD file systems work on Linux. Finally, the implementation is shown to be maintainable by examining the 1.5 year period it has been a part of NetBSD.

1 Introduction

Motivation. *"Userspace or kernel?"* A typical case of driver development starts with this exact question. The tradeoffs are classically well-understood: speed, efficiency and stability for the kernel or ease of programming and a more casual development style for userspace. The question stems from the different programming environments offered by the two choices. Even if code written for the kernel is designed to be run in userspace for testing, it is most likely implemented with `#ifdef`, crippled and does not support all features of kernel mode.

Typical operating system kernels offer multitudes of tested and working code with reuse potential. A good illustration is file system code, which in the case of most operating systems also comes with a virtual file system abstraction [18] making access file system independent.

By making kernel file systems function in userspace, existing code can be utilized for free in applications. We accomplished this by creating a shim layer to emulate enough of the kernel to make it possible to link and run the kernel file system code. Additionally, we have created supplementary components necessary to integrate the file system code with a running system, i.e. mount it as a userspace file server. Our scheme requires no modifications to existing kernel file system code.

We define a Runnable Userspace Meta Program file system (*rump fs*) to be kernel file system code used in a userspace application or as a userspace file server.

Results. NetBSD [20] is a free 4.4BSD derived OS running on over 50 platforms and used in the industry especially in embedded systems and servers. A real world usable implementation of rump file systems, included in NetBSD since August 2007, has been written.

The following NetBSD kernel file systems are *usable and mountable in userspace without source modifications*: cd9660, EFS, Ext2fs, FFS, HFS+, LFS, MSDOSFS, NFS (client[1]), NTFS, puffs, SysVBFS, tmpfs, and UDF. All are supported from the same codebase without file system specific custom code.

Additionally, a quick prototype of a similar system for the Linux kernel was implemented. Under it, the relatively simple jffs2 [31] journaling file system from the Linux kernel is mountable as a userspace server on NetBSD. Other Linux file systems could also be made

to work using the same scheme, but since they are more complex than jffs2, additional effort would be required.

Finally, we introduce the *fs-utils* suite and an improved *makefs* utility [19]. Both use rump for generic file system access and do not implement private userspace file system drivers. In contrast, software packages such as mtools and e2fsprogs reimplement thousands of lines of file system code to handle a single file system.

Contributions. This paper shows that it is possible and convenient to *run pre-existing kernel file system code in a userspace application*. This approach has been desired before: Yang et al. described it as ideal for their needs but rated implementation hopelessly difficult [32].

We also describe a way to make a monolithic style kernel operate like a multiserver microkernel. In contrast to previous work, our approach *gives the user the choice of micro- or monolithic kernel* operation, thereby avoiding the need for the performance discussion.

The paper also shows it is possible to use kernel code in userspace on top of a POSIX environment irrespective of the kernel platform the code was originally written for. This paves way to thinking about *kernel modules as reusable operating system independent components*.

Userspace file systems. This paper involves file systems in userspace but it is not a paper on userspace fs frameworks. Userspace fs frameworks provide a programming interface for the file server to attach to and a method for transporting file system requests in and out of the kernel. This paper explores running kernel file system code as an application in userspace. Our approach requires a userspace fs framework only in case mounting the resulting rump file system is desired. The choice of the framework is mostly orthogonal. puffs [15] was chosen because of the author's familiarity and because it is the native solution on NetBSD. Similarly, would the focus of implementation have been Linux or Windows NT, the choice could have been FUSE [28] or FIFS [3].

Paper organization. The rest of this paper is organized as follows: Section 2 deals with architecture issues. Some major details of the implementation are discussed in Section 3. The work is measured and evaluated in Section 4. Section 5 surveys related work and finally Section 6 provides conclusions and outlines future work.

2 Architecture

Before going into details about the architecture of the implementation, let us recall how file systems are implemented in a monolithic kernel such as NetBSD or Linux.

- The interface through which the file system is accessed is the virtual file system interface [18]. It provides virtual nodes as abstract objects for accessing files independent of the file system type.

- To access the file system backend, the file system implementation uses the necessary routines from the kernel. These are for example the disk driver for a disk-based file system such as FFS, sockets and the networking stack for NFS or the virtual memory subsystem for tmpfs [27]. Access is usually done through the buffer cache.

- For file content caching and memory mapped I/O a file system is heavily tied to the virtual memory subsystem [25]. In addition to the pager's get and put routines, various supporting routines are required. This integration also provides the page cache.

- Finally, a file system uses various kernel services. Examples range from a hashing algorithm to timer routines and memory allocation. The kernel also performs access brokering and makes sure the same image is not mounted twice.

If the reuse of file system code in userspace is desired, all of these interfaces must be provided in userspace. As most parts of the kernel do not have anything to do with hardware but rather just implement algorithms, they can be simply linked to a userspace program. We define such code to be *environment independent* (EI). On the other hand, for example device drivers, scheduling routines and CPU routines are *environment dependent* (ED) and must be reimplemented.

2.1 Kernel and Userspace Namespace

To be able to understand the general architecture, it is important to note the difference between the namespaces defined by the C headers for kernel and for user code. Selection of the namespace is usually done with the preprocessor, e.g. `-D_KERNEL`. Any given module must be *compiled* in either the kernel or user namespace. After compilation the modules from different namespaces can be *linked* together, assuming that the application binary interface (ABI) is the same.

To emulate the kernel, we must be able to make user namespace calls, such as memory allocation and I/O. However, code cannot use kernel and user namespaces simultaneously due to collisions. For example, on a BSD-based system, the libc `malloc()` takes one parameter while the kernel interface takes three. 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 unification.

The namespace collision issue is even more severe if we wish to use rump file systems on foreign platforms. We cannot depend on anything in the NetBSD kernel namespace to be available on other systems. Worse,

