ReFUSE: Userspace FUSE Reimplementation Using *puffs*

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

Alistair Crooks
The NetBSD Foundation
agc@NetBSD.org

Abstract

In an increasingly diverse and splintered world, interoperability rules. The ability to leverage code written for another platform means more time and resources for doing new and exciting research instead of reinventing the wheel. Interoperability requires standards, and as the saying goes, the best part of standards is that everyone can have their own. However, in the userspace file system world, the Linux-originated FUSE is the clear yardstick.

In this paper we present ReFUSE, a userspace implementation of the FUSE interface on top of the NetBSD native puffs (Pass-to-Userspace Framework File System) userspace file systems framework. We argue that an additional layer of indirection is the right solution here, as it allows for a more natural export of the kernel file system interface instead of emulating a foreign interface in the kernel. Doing so also reaps other minor benefits such as clarifying the license as the whole chain from the kernel to the file system is BSD-licensed. Another obvious benefit is having a fully performant native userspace file system interface available.

We summarize the puffs and FUSE interfaces and explain how the mapping between the two was done, including experiences from the implementation. After this we show by example that FUSE file systems work with ReFUSE as expected, present a virtual directory extension for the FUSE interface and conclude by outlining future work in the area.

1. Introduction

Userspace file systems are a growing phenomenon in Unix-type operating systems. Their use opens up many unforeseen opportunities. Most of these opportunities are related to being able to integrate information behind the file system namespace using userland tools and utilities. Examples include GmailFS [6] for using Gmail as a file system storage backend and FUSEPod [5] which facilitaties iPod access and management.

Userspace file systems operate by attaching an in-kernel file system to the kernel's virtual file system layer, vfs [17]. This component transforms incoming requests into a form suitable for delivery to userspace, sends the request to userspace, waits for a response, interprets the result and feeds it back to caller in the kernel. The kernel file system calling conventions dictate how to interface with the virtual file system layer, but other than that the userspace file systems framework is free to decide how to operate and what kind of interface to provide to userspace.

While extending the operating system to userspace is not a new idea [12, 21], the FUSE [2] userspace file systems interface is the first to become a veritable standard. It originated in Linux, but support has since been added to multiple different operating systems. This paper's main focus is exploring *ReFUSE*, the FUSE interface implementation in NetBSD [9]. *ReFUSE* is a reimplementation of a subset of the FUSE interfaces[1] and is implemented on top of the NetBSD's native userspace file systems framework: *puffs*, Pass-to-Userspace Framework File System. The goal is to eventually have the entire FUSE API supported.

The question, in essence, is where to implement compatibility for a foreign interface which is kernel-influenced. Unlike other systems which implement the whole FUSE chain starting from the kernel, the NetBSD implementation is only a lightweight emulation library in userspace. We maintain that this is the right way to implement support for a foreign component and nudge the community in the direction of more lightweight kernel components.

The remainder of this paper is laid out in the following manner. Chapter 2 gives a short overview of

[1] See Chapter 5 for deeper dissertation of what is implemented currently and what is not. The short version: enough to run almost all classes of file systems.

puffs necessary for the rest of the paper and Chapter 3 does the same for FUSE. Chapter 4 presents FUSE work in other operating systems. *ReFUSE* is presented in Chapter 5, while Chapter 6 talks about a virtual directory add-on the help with a commonly encountered problem in writing a FUSE file system. Chapter 7 summarizes key experiences with FUSE file systems under *ReFUSE*, including performance figures. Finally, Chapter 8 presents conclusions and outlines future efforts.

2. A Short Introduction to *puffs*

This chapter provides readers unfamiliar with *puffs* the necessary overview to be able to follow this paper. A more complete description of *puffs* can be found elsewhere [15, 16, 19]. Readers previously familiar with *puffs* may skip this chapter or merely skim over it.