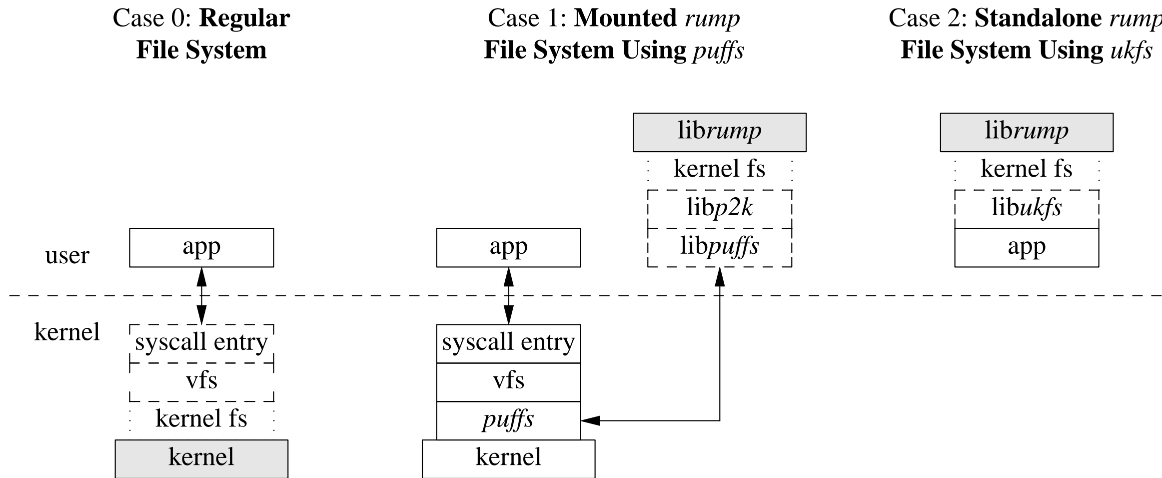

Figure 1: Rump File System Architecture

we cannot include any headers from the NetBSD kernel namespace in applications on other platforms, since it will create conflicts. For example, think what will happen if an application includes both the native and NetBSD `<sys/stat.h>`. To address this, we provide a namespace which applications can use to make calls to the rump kernel namespace. For example, the kernel vnode operation `VOP_READ()` is available under the name `RUMP_VOP_READ()`.

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

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

int rumpuser_thread_create(void *(*f)(void*), void*);

void rumpuser_mutex_enter(struct rumpuser_mtx *);
```

Figure 2: Examples of *rumpuser* interfaces

2.2 Component 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 (Case 0). To give context, "Regular File System" shows a file system in the kernel. Requests from an application are routed through the system call layer and vfs to be handled by the file system driver.

Mounted rump File System Using puffs (Case 1). From the application perspective, a mounted rump file system behaves like the same file system running in the kernel. The NetBSD userspace file systems framework, puffs [15], is used to attach the file system to the kernel.

Standalone rump File System Using ukfs (Case 2). A standalone rump file system is not mounted into the file system namespace. Rather, applications use a special API to mount and access the file system. While this requires code level awareness, it allows complete freedom, including the ability to target a certain file system and make calls past the virtual file system abstraction. The key benefits of standalone rump file systems are that they do not require kernel support or the use of operations normally reserved for the superuser, e.g. `mount()`.

2.3 The File System Driver

The file system driver is compiled as a regular userspace shared library. It can be linked directly into the file server or loaded dynamically at runtime using `dlopen()`.

2.4 librump

The interfaces required by a file system were classified in the beginning of Section 2. The component to provide these interfaces in the absence of the real kernel is librump. It emulates enough of the kernel environment for the file system code to be able to run.

To solve the namespace problem described in Section 2.1, librump is split into two: rumpkern and rumpuser. The first is compiled as a kernel component and the latter as a userspace component. Both must be linked into rump file systems.

Figure 2 presents examples of routines provided by rumpuser. There are two main classes of calls provided by rumpuser: system calls and thread library calls. Additionally, support calls such as memory allocation exist.

A convenient observation is to note that the file systems only call routines within themselves and interfaces in our case provided by rumpkern. Rumpkern only calls routines within itself, the file system (via callbacks) and

Component	# of lines
rumpuser	491
rumpkern (ED)	3552
std kern (EI)	27137
puffs (kernel)	3411
FFS	14912

Table 1: rump library size analysis

rumpuser. Therefore, by closure, rumpuser is the component defining the portability of a rump file system.

Librump was engineered bottom-up to provide kernel interfaces for file systems. Most of the interfaces could be used from the kernel source tree as such (environment independent), but some had to be reimplemented for userspace (environment dependent). For example, the vm subsystem was completely reimplemented for rump and could be simplified from tens of thousands of lines of code to just hundreds of lines because of most vm functionality being irrelevant in userspace. Table 1 shows effort in lines of code without comments or empty lines. Two kernel file systems are included for comparison. Code size is revisited in Section 4.5.

2.5 libp2k

Mounted rump file systems (Case 1, Figure 1) use the NetBSD userspace file systems framework, puffs [15]. We rely on two key features. The first one is transporting file system requests to userspace, calling the file server, and sending the results back. The second one is handling an abruptly unmounted file system, such as a file server crash. The latter prevents any kernel damage in the case of a misbehaving file server.

puffs provides an interface for implementing a userspace file systems. While this interface is clearly heavily influenced by the virtual file system interface, there are multiple differences. For example, the kernel virtual file system identifies files by a `struct vnode` pointer, whereas puffs identifies files using a `puffs_cookie_t` value. Another example of a parameter difference is the `struct uio` parameter. In the kernel this is used to inform the file system how much data to copy and in which address space. puffs passes this information to the read interface as a pointer where to copy to along with the byte count - the address space would make no difference since a normal userspace process can only copy to addresses mapped in its vmspace. In both cases the main idea is the same but details differ.

The *p2k*, or puffs-to-kernel, library is a request translator between the puffs userspace file system interface and the kernel virtual file system interface. It also interprets the results from the kernel file systems and converts them back to a format that puffs understands.