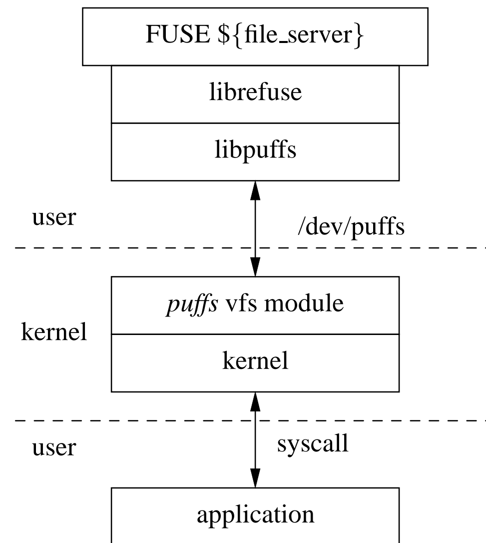
puffs is a framework for building file servers in userspace. It provides an interface similar to the kernel virtual file system interface, vfs [17], to a user process. *puffs* attaches itself to the kernel vfs layer. It passes requests it receives from the vfs interface in the kernel to userspace, waits for a result and provides the caller with the result. Applications and the rest of the kernel outside the vfs module cannot distinguish a file system implemented on top of *puffs* from a file system implemented purely in the kernel.

The file system interface in userspace consists of two sets of callbacks. Just as the kernel interface is divided into file system wide *VFS* operations and individual *vnode* operations (*VOP*s), the *puffs* interface is divided into *fs* operations and *node* operations. For reference, these callbacks are listed in Appendix B.

For the userspace implementation a library, libpuffs, is provided. libpuffs not only provides a programming interface to implement the file system on, but also includes convenience functionality frequently required for implementing file systems. An example of such convenience functionality is the pathname framework described later in this chapter.

To operate, a file server registers a number of callbacks with libpuffs and requests the kernel to mount the file system. After this, control is either passed to the library mainloop or kept with the caller which processes requests manually, although almost always the file server will want to hand control over to *puffs*. The library provides routines to decode file system requests from the kernel and call the appropriate callbacks. The results are passed to the kernel using routines provided by the library.

Figure 1. *puffs* and ReFUSE Architecture



2.1. Multitasking: Continuations

For multitasking the typical approach is for each application to create threads as it pleases. *puffs* takes a different route. It provides continuations as part of the library framework. A continuation cookie is passed to the file server from the framework. This cookie can be used to yield the current execution context and continue directly from it at a later moment. In a sense, it is much like threads with explicit scheduling points. For example the *puffs* framework for distributed file systems uses this extensively [16].

An important distinction between threads and continuations is that while continuation provide scheduling points to defeat the latency of e.g. network I/O, they do not parallelise. From a programmer's perspective explicit scheduling usually means a simpler program without having to worry about data structure locking.

2.2. Files and pathnames

Kernel file systems have only minimal involvement with file names. More importantly, the kernel's abstract file entity, the vnode, does not contain a pathname. This has several benefits. First, it avoids confusion with hardlinks where there are several pathnames referring to a single file. Second, it makes directory rename a cheap operation, since the pathnames of all nodes under the given directory do not need to be modified. Only operations which require

a pathname component are passed one. Examples are lookup, create and rmdir. The latter two require the pathname component to know the name of the directory entry they should modify.

However, most file system backends operate on paths and filenames. Examples include the sftp backend used by sshfs and in this case specifically the FUSE regular interface. To facilitate easier implementation of these file systems, *puffs* provides the mount flag `PUFFS_FLAG_BUILDPATH` to include full pathnames starting from the mountpoint in componentnames passed to interface functions as well as store the full path in nodes for direct use by the file server.

In addition to providing automatic support for building pathnames, *puffs* also provides hooks for file systems to register their own routines for pathname building in case a file system happens to support an alternative pathname scheme. This can be used to implement, for example, a layer making each pathname case-insensitive. In Chapter 7.3 this paper presents a comparison of the implementation of such a file system implemented both on top of the *puffs* and FUSE interfaces.

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

2.3. Current status and future development

After a year of active development in the NetBSD tree, *puffs* has gained a number of features apart from the basic support for implementing userspace file systems. In addition to the pathname framework mentioned above, there is support for:

- file server defineable file handles (file systems with "stable" files can use it for proper nfs export support)
- a generic event and network buffer framework for distributed file systems [16] (used by e.g. the sshfs and 9p file system drivers)
- kernel file system suspension API
- kernel cache flushing and invalidation operations
- notification messages for kernel page cache read and write accesses
- running unmodified kernel file systems such as ffs in userspace (useful especially for testing and development)

Current and future items under active development are improving the user-kernel communication barrier, providing accessors for the structures exposed by the interface in the early stages of development, adding support for caching metadata in the kernel and support for layering. Also, a code security audit of the kernel interface will soon be performed so that *puffs* can be enabled by default on NetBSD installations.