```
int
p2k_node_read(struct puffs_usermount *pu,
        puffs_cookie_t 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;
}
```

Figure 3: `p2k_node_read()` Implementation

To give an example of p2k operation, we discuss reading a file. This is illustrated by the p2k read routine in Figure 3. 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 opaque *kauth_cred_t* type used in the kernel. This is done by the p2k library's `cred_create()` routine, which in turn uses `rump_cred_create()`. The `VLS()` and `VUL()` macros in p2k to deal with NetBSD kernel virtual file system locking protocol. They take a shared (read) lock on the vnode and unlock it, respectively.

Mount utilities and file servers

Standard kernel file systems are mounted with utilities such as `mount_efs`, `mount_tmpfs`, etc. These utilities parse the command line arguments and call the `mount()` system call.

Our equivalent mountable rump file system counterparts are called `rump_efs`, `rump_tmpfs`, etc. Instead of calling the regular mount call, they attach to the system by p2k and rump. To maximize integration, these file servers share the same command line argument parsing code with the regular mount utilities. This was done by restructuring the mount utilities to provide an interface for command line argument parsing.

Sharing the argument parsing means that the file servers have the same syntax and makes usage interchangeable just by altering the command name. We also modified the system to handle a `rump` option in `/etc/fstab`. This allows to toggle certain mountpoints such as USB devices and CD/DVD to be handled using rump file systems by just adding one option.

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

int ukfs_modload(const char *libpath);
int ukfs_modload_dir(const char *directory);

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

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

Figure 4: Examples of ukfs interfaces

2.6 libukfs

The ukfs library, or user-kernel file system, provides a standalone approach (Case 2 from Figure 1). Two classes of interfaces are provided by libukfs, both having examples in Figure 4, and are discussed below:

Initialization. To use a file system, it must be virtually mounted. The mount call returns a `struct ukfs` handle which is passed to all other calls. This handle is analogous to the mountpoint path in a mounted file system.

Additionally, routines for dynamically loading file system libraries are provided. This is similar to loading kernel modules, but since we are in userspace, `dlopen`() is used for loading.

File system access. Accessing file system contents is done with calls in this class. Most calls have an interface similar to system calls, but as they are self-contained, they take a filename instead of for example requiring a separate open before passing a file descriptor to a call. The rootpath is the root of the file system, but the library provides tracking of the current working directory, so passing non-absolute paths is possible.

If an application wishes to do low level calls such as vfs operations for performance or functionality reasons, it is free to do so even if it additionally uses ukfs routines.

3 Implementation

This section deals with major points of interest in the implementation. While the discussion is written with NetBSD terminology, it attempts to take into account the general case with all operating systems.

3.1 Disk Driver

A disk block device driver provides storage medium access and is instrumental to the operation of disk-based file systems. The main interface is simple: a request instructs the driver to read or write a given number of sectors at a given offset. The disk driver queues the request and returns. The request is handled in an order according to a set policy, e.g. the disk head elevator. The request must be handled in a timely manner, since during the period that the disk driver is handling the request the object the data belongs to (e.g. vm page) is held locked. Once the request is complete, the driver signals the kernel that the request has been completed. In case the caller waits for the request to complete, the request is said to be synchronous, otherwise asynchronous.

There are two types of backends: buffered and unbuffered. A buffered backend stores writes to a buffer and flushes them to storage later. An unbuffered backend will write to storage immediately. Examples are a regular file and a character device representing a disk partition, respectively. A block driver signals a completed write only after data has hit the disk. This means that a disk driver operating in userspace must make sure the data is not still in a kernel cache before it issues the signal.

There are three approaches to implementing the block driver using standard userspace interfaces.

- **Use `read()` and `write()` in caller context**: this is the simplest method. However, it effectively makes all requests synchronous and kills write performance.

- **Asynchronous read/write**: in this model the request is handed off to an I/O thread. When the request has been completed, the I/O thread issues an "interrupt" to signal completion.

  A buffered backend must flush synchronously executed writes. The only standard interface available for this is `fsync()`. However, it will flush all buffered data before returning, including previous asynchronous writes. Non-standard ranged interfaces such as `fsync_range()` exist, but they usually flush at least some file metadata in addition the the actual data causing extra unnecessary I/O.

  A userlevel write to an unbuffered backend goes directly to storage. The system call will return only after the write has been completed. No flushing is required, but since userlevel I/O is serialized in Unix, it is not possible to issue another write before the first one finishes. This means that a synchronous write must block and wait until an ongoing asynchronous write has been fully executed.

  The `O_DIRECT` file descriptor flag causes a write on a buffered backend to bypass cache and go directly to storage. The use of the flag also invalidates the cache for the written range, so it is safe to use in conjunction with buffered I/O. However, the flag is advisory. If conditions are not met, the I/O will silently fall back to the buffer. This method can therefore be used only when it applies for sure.

- **Memory-mapped I/O**: this method works only for regular files. The benefits are that the medium ac-

cess fastpath does not involve any system calls and that the `msync()` system call can be portably used to flush ranges instead of the whole memory cache.

The file can be mapped using windows. This provides two advantages. First, files larger than the available VA can be used. Second, in case of a crash, the core dump is only increased by the size of the windows instead of the size of the image. This is a very important pragmatic benefit. We found that the number of windows does not make a huge difference; we default to 16 1MB windows with LRU.

The downside of the memory mapping approach is that to overwrite data, the contents must first be paged in, then modified, and only after that written. This is to be contrasted to explicit I/O requests, where it is possible to decide if a whole page is being overwritten, and skip pagein before overwrite.

Of the above, we found that on buffered backends `O_DIRECT` works best. Ranged syncing and memory mapped I/O have roughly equal performance and full syncing performs poorly. The disk driver question is revisited in Section 4.6, where we compare performance against a kernel mount.

3.2 Locking and Multithreading

File systems make use of locking to avoid data corruption. Most file systems do not create separate threads, but use the context of the requesting thread to do the operations. In case of multiple requests there may be multiple threads in the file system and they must synchronize access. Also, some file systems create explicit threads, e.g. for garbage collection.

To support multithreaded file systems in userspace, we must solve various subproblems: locks, threads, legacy interfaces and the kernel giantlock. We rely on the userspace pthread library instead of implementing our own set of multithreading and synchronization routines.

Locks and condition variables. There are three different primitives in NetBSD: mutexes, rwlocks and condition variables. These map to pthread interfaces. The only differences are that the kernel routines are of type `void` while the pthread routines return a success value. However, an error from e.g. `pthread_mutex_lock()` means a program bug such as deadlock and in case of failure, the program is aborted and core is dumped.

Threads. The kernel provides interfaces to create and destroy threads. Apart from esoteric arguments such as binding the thread to a specific CPU, which we ignore, the kernel interfaces can be mapped to a pthread library calls. This means that `kthread_create()` will call `pthread_create()` with suitable arguments.

Kernel giantlock. Parts of the NetBSD kernel not converted to fine grained locking are still under the kernel biglock. This lock is special, as it is recursive and must be completely released when blocking. As all the system calls rump makes are in rumpuser, the blocking points are there also. We wrap the potentially blocking calls to drop and retake the biglock.

Legacy interfaces A historic BSD interface still in use in some parts of the kernel is `tsleep`(). It is a facility for waiting for events and maps to pthread condition variables.

Observations. It is clear that the NetBSD kernel and pthread locking and threading interfaces are very close to each other. However, there are minimal differences such as the naming and of course under the hood the implementations are different. Providing a common interface for both [8] would be a worthwhile exercise in engineering for a platform where this was not considered initially.

3.3 puffs as a rump file system

Using rump, puffs can be run in userspace on top of rump and a regular userspace file system on top of it. It gives the benefit of being able to access any file system via ukfs, regardless of whether it is a kernel file system or a userspace file system. Since puffs provides emulation for the FUSE interface [16] as well, any FUSE file system is usable through the same interface too. For instance, a utility which lists the directory tree of a file system works regardless of if the file system is the NetBSD kernel FFS or FUSE ntfs-3g.

Naturally, it would be possible to call userspace file system interfaces from applications without a system as complex as rump. However, since we already do have rump, we can provide total integration for all file systems with this scheme. It would be entirely possible to make ukfs use different callpaths based on the type of file system used. However, that would require protocol conversion in ukfs to e.g. FUSE. Since the puffs stack already speaks all the necessary protocols, it is more elegant to run everything through it.

3.4 Installation and Linking

Rump file systems are installed for userspace consumers as a number of separate libraries. The base libraries are: librump (rumpkern), librumpuser (rumpuser), libukfs and libp2k. Additionally, there are all the individual file system drivers such as librumpfs_efs, librumpfs_ntfs and so forth. To use rump file systems, the base libraries should be linked in during compilation. The file system driver libraries may be linked in during compilation or loaded dynamically.

The NetBSD kernel expects all built-in file systems to be stored in a *link set* for bootstrap initialization. A link set is a method for a source module to store information to a specific section in the object. A static linker unifies the section contents from all source modules into a link set when the program is linked. However, this scheme is not fully compatible with dynamic linking: the dynamic loader would have to create storage to accommodate for section information from each shared library. We discovered that a link set entry only from the first shared library on the linker command line is present runtime. We could have attempted to modify the dynamic linker to support this non-standard feature, but instead we chose to require dynamic loading of file systems when support for more than one is required. Loading is done using the ukfs interfaces described in Section 2.6.

Since the kernel environment is in constant flux, the standard choice of bumping the major library version for each ABI change did not seem reasonable. Instead, currently the compatibility between librump and the file system libraries is handled exactly like for kernel modules: both librump and the file system libraries are embedded with the ABI version they were built against. When a file system library is attached to librump the versions are compared and if incompatible the attach routine returns `EPROGMISMATCH`.

3.5 Foreign Platforms

Different kernel version. An implication of rump file systems is the ability to use file system code from a different OS version. While it is possible to load and unload kernel modules on the fly, they are closely tied by the kernel ABI. Since a rump file system is a self-contained userspace entity, it is possible to use a file system from a newer or older kernel. Reasons include taking advantage of a new file system feature without having to reboot or avoiding a bug present in newer code.

NetBSD rump file systems on Linux. This means using NetBSD kernel file systems on a Linux platform. As Linux does not support puffs, libp2k cannot be used. A port to FUSE would be required. Despite this, the file system code can be used via ukfs and accessing a file system using NetBSD kernel code on Linux has been verified to work. A notable fact is that structures are returned from ukfs using the ABI from the file system platform, e.g. `struct dirent` is in NetBSD format and must be interpreted by callers as such. Eventually, a component translating structures between different operating systems will be provided.

Linux kernel file systems on NetBSD. Running Linux kernel file systems on NetBSD is interesting because there are several file systems written against the Linux kernel which are not available natively in NetBSD or in more portable environments such as userspace via FUSE. Currently, our prototype Linux implementation supports only jffs2 [31]. This file system was chosen as the initial target because of its relative simplicity and because it has potential real-world use in NetBSD, as NetBSD lacks a wear-leveling flash file system.

An emulation library targeted for Linux kernel interfaces, *lump*, was created from scratch. In addition, a driver emulating the MTD flash interface used by jffs2 for the backend storage was implemented.

Finally, analogous to libp2k, we had to match the interface of puffs to the Linux kernel virtual file system interface. The main difference was that the Linux kernel has the *dcache* name cache layer in front of the virtual file system nodes instead of being controlled from within each file system individually. Other tasks were straightforward, such as converting the `struct kstat` type received from Linux to the `struct vattr` type expected by puffs and the NetBSD kernel.

ABI Considerations. Linking objects compiled against NetBSD headers to code compiled with Linux headers is strictly speaking not correct: there are no guarantees that the application binary interfaces for both are identical and will therefore work when linked together. However, the only problem encountered when testing on i386 hardware was related to the `off_t` type. On Linux, `off_t` is 32bit by default, while it is 64bit on NetBSD. Making the type 64bit on Linux made everything work.

If mixing components from different NetBSD versions, care must be taken. For example, `time_t` in NetBSD was recently changed from 32bit to 64bit. We must translate `time_t` in calls crossing this boundary.

4 Evaluation

To evaluate the usefulness of rump file systems, we discuss them from the perspectives of security, development uses, application uses, maintenance cost, and performance. We estimate the differences between a rump environment and a real kernel environment and the impact of the differences and provide anecdotal information on fixing several real world bugs using rump file systems.

4.1 Security

General purpose OS file systems are commonly written assuming that file system images contain trusted input. While this was true long ago, in the age of USB sticks and DVDs it no longer holds. Still, users mount untrusted file systems using kernel code. The BSD and Linux manual pages for mount warn: “*It is possible for a corrupted file system to cause a crash*”. Worse, arbitrary memory access is known to be possible and fixing each file system to be bullet-proof is at best extremely hard [32].

In a mounted rump file system the code dealing with the untrusted image is isolated in its own process, thus mitigating an attack. As was seen in Table 1, the size difference between a real kernel file system and the kernel portion of puffs is considerable, about five-fold. Since an OS usually supports more than one file system, the real code size difference is much higher. Additionally, puffs was written from ground up to deal with untrusted input.

To give an example of a useful scenario, a recent mailing list posting described a problem with mounting a FAT file system from a USB stick causing a kernel crash. By using a mountable rump file system, this problem was reduced to an application core dump. The problematic image was received from the reporter and problem in the kernel file system code was debugged and dealt with.