3. FUSE

This chapter summarizes FUSE for the parts necessary to understand the rest of this paper. Readers familiar with FUSE may skip this chapter or only skim over it.

FUSE stands for "Filesystem in Userspace" and, like *puffs*, provides an interface for building userspace file system servers. When we talk about FUSE in this paper, we are usually referring to the interfaces the FUSE userspace library, libfuse, provides. Of course, for a system to provide FUSE support, it must implement the full chain of intergration from the kernel virtual file system right up to the userspace file system server.

As *puffs* is influenced by the NetBSD kernel vfs interface, FUSE is influenced by the Linux kernel vfs interface. This is more evident from the semantics of how the interfaces are called instead of the interface linkage itself. For example, the *flush*() operation mimics the Linux kernel habits [3].

Similarly to *puffs*, FUSE can be logically split in two:

- A set of file system callback interfaces which are used to process incoming requests from the kernel.
- A set of forward operations which are used to control the operation of FUSE, e.g. `fuse_main`() and `fuse_opt_parse`().

Both of these are of interest to use if we wish to be compatible with file servers written against the FUSE interface.

Various language bindings are available for FUSE - the most used is, obviously, C, but C++, Python, Perl and Mono/C# bindings are all available. They enable file system development in other languages besides the native C: for example, the file system interface for GmailFS is implemented in Python.

3.1. Callback Interfaces

For the file system callbacks, FUSE provides two different interfaces against which to write a file system. A single file system will only implement one of these.

- The standard FUSE interface is based on path names. The operations resemble system calls more closely than the virtual file system interface. The file node the operation is affecting is identified using the pathname. This is a simple interface enough for all but the most demanding file systems. This interface is presented in Appendix C.
- The FUSE low level interface provides a completely different set of interface operations for the file system. It resembles the kernel virtual file system interface closely. Additionally, it requires that the file system manually handles all operation traffic between the file system and kernel. This interface is presented in Appendix D.

Almost all available FUSE file systems are written against the first interface. The most prominent examples of file systems written against the low level interface are ZFS, which is the FUSE implementation of Sun's ZFS and GlusterFS, which is a clustered file system, capable of scaling to several petabytes - it aggregates storage bricks over Infiniband or TCP/IP interconnect into one large parallel network file system.

To initialize the callback interfaces, C99 initializers are commonly used to fill out a structure. An example is presented in Figure 2.

FUSE presents a multi-threading interface, as well as the standard one, with the suffix "_mt" being added to the function names. This provides a speed up, and can be made very parallel. For example, the the ZFS file system running under FUSE on Linux uses multiple threads for listening to incoming requests and can go over 150 threads runtime [11].

3.2. Backward Compatibility

The FUSE interface has evolved over time. Supported systems such as Linux have to provide both

Figure 2. FUSE callback initialization

```
/* operations struct */
struct fuse_operations id3fs_ops = {
   .getattr = id3fs_getattr,
   .readlink = id3fs_readlink,
   .readdir = id3fs_readdir,
   .open = id3fs_open,
   .read = id3fs_read,
   .statfs = id3fs_statfs
};
```

Figure 3. FUSE backward compatibility

```
/*
 * Set the default version to use
 * for the fuse interface.  This value
 * determines the API to be used
 */

#ifndef FUSE_USE_VERSION
#define FUSE_USE_VERSION    26
#endif

#if FUSE_USE_VERSION >= 26
#define fuse_main(argc,argv,op,arg) \
    fuse_main_real(argc,argv,op,    \
      sizeof(*(op)), arg)
#else
#define fuse_main(argc, argv, op) \
    fuse_main_real(argc, argv, op,\
      sizeof(*(op)), NULL)
#endif
```

API and ABI compatibility for the old interfaces to keep old file systems running.

To enable older functionality, the `FUSE_USE_VERSION` definition is set by the file system writer to the value required.

The default FUSE API version currently is 2.6. Note that this number bears no connection with the Linux kernel version number. For example, to enable the FUSE 2.2 API, the file system writer would define `FUSE_USE_VERSION` to `22`.