```
golem> rump_msdos ~/img/msdosfs.img /mnt
panic: buf mem pool index 23
Abort (core dumped)
golem>
```

4.2 Development and Debugging

Anyone who has ever done kernel development knows that the kernel is not the most pleasant environment for debugging and iteration. A common approach is to first develop the algorithms in userspace and later integrate them into the kernel, but this adds an extra phase.

The following items capture ways in which rump file systems are superior to any single existing method.

- **No separate development cycle**: There is no need to prototype with an ad-hoc userspace implementation before writing kernel code.
- **Same environment**: userspace operating systems and emulators provide a separate environment. Migrating applications (e.g. OpenOffice or FireFox) and network connections there may be challenging. Since rump integrates as a mountable file system on the development host, this problem does not exist.
- **No bit-rot**: There is no maintenance cost for case-specific userspace code because it does not exist.
- **Short test cycle**: The code-recompile-test cycle time is short and a crash results in a core dump and inaccessible files, not a kernel panic and total application failures.
- **Userspace tools**: dynamic analysis tools such as Valgrind [21] can be used to instrument the code. A normal debugger can be used.
- **Complete isolation**: Changing interface behavior for e.g. fault and crash injection [14, 23] purposes can be done without worrying about bringing the whole system down.

To give an example, support for allocating an in-fs journal was added to NetBSD ffs journaling. The author, Simon Burge, is a kernel developer who normally does not work on file systems. He used rump and ukfs for development and described the process thusly: "Instead of rebooting with a new kernel to test new code, I was just able to run a simple program, and debug any issues with gdb. It was also a lot safer working on a simple file system image in a file." [4].

Another benefit is prototyping. One of the reasons for implementing the 4.4BSD log-structured file system cleaner in userspace was the ability to easily try different cleaning algorithms [24]. Using rump file systems this can easily be done without having to split the runtime environment and pay the overhead for easy development during production use.

Although it is impossible to measure the ease of development by any formal method, we would like to draw the following analogy: kernel development on real hardware is to using emulators as using emulators is to developing as a userspace program.

Differences between environments

rump file systems do not duplicate all corner cases accurately with respect to the kernel. For example, Zhang and Ghose [34] list problems related to flushing resources as the challenging implementation issues with using BSD VFS. Theoretically, flushing behavior can be different if the file system code is running in userspace, and therefore bugs might be left unnoticed. On the flip-side, the potentially different behavior exposes bugs otherwise very hard to detect when running in the kernel. Rump file systems do not possess exactly the same timing properties and details of the real kernel environment. Our position is that this is not an issue.

Differences can also be a benefit. Varying usage patterns can expose bugs where they were hidden before. For example, the recent NetBSD problem report[2] kern/38057 described a FFS bug which occurs when the file system device node is not on FFS itself, e.g. /dev on tmpfs. Commonly, /dev is on FFS, so regular use did not trigger the problem. However, since this does not hold when using FFS through rump, the problem was triggered more easily. In fact, this problem was discovered by the author while working on the aforementioned journaling support by using rump file systems.

Another bug which triggered much more frequently by using rump file systems was a race which involved taking a socket lock in the nfs timer and the data being modified while blocking for the socket lock. This bug was originally described by the author in a kernel mailing list post entitled "how can the nfs timer work?". It caused the author to be pointed at a longstanding NFS

problem of unknown cause described in kern/38669. A more detailed report was later filed under kern/40491 and the problem subsequently fixed.

In our final example the kernel FAT file system driver used to ignore an out-of-space error when extending a file. The effect was that written data was accepted into the page cache, but could not be paged out to disk and was discarded without flagging an application error. The rump vnode pager is much less forgiving than the kernel vnode pager and panics if it does not find blocks which it can legally assume to be present. This drew attention to the problem and it was fixed by the author in revision 1.53 of the source module msdosfs_vnops.c.

Locks: Bohrbugs and Heisenbugs

Next we describe cases in which rump file systems have been used to debug real world file system locking problems in NetBSD.

The most reliably repeatable bugs in a kernel environment and a rump file system are ones which depend only on the input parameters and are independent of the environment and timing. Problem report kern/38219 described a situation where the tmpfs memory file system would try to lock against itself if given suitable arguments. This made it possible for an unprivileged user to panic the kernel with a simple program. A problem described in kern/41006 caused a dangling lock when the `mknod()` system call was called with certain parameters. Both cases were reproduced by running a regular test program against a mounted rump file systems, debugged, fixed and tested.

Triggering race conditions depends on being able to repeat timing details. Problem report kern/40948 described a bug which turned out to be a locking problem in an error branch of the FFS rename vnode operation. It was triggered when the rename source file was removed halfway through the operation. While this is a race condition, it was equally triggerable by using a kernel file system and a mounted rump file system. After being debugged by using rump and fixed, the same problem was reported for tmpfs in kern/41128. It was similarly debugged and dealt with.

Even if the situation depends on components not available in rump file systems, using rump may be helpful. Problem report kern/40389 described a bug which caused a race condition deadlock between the file system driver and the virtual memory code due to lock order reversal. The author wrote a patch which addressed the issue in the file system driver, but did not have a system for full testing available at that time. The suggested patch was tested by simulating the condition in rump. Later, when it was tested by another person in a real environment, the patch worked as expected.

Preventing undead bugs with regression testing

When a bug is fixed, it is good practice to make sure it does not resurface [17] by writing a regression test.

In the case of kernel regression tests, the test is commonly run against a live kernel. This means that to run the test, the test setup must first be upgraded with the test kernel, bootstrapped, and only then can the test be executed. In case the test kernel crashes, it is difficult to get an automatic report in batch testing.

Using a virtual machine helps a little, but issues still remain. Consider a casual open source developer who adds a feature or fixes a bug and to run the regression tests must 1) download or create a full OS configuration 2) upgrade the installation kernel and test programs 3) run tests. Most likely steps “1” and “2” will involve manual work and lead to a diminished likelihood of testing.

Standalone rump file systems are standalone programs, so they do not have the above mentioned setup complications. In addition to the test program, file system tests require an image to mount. This can be solved by creating a file system image dynamically in the test program and removing it once the test is done.

For example, our regression test for the ffs rename race (kern/40948) creates a 5MB FFS image to a regular file and runs the test against it. If the test survives for 10 seconds without crashing, it is deemed as successful. Otherwise, an error is reported:

```
Tests root: /srcs/tests/fs/ffs

t_renamerace (1/1): 1 test cases
    renamerace: Failed: Test case did not exit
                cleanly: Abort trap (core dumped)
```

4.3 File system access utilities: fs-utils

Several application suites exist for accessing and modifying file system images purely from userspace programs without having to mount the image. For example, Mtools accesses FAT file systems, ntfsprogs is used with NTFS and LTOOLS can access Ext2/3 and ReiserFS. While these suites generally provide the functionality of POSIX command line file utilities such as `ls` and `cp`, the name and usage of each command varies from suite to suite.

The fs-utils [33] suite envisioned by the author and implemented by Arnaud Ysmal has been done using standalone rump file systems. The utilities provide file system independent command line utilities with the same functionality and usage as the POSIX counterparts. The code from the POSIX utilities were used as a base for the implementation and the I/O calls were modified from system calls to ukfs calls.

For example, the `fsu_ls` program acts like regular `ls` and provides the same 31 flags as `ls`. The command `fsu_ls ~/img/ffs2.img -laF temp` produces

the long listing of the contents of the "/temp" directory of an FFS image located in the user's home directory and `fsu_ls /dev/rsd0e -laF temp` does the same for a FAT located on a USB stick. The file system type is autodetected based on the image contents. Other examples of utilities provided by fs-utils are `fsu_cat`, `fsu_find`, `fsu_chown` and `fsu_diff`.

Additionally, fs-utils provides utilities which are necessary to move data over the barrier between the host system fs namespace and the image. To illustrate the problem, let us consider `fsu_cp`. Like with `cp`, all pathnames given to it are either relative to the current working directory or absolute with respect to the root directory. Since for a standalone rump file system the root directory is the file system image root directory, `fsu_cp` can be used only to copy files within the image. Conversely, command output redirection (>) will write to a file on the host, not the image. To remedy these problems, fs-utils provides utilities such as `fsu_ecp`, which copies files over the boundary, as well as `fsu_write`, which reads from standard input and writes to a given file in the file system. For example, the command `ls | fsu_write ~/img/ffs2.img ls.txt` "redirects" the output of ls to the file /ls.txt on the image.

4.4 makefs

NetBSD is fully cross-compilable without superuser privileges on a POSIX system [19]. This capability is commonly referred to as *build.sh* after the shell script which bootstraps the build process. For the system to be cross-buildable the build process cannot rely on any non-standard kernel functionality to be available, since it might not exist on a non-NetBSD build host.

The canonical way to build a file system image for boot media used to be to create a regular file, mount it using the loopback driver, copy the files to the file system and unmount the image. This required the target file system to be supported on the build host and was not compatible with the goals of build.sh.

When build.sh was originally introduced to NetBSD, it came with a tool called *makefs*, which creates a file system image from a given directory tree using only application code. In other words, the makefs application contains the file system driver. This approach does not require privileges to mount a file system or support for the target file system in the kernel. The original utility had support for Berkeley FFS and was implemented by modifying and reimplementing the FFS kernel code to be able to run in userspace. This was the only good approach available at the time. Support for the ISO9660 CD file system was added later.

The process of makefs consists of four phases:

1. scan the source directory

	original	rump
FFS SLOC	1748	247
supported file systems	FFS, iso9660	FFS, ext2, iso9660, FAT, SysVBFS
FFS effort	$>$ 2.5 weeks or 100 hours	2 hours
total effort	7 weeks or 280 hours	2 days or 16 hours

Table 2: Comparison between original and rump makefs. Implementation effort for the original was provided by Luke Mewburn, the author of the original utility. The FFS figure stands only for driver implementation and does not include additional time spent debugging.

2. calculate target image size based on scan data
3. create the target image
4. copy source directory files to the target image

In the original version of makefs all of the phases as implemented in a single C program. Notably, phase 4 is the only one that requires a duplicate implementation of features offered by the kernel file system driver.

For comparison, we have implemented makefs using kernel file system drivers for phase 4. It is currently available as an unofficial alternative to the original makefs. We partially reuse code from the original makefs, since we need to analyze the source tree to determine the image size (phases 1&2). We rely on an external newfs/mkfs program for creating an empty file system image (phase 3). For phase 4 we use fs-utils and the `fsu_put` utility, which copies a directory hierarchy to a file system image. The exception is ISO9660 support, for which we use the original makefs utility; the kernel CD file system driver is read-only.

For phase 3, we had to make sure that the mkfs/newfs utility can create a file system to a regular file – typically such utilities operate on device special files. Out of the supported file systems, we had to add support for this to the NetBSD FAT and SysVBFS utilities. Support for each was approximately 100 lines of modification.

We compare the two implementations in Table 2. As can be observed, over a third of the original effort was for implementing support for a single file system driver. Since we reuse the kernel driver, we get this functionality for free. Additionally, `fsu_put` from fs-utils could be used as such. All of the FFS code for the rump implementation is involved in calculating the image size and was available from makefs. If code for this had not been available, we most likely would have implemented it using shell utilities. However, since determining the size involves multiple calculations such as dealing with hard links and rounding up directory entry sizes, we concluded that reusing working code was a better option.

Total commits to the kernel	9640
Total commits to rump	438
Commits touching only rump	347
Build fixes	17
Functionality fixes	5
Unique committers	30

Table 3: Commit analysis for the rump source tree from August 2007 to December 2008.

4.5 Maintaining rump in NetBSD

As rump implements environment dependent code in parallel with the the kernel, the implementation needs to keep up. There are two kinds breakage: the kind resulting in compile failure and the kind resulting in non-functional compiled code. The numbers in Table 3 have been collected from version control logs from the period August 2007 - December 2008, during which rump has been part of the official NetBSD source tree. The commits represent the number of changes on the main trunk.

The number of build fixes is calculated from the amount of commits that were done after the kernel was changed and rump not build anymore as a result. For example, a file system being changed to require a kernel interface not yet supported by rump is this kind of failure. Commits in which rump was patched along with the kernel proper were not counted in with this figure.

Similarly, functionality fixes include changes to kernel interfaces which prevented rump from working, in other words the build worked but running the code failed. Regular bugs are not included in this figure.

Unique committers represents the number of people from the NetBSD community who committed changes to the rump tree. The most common case was to keep up with changes in other parts of the kernel.