4. Related Work

This section briefly discusses the measures other operating systems have taken to implement FUSE support.

The Linux FUSE implementation consists of both

Table 1. FUSE implementation sizes

OS	# of lines	% of Linux
Linux	5149	100%
OpenSolaris	5675	110%
FreeBSD	8788	171%
Mac OS X	12195	237%

libfuse and headers	10369

a kernel component and a userspace library. The kernel components of the Linux FUSE functionality are made available under the GPL and the userland library is distributed under the LGPL.

Other systems mentioned here use two parts from the original Linux implementation:

1. *libfuse*: the userspace library which provides the programming interface toward the userspace file server and communicates with the kernel. This means that the library implementation can be used as such, but also dictates that the kernel component must be compliant with the library protocol.

2. *fuse_kernel.h*: the header which describes the messages between the kernel and the library. This file is special in the sense that it is distributed in FUSE under a dual BSD/GPL license.

FreeBSD provides FUSE support in a package called fuse4bsd [4]. The FreeBSD FUSE module in the kernel is a file system implementation conforming to the FUSE user-kernel protocol using the original FUSE user-kernel protocol.

OpenSolaris FUSE support [8] is another kernel level reimplementation of the FUSE requirements. It is mostly licensed under CDDL, although a part of it is derived from the fuse4bsd implementation and comes with a BSD license.

MacFUSE [7] is also derived from the fuse4bsd project and is available under a 3-clause BSD license.

A very simple comparison of the various implementations is presented in the form of code lines for the kernel module in Table 1. This table is produced by calculating the raw number of lines in the comments, comments and blank lines included. While it is not possible to draw deep conclusions from such a simple analysis, we present rough size figures. It is clear that Linux comes out with the least lines of code. Also, by definition, Linux implements the full set of FUSE features. Hence, either other systems have more complex kernel file system interfaces, are implemented in a more loose style of coding, or the implementation of the FUSE kernel module on a non-Linux platform is a more complex operation than implementing it on Linux.

Table 2. *ReFUSE* component sizes

Component	# of lines
puffs vfs	6606
libpuffs	6212
librefuse	2102

5. *ReFUSE*

Whilst *puffs* provides the equivalent functionality of FUSE, it is an interface which is specific to NetBSD and there are just a handful of file systems written against it. FUSE, on the other hand, has a wealth of file systems readily available. To support those file systems on NetBSD with *puffs* all that is necessary is a thin emulation library in userspace on top of libpuffs. This emulate-in-userspace approach provides both a cleaner kernel implementation for the userspace file systems framework, and a BSD-licensed implementation.

ReFUSE supports only the FUSE regular API and not the lowlevel API. Work to support that is underway, but is unfinished at the time of writing.

The implementation and API considerations are further described in this chapter. Unless otherwise mentioned, this section discusses the regular FUSE API, i.e. not the low level version.

A similar Table as was presented for other FUSE implementations in Table 1 is presented for *puffs* + *ReFUSE* in Table 2. The *puffs* portions of the Table are presented only for general interest, and cannot be used for any comparisons as they are not functionally equivalent with FUSE. However, our *ReFUSE* userspace library is less code than what other systems have had to implement in the kernel for FUSE functionality. Of course, a full comparison of the effort can be made only when the FUSE API is fully supported.

5.1. Compatibility

ReFUSE provides an interface which is source code compatible with the FUSE API. This means that existing FUSE file systems can be compiled and linked against *ReFUSE* as-is. While currently *ReFUSE* does not need to support older ABI ver-

sions[2], ABI compatibility will be tracked once the *ReFUSE* library has been part of a NetBSD release. API compatibility will naturally follow the lead of FUSE.

As explained in Chapter 3.2, many revisions of the FUSE interface already exist. Since we wish to support all file systems regardless of the FUSE version they are written against, we also implement support for older versions in the translation layers. This includes making a runtime decision about calling for instance the older `getdir()` FUSE interface function or the new replacement `readdir()`.

The default FUSE version supported by *ReFUSE* is 2.6. It is possible to compile file systems based on the earlier 2.4 interface, although there are not many of those file systems in existence at the present time.

In general working, we have found that there are no common changes which need to be made to FUSE file systems to allow them to work with librefuse. Earlier versions of the *ReFUSE* header files did not include the options file in the same way, but that was corrected very early on in development.

5.2. *ReFUSE* Implementation

ReFUSE is implemented as a translation layer between libpuffs and a FUSE file server, recall Figure 1. It registers itself as a normal file system to *puffs*, while the file server in turn registers itself to the (Re)FUSE interface. Upon receiving a request, *ReFUSE* translates it from the *puffs* format to the FUSE format, calls the appropriate FUSE interfaces and changes the returned data to a format expected by *puffs*. Two examples follow. See also Appendix B and Appendix C to view the interfaces.

- **mkdir**: *ReFUSE* receives the directory, path component information and attributes from *puffs*. Using the full pathname constructed by libpuffs, it calls the FUSE file system's `mkdir()` method. If the call is successful, *ReFUSE* creates an internal node for its bookkeeping, which corresponds to the newly created directory and returns that node as the cookie to *puffs* and the kernel to be used for future references to the newly created directory.

- **setattr**: *ReFUSE* receives attributes and the node to be changed from *puffs*. FUSE does not contain a single `setattr()` interface but rather splits it up into `chmod()`, `chown()`, `utimens()` / `utime()` and `truncate()` / `ftruncate()`. The relevant FUSE interface functions are called depending on the set of attributes received in the call. The pathname required for the FUSE call is built by libpuffs into the *ReFUSE* node.

[2] There trivially are no programs compiled against *ReFUSE* with an older FUSE ABI since *ReFUSE* has never provided that ABI.

5.3. Differences Between *puffs* and FUSE

Here we list some minor differences between *puffs* and FUSE which had to be taken into account when authoring *ReFUSE*.

1. Node attributes are represented in the POSIX API by *struct stat*. FUSE also uses this structure for the same purpose. In the BSD kernel the structure for this purpose is called *struct vattr* and this is what *puffs* uses. Conversion is mostly a straightforward operation and is actually handled by a routine which was already provided by libpuffs.

2. The error return values in Linux are returned as negative integers. In BSD, these values are positive integers. These values need to be manipulatedin librefuse after processing any FUSE callback function.

3. *puffs* always passes full context to all callback functions in the form of both the complete arguments from the kernel and a *struct puffs_cc* context cookie pointer. FUSE relies on a routine called `fuse_get_context`(void) and thread local storage to provide some of this information. We must set up the information before calling the callback so that it is available if the callback calls `fuse_get_context()`.

4. The directory read in *puffs* and FUSE is buffered in a different fashion. In *puffs* and any BSD kernel file system, multiple calls to the `readdir()` routine are issued with different offsets. In FUSE, there is no offset parameter and only one call is made and the whole directory is produced in one go. *ReFUSE* must internally buffer this list to satisfy *puffs* `readdir()` calls starting from nonzero offsets.

5.4. Development Technique

The development of librefuse was done by first writing a skeleton implementation, which implemented only the bare minimum to get the "hello world" file system running. After that, various file systems were tested against this interface. At first

most file systems refused to compile and did not work because of missing interface support. But as more and more file systems were tested, some started compiling against librefuse without any modification to librefuse. Some even worked "out of the box". This approach, as opposed to aiming to write a complete implementation in the first try, was selected because some of the FUSE interfaces were not obvious without example file systems to use them. Also, it allowed people tracking NetBSD-current to use some known-to-work file systems in pkgsrc, such as ntfs-3g, while librefuse was still under development. In retrospect, it was the right development technique.

5.5. Restrictions

The current restrictions in using FUSE-based file systems with librefuse are:

- File systems using the FUSE lowlevel API are not supported. Support for that is in progress.
- *puffs* is logically a single-threaded entity, although its continuation framework provides multitasking support. Some of the FUSE operations currently require multithreading to be handled properly. For example the `fuse_unmount()` call can be called from file server context. However, this is the same context as the context for handling the kernel events. Therefore, simply calling `unmount()` from *ReFUSE* will deadlock the process. There is currently no clean solution for these kinds of problems in *ReFUSE*, but it does its best not to deadlock.

  Typically single-threadedness means that applications will be less parallel, not that they will fail function properly. We could introduce threading to *ReFUSE*, but as extending the worker models provided by *puffs* is planned, we will rather solve the problem at that level.