Based on observations, the most important factor in keeping rump functional in a changing kernel is educating developers about its existence and how to test it. Initially there was a lot of confusion in the community about how to test rump, but things have since gotten better.

It should be kept in mind that over the same time frame the NetBSD kernel underwent very heavy restructuring to better support multicore. As it was the heaviest set of changes over the past 15 years, the data should be considered "worst case" instead of "typical case".

For an idea of how much code there is to maintain, Figure 5 displays the number of lines of code lines in for rump in the NetBSD source tree. The count is without empty lines or comments. The number of lines of environment dependent code (rumpkern + rumpuser) has gone up from 1443 to 4043 (281%) while the number of code lines used directly from the kernel has gone up from 2894 to 27137 (938%). Features have been added, but

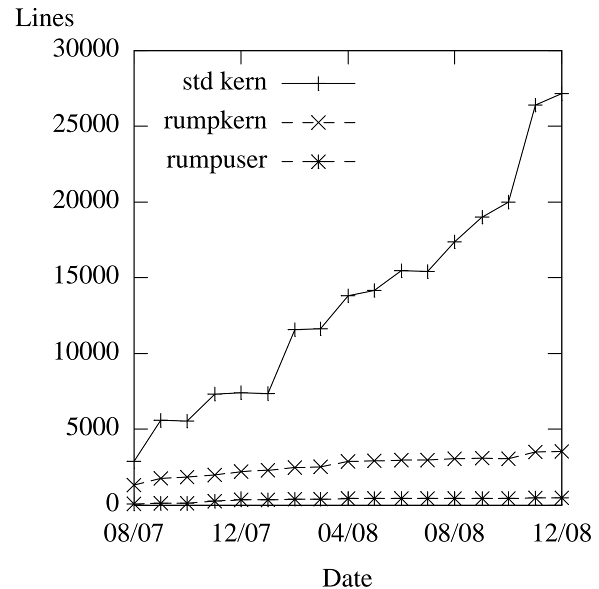

Figure 5: Lines of Code History

much of this has been done with environment independent code. Not only does this reduce code duplication, but it makes rump file systems behave closer to kernel file systems on a detailed level.

There have been two steep increases in code size. The first one was in January 2008, when all of the custom file system code written for userspace, such as namei, was replaced with kernel code. While functionality provided by the interfaces remained equivalent, the special case implementation for userspace was much smaller than the more general kernel code. The general code also required more complex emulation. The second big increase was in October 2008, when the kernel TCP/IP networking stack and support for socket I/O was added to rumpkern.

4.6 Performance

We measure the performance of three macro level operations: directory traversal with `ls -lR`, recursively copying a directory hierarchy containing both small and large files with `cp -R` and copying a large file with `cp`. For testing rump, standalone rump file systems were used with fs-utils. Mounted rump file systems were not measured, as they mostly test the performance of puffs and its kernel cache. For the copy operations the source data was precached. The figures are the duration from mount to operation to unmount.

The hardware used was a 2GHz Core2Duo PC laptop with a 100GB ATA drive. We performed the measurements on a 4GB FFS disk image hosted on a regular file and a 20GB FFS partition directly on the hard disk. Both file systems were aged [26]: the first one artificially

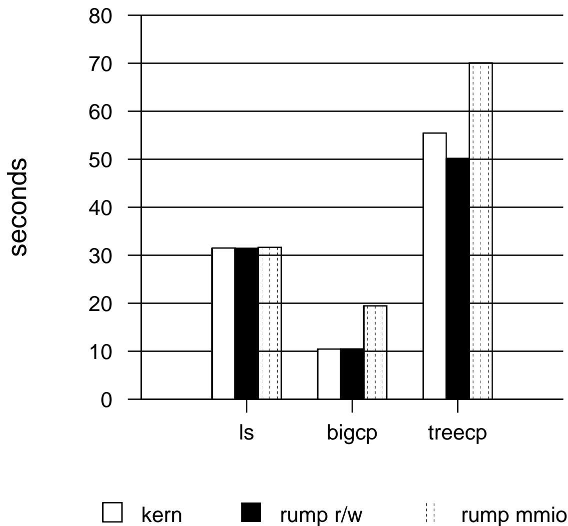

Figure 6: FFS on a regular file (buffered). rump r/w uses direct I/O, as discussed in Section 3.1.

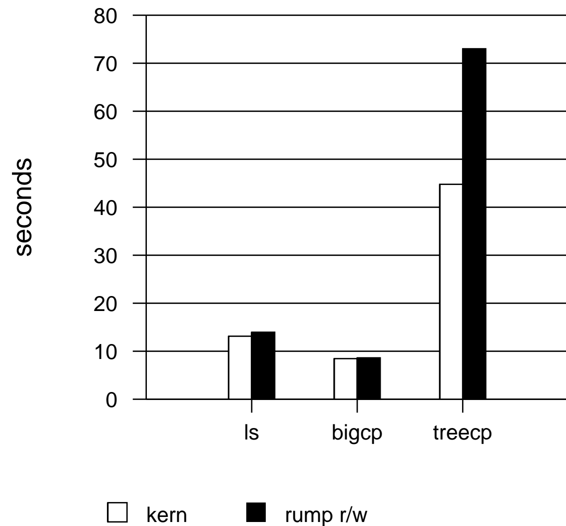

Figure 7: FFS on a HD partition (unbuffered), accessed through a character device.

by copying and deleting files. The latter one is in daily use on the author's laptop and has aged through natural use. The file systems were always mounted so that I/O is performed in the classic manner, i.e. FFS integrity is maintained by performing key metadata operations synchronously. This is to exacerbate the issues with a mix of async and sync I/O requests.

The results are presents in Figure 6 and Figure 7. The figures between the graphs are not directly comparable, as the file systems have a different layout and different aging. The CD image used for the large copy and the kernel source tree used for the treecopy are the same. The file systems have different contents, so the listing figures are not comparable at all.

Analysis. The results are in line with the expectations.

- The directory traversal shows that the read operations perform roughly the same on a regular file and 6% slower for an unbuffered backend. This difference is explained by the fact that the buffered file includes read ahead for a userspace consumer, while the kernel mount accesses the disk unbuffered.

- Copying the large file is 98.5% asynchronous data writes. Memory mapped I/O is almost twice as slow as read/write, since as explained in Section 3.1, the relevant parts of the image must be paged in before they can be overwritten and thus I/O bandwidth requirement is double. Unbuffered userspace read/write is 1.5% slower than the kernel mount.

- Copying a directory tree is a mix of directory metadata and file data operations and one third of the I/O is done synchronously in this case. The memory mapped case does not suffer as badly as the large copy, as locality is better. The rump read/write case performs 10% better than the kernel due to a buffered backend. The tradeoff is increased memory use. In the unbuffered case the problem of not being able to execute a synchronous write operation while an asynchronous one is in progress shows.

Notably, we did not look into modifying the NetBSD kernel to provide better system call interfaces for selective cache flushing and I/O to character devices. For now, we maintain that performance for the typical workload is acceptable when compared to a kernel mount.

5 Related Work

The Alpine [9] network protocol development infrastructure provides an environment for running kernel code in userspace. It is implemented before the system call layer by overriding libc and is run in application process context. 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 networking. Furthermore, from a file system perspective, this kind of approach shuts out kernel-initiated file system access, e.g. NFS servers.

Rialto [8] 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, we do not have the luxury to redo everything.

Mach is capable of running Unix as a user process [11]. Lites [12] is a Mach server 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 the debugger and one for the file system being developed, including applications using the file system. If the Lites server being debugged crashes, applications inside it will be terminated. Being a single server solution, it does not provide isolation from the trusted computing base, either. A multiserver microkernel such as SawMill [10] addresses the drawbacks of a single server, but does not permit a monolithic mode or use of the file system code as an application library.

Operating systems running in userspace, such as User Mode Linux [7], 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, for development purposes, this approach does not provide 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. Neither does a userspace operating system integrate into the host, i.e. it is not possible to mount the userspace operating system as a file server. Even if that could be addressed, booting an entire kernel every time a ukfs application is run is a very heavyweight solution.

Sun's ZFS file system ships with a userspace testing library, libzpool [1]. In addition to 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 program plugs directly to these components. This approach has several shortcomings compared to rump file systems. First, it does not include the entire file system architecture, e.g. the VFS layer. The effort of implementing the VFS interface (in ZFS terms the *ZFS POSIX Layer*) was specifically listed as the hardest part of porting ZFS to FreeBSD [6]. Second, it does not facilitate userspace testing with real applications because it cannot be mounted. Third, the test program is specific to ZFS.

Many projects reimplement file system code for userspace purposes. Examples include e2fsprogs [30] and mtools [22]. Their implementation overlaps that which is readily already provided by the kernel. Especially e2fsprogs must track Linux kernel features and perform an independent reimplementation.

fuse-ext2 [2] is a userspace file server built on top of e2fsprogs. It implements a translator from FUSE to e2fsprogs. The functionality provided by fuse-ext2 is the same as that if rump_ext2fs, but requires specifically written code. The ChunkFS [13] prototype is fully mountable, but it is implemented on FUSE and userspace interfaces instead of the kernel interfaces.

Simulators [5, 29] can be used to run traces on file systems. Thekkath et al. [29] go as far as to run the HPUX FFS implementation in userspace. However, these tools execute against a recorded trace and do not permit mounting.