Aside from those restrictions, FUSE file systems will work properly on NetBSD.

5.6. Solutions

By default, extended attributes are not supported in NetBSD's ffs. For support, a kernel compilation option needs to be enabled. It is arguable how best to add extended attribute support to ffs - one method would be to write a librefuse-based file system to do this. This shows how powerful the FUSE mechanism is - that a FUSE layer can be added to any existing file system to enable extended attributes to be used on that lower-level filesystem. Extending this idea still further allows us to think of Mac OS resource fork support on existing file systems, or long-name support for filesystems which currently use 8.3 file name formats.

5.7. Integration with pkgsrc

The third-party packaging system for NetBSD, pkgsrc [10], uses a number of features to abstract packages from the operating system instances. This dramatically increases portability of third-party software.

At the time of writing, the following file systems are known to work on NetBSD and are provided in pkgsrc/filesystems. This is not to say others will not, merely that we have not gotten around to adding them to pkgsrc yet.

- fuse-afpfs-ng
- fuse-archivemount
- fuse-cddfs
- fuse-cryptofs
- fuse-curlftpfs
- fuse-encfs
- fuse-gmailfs
- fuse-gphotofs
- fuse-httpfs
- fuse-loggedfs
- fuse-lzofs
- fuse-ntfs-3g
- fuse-obexfs
- fuse-pod
- fuse-unionfs
- fuse-wdfs
- fuse-wikipediafs

To achieve platform abstraction, pkgsrc uses a buildlink file to control building and linking of the third party package. One example is as follows: NetBSD has a *librefuse.so*, and no *libfuse.so*. In addition, NetBSD requires that the resulting programs are linked with *libpuffs.so*. This is actually already done in the build stage of *librefuse.so*, but it means that *libpuffs.so* must be present at run time.

To enable pkgsrc entries to use either *ReFUSE* on NetBSD, and FUSE on other operating systems[3], a line including the proper buildlink file is added to the Makefile for the pkgsrc entry. An example of the last two lines in a proper FUSE pkgsrc Makefile are presented in Figure 4.

[3] pkgsrc runs on more than 13 platforms, where NetBSD is considered one platform.

Figure 4. pkgsrc buildlink example

```
.include "../../mk/fuse.buildlink3.mk"
.include "../../mk/bsd.pkg.mk"
```

Any package which uses either FUSE or *ReFUSE* simply needs the fuse.buildlink3.mk file included in the Makefile for their own pkgsrc entry.

6. The Virtual Directory Interface

It was found, whilst writing various FUSE based file systems, that a number of them share a common need - that of being able to provide a consistent means of manipulating virtual information to present it as file system directories and directory entries to the system calls. The virtual directory interface was developed to do this, in a (key, value) pair architecture, and is used by a number of FUSE based file systems, including icfs (to cache the lower case name of the directory entry), id3fs (to cache the name and target file names of the virtual mp3-based hierarchy), and in dmesgfs (having parsed the dmesg information to determine parent and child nodes in the device information). In a sense, a virtual directory interface can be thought to be a lighter weight, in-code, dynamic version of a file system generating language [23]. The virtual directory interface is presented for reference in Appendix A.

The traversing interface (openvirtdir, readvirtdir, closevirtdir) was designed to mirror that of the directory(3) routines, using an opaque VIRTDIR cookie in an analogous way to the corresponding DIR cookie. The other routines manipulate entries in the virtual directory database. These routines allow a file system writer to build up a list of file names, and associate either a structure or another string with them (labelled "value" in the routines), and to locate (by ordered search of names, or by sequential search of values) virtual directory entries. Traversal is then by means of an openvirtdir(), readvirtdir(), closevirtdir() sequence of calls.

These routines can be found in the NetBSD CVS repository in *src/share/examples/refuse/virtdir*.

Since some file systems may contain virtual entries, standardising on a set of routines makes sense for writers of userland file systems. For examples of their use, please see the id3fs file system in *src/share/examples/refuse/id3fs*, and the icfs file system in *src/share/examples/refuse/icfs*, both in the NetBSD CVS repository.