6 Conclusions and Future Work

In this paper we described the *Runnable Userspace Meta Program file system (rump fs)* method for using preexisting kernel file system code in userspace. There are two different modes of use for the framework: the p2k mode in which file systems are mounted so that they can be accessed transparently from any application, and a standalone mode in which applications can use file system routines through the ukfs library interface. The first mode brings a multiserver microkernel touch to a monolithic kernel Unix OS, but preserves a user option for monolithic operation. The second mode enables reuse of the available kernel code in applications such as those involved in image access. Implementations discussed in this paper were makefs and fs-utils.

The NetBSD implementation was evaluated. We discovered that rump file systems have security benefits especially with untrusted removable media. Rump file systems made debugging and developing kernel file system code easier and more convenient, and did not require additional case-specific "glue code" for making kernel code runnable in userspace. The issues regarding the maintenance of the rump shim were examined by looking at over a year's worth of version control system commits. The build had broken 17 times and functionality 5 times. These were attributed to the lack of a full regression testing facility and developer awareness.

The performance of rump file systems using FFS was measured to be dependent of the type of backend. For file system images on a buffered regular file, properly synchronized performance was at best 10% faster than a kernel mount. Conversely, for a an unbuffered character device backend the performance was at worst 40% slower. We attribute lower unbuffered performance to there being no standard interfaces for intermingling synchronous and asynchronous writes. We estimate typical workload performance to be $\pm$5% of kernel mount performance. Future work may include optimizing performance, although for now we are fully content with it.

As a concluding remark, the technology has shown real world use and having kernel file systems from major open source operating systems available as portable userspace components would vastly increase system cross-pollination and reduce the need for reimplemen-

tations. We encourage kernel programmers to not only think about code from the classical machine dependent/machine independent viewpoint, but also from the environment dependent/environment independent perspective to promote code reuse.

Acknowledgments

This work has been funded by the Finnish Cultural Foundation, The Research Foundation of Helsinki University of Technology and Google Summer of Code.

The author thanks the anonymous reviewers for their comments and Remzi H. Arpaci-Dusseau, the shepherd, for his guidance in improving the final paper.

A special thank you goes to Arnaud Ysmal for implementing fs-utils. Additionally, the author thanks Simon Burge, André Dolenc, Johannes Helander, Luke Mewburn, Heikki Saikkonen, Chuck Silvers, Bill Stouder-Studenmund, Valeriy E. Ushakov and David Young for ideas, conversations, inspiring questions and answers.

Availability

The source code described in this paper is available for use and examination under the BSD license from the NetBSD source repository in the directory `src/sys/rump`. See `http://www.NetBSD.org/` for more information on how to obtain the source code.

References

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

[2] AKCAN, A. fuse-ext2. http://sourceforge.net/projects/fuse-ext2/.

[3] ALMEIDA, D. FIFS: a framework for implementing user-mode file systems in windows NT. In *WINSYM'99: Proc. of USENIX Windows NT Symposium* (1999).

[4] BIANCUZZI, F. Interview about NetBSD WAPBL. *BSD Magazine 2*, 1 (2009).

[5] BOSCH, P., AND MULLENDER, S. J. Cut-and-paste file-systems: Integrating simulators and file-systems. In *Proc. of USENIX* (1996), pp. 307–318.

[6] DAWIDEK, P. J. Porting the ZFS file system to the FreeBSD operating system. In *Proc. of AsiaBSDCon* (2007), pp. 97–103.

[7] DIKE, J. A user-mode port of the Linux kernel. In *ALS'00: Proc. of the 4th Annual Linux Showcase & Conference* (2000).

[8] DRAVES, R., AND CUTSHALL, S. Unifying the user and kernel environments. Tech. Rep. MSR-TR-97-10, Microsoft, 1997.

[9] ELY, D., SAVAGE, S., AND WETHERALL, D. Alpine: A User-Level infrastructure for network protocol development. In *Proc. of USITS'01* (2001), pp. 171–184.

[10] GEFFLAUT, A., JAEGER, T., PARK, Y., LIEDTKE, J., ELPHINSTONE, K. J., UHLIG, V., TIDSWELL, J. E., DELLER, L., AND REUTHER, L. The sawmill multiserver approach. In *Proc. of the 9th ACM SIGOPS Europ. workshop* (2000), pp. 109–114.

[11] GOLUB, D. B., DEAN, R. W., FORIN, A., AND RASHID, R. F. UNIX as an application program. In *Proc. of USENIX Summer* (1990), pp. 87–95.

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

[13] HENSON, V., VAN DE VEN, A., GUD, A., AND BROWN, Z. ChunkFS: using divide-and-conquer to improve file system reliability and repair. In *Proc. of HOTDEP'06* (2006).

[14] HSUEH, M.-C., TSAI, T. K., AND IYER, R. K. Fault injection techniques and tools. *IEEE Computer 30*, 4 (1997), 75–82.

[15] KANTEE, A. puffs - Pass-to-Userspace Framework File System. In *Proc. of AsiaBSDCon* (2007), pp. 29–42.

[16] KANTEE, A., AND CROOKS, A. ReFUSE: Userspace FUSE Reimplementation Using puffs. In *EuroBSDCon 2007* (2007).

[17] KERNIGHAN, B. Code testing and its role in teaching. *;login: The USENIX Magazine 31*, 2 (Apr. 2006), 9–18.

[18] KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in sun UNIX. In *Proc. of USENIX* (1986), pp. 238–247.

[19] MEWBURN, L., AND GREEN, M. build.sh: Cross-building NetBSD. In *Proc. of BSDCon* (2003), pp. 47–56.

[20] NETBSD PROJECT. http://www.NetBSD.org/.

[21] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI* (2007), pp. 89–100.

[22] NIEMI, D., AND KNAFF, A. Mtools, 2007. http://mtools.linux.lu/.

[23] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. *SIGOPS OSR 39*, 5 (2005), 206–220.

[24] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *Proc. of USENIX Winter* (1993), pp. 307–326.

[25] SILVERS, C. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *Proc. of USENIX, FREENIX Track* (2000), pp. 285–290.

[26] SMITH, K. A., AND SELTZER, M. I. File system aging—increasing the relevance of file system benchmarks. *SIGMETRICS Perform. Eval. Rev. 25*, 1 (1997), 203–213.

[27] SNYDER, P. tmpfs: A virtual memory file system. In *Proc. EUUG Conference* (1990), pp. 241–248.

[28] SZEREDI, M. Filesystem in USErspace. http://fuse.sourceforge.net/.

[29] THEKKATH, C. A., WILKES, J., AND LAZOWSKA, E. D. Techniques for file system simulation. *Software - Practice and Experience 24*, 11 (1994), 981–999.

[30] TS'O, T. E2fsprogs: Ext2/3/4 Filesystem Utilities, 2008. http://e2fsprogs.sourceforge.net/.

[31] WOODHOUSE, D. Jffs2 the journalling flash file system. In *Ottawa Linux Symposium* (2001).

[32] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *SP '06: Proc. of 2006 IEEE Symp. on Security and Privacy* (2006), IEEE Computer Society, pp. 243–257.

[33] YSMAL, A. FS Utils. http://NetBSD.org/˜stacktic/fs-utils.html.

[34] ZHANG, Z., AND GHOSE, K. hFS: a hybrid file system prototype for improving small file and metadata performance. In *Proc. of EuroSys* (2007), pp. 175–187.

Notes

[1] The kernel NFS server works in userspace, but is not yet part of the official source tree. There are conflicts between the RPC portmapper and mount protocol daemon for user- and kernel space nfs service. Basically, there is currently no way to differentiate if an exported directory hierarchy should be served by the kernel or userspace daemon.

[2] The NetBSD problem report database can be viewed with a web browser by accessing `http://gnats.NetBSD.org/<num>`, e.g. in the case of kern/38057 the URL is http://gnats.NetBSD.org/38057. The string "kern" stands for kernel and signifies the relevant subsystem.