7. Experiences with *ReFUSE*

7.1. Porting FUSE File Systems

There are a large number of FUSE-based file systems, almost all of them written for Linux. In porting these file systems from Linux to NetBSD, it is apparent that there are a lot of assumptions made about the target operating system - "all the world's a Linux". Whilst this is understandable, it can prove annoying at times. This has nothing to do with FUSE per se and regular Unix software porting skills [18] help here.

Some examples are the use of internal timespec values in the stat structure, the implicit assumption that certain functions will be available, definitions present only upon Linux systems, and the assumed presence of certain header files.

7.2. Writing a new File System

In addition to using *ReFUSE* to make FUSE based file systems on NetBSD, several file systems have been written against the FUSE interface with *ReFUSE* as the starting point.

The typical way of writing a new file system is to use an existing FUSE based file system as a template. To that end, the examples provided in the NetBSD CVS repository in *src/share/examples/refuse* are useful:

- **dbfs**: this file system allows an existing Berkeley B-tree db-based file to be mounted as a file system. The key value is used as the file system pathname, and the value can either be represented as the target of a symbolic link, or as the contents of virtual files.

- **dmesgfs**: uses the devices found and announced via dmesg to form a virtual directory hierarchy. Buses show up as virtual directories, and devices attached to the bus are represented as directory entries within the bus "directory".

- **fanoutfs**: a fanout file system is a layered file system that acts like a "fan" and can access data from multiple file systems below [22]. This functionality is implemented by our fanoutfs in userspace. Another example of a fanout file system is the BSD union file system [20]

  In fanoutfs, the "first" directory hierarchy is a writable directory hierarchy, and all the other directory hierarchies (there may, of course, be no others) are treated as read-only hierarchies. If an "existing" file needs to be opened for writing, its

existing directory hierarchy is first created in the writable hierarchy, and the file copied to there before being opened for write. A useful example for this is an easy way to enable package views [14], and mirrors the way that Microsoft Windows installs software in a ”transactional” manner.

- **icfs**: this file system allows an existing directory hierarchy to be mounted and accessed with the path names presented and manipulated in lower case. Newly-created files are made in a case-retentive manner. There is also an analogous *puffs*-based icfs, for comparison purposes.
- **id3fs**: this file system presents an interface similar to some personal mp3 players, whereby an existing mp3 file hierarchy can be accessed by Artist and Genre, as well as by existing pathname.
- **pcifs**: this file system uses pcictl to present a read-only list of devices attached to the PCI buses in a system in a virtual hierarchy

While written against the FUSE interface, some of these file systems such as dmesgfs depend on NetBSD for their backend. Others are fully independent of the host they are run on and can be run on Linux.

7.3. Example by comparison: icfs

As a method for comparing the FUSE and *puffs* interfaces, a file system layer translating all names to lowercase is discussed. The same functionality was implemented both on top of FUSE using the *ReFUSE* virtual directory interface and on top of *puffs* using the *puffs* pathname framework.

7.3.1 *puffs* icfs

The *puffs* version for achieving a case-insensitising layer uses the *puffs* pathname framework. It defines only the *rename* callback operation and points all the other operations to the *puffs* null layer implementation[4]. The readdir operation calls the null layer also, but before returning, it converts all read directory entries to lowercase.

Additionally, *puffs* icfs registers two callbacks to the *puffs* pathname framework: a path comparison routine and a path transformation routine. The comparison routine is used by the framework to determine if two paths are equal and this case the comparison routine does `strcasecmp()` instead of the regular `strcmp()`.

The path transformation routine transforms the input pathname into a format used by the file system. In this case the routine scans the underlying mount's directory for pathnames which have a case-insentive match with the pathname under translation. If one is found, it is stored in the node. Further operations will then use the correct underlying pathname instead of the case-insentive one. For example, if ”muusi” from the path ”/nakit/ja/muusi” is under translation, the actual filename returned by the translation routine might be ”/NaKiT/Ja/MuuSi”. This properly-capitalised path is then used for all the operations on the node.

7.3.2 FUSE icfs

The FUSE API does not provide functionality similar to the path translation API of *puffs*. Instead, the FUSE version of icfs has to register a full set of file system callbacks and manipulate paths in each of those entry points.

At startup the FUSE icfs builds a directory hierarchy mapping the proper pathnames to lowercase pathnames. Whenever creating or removing a file in icfs, this mapping is updated.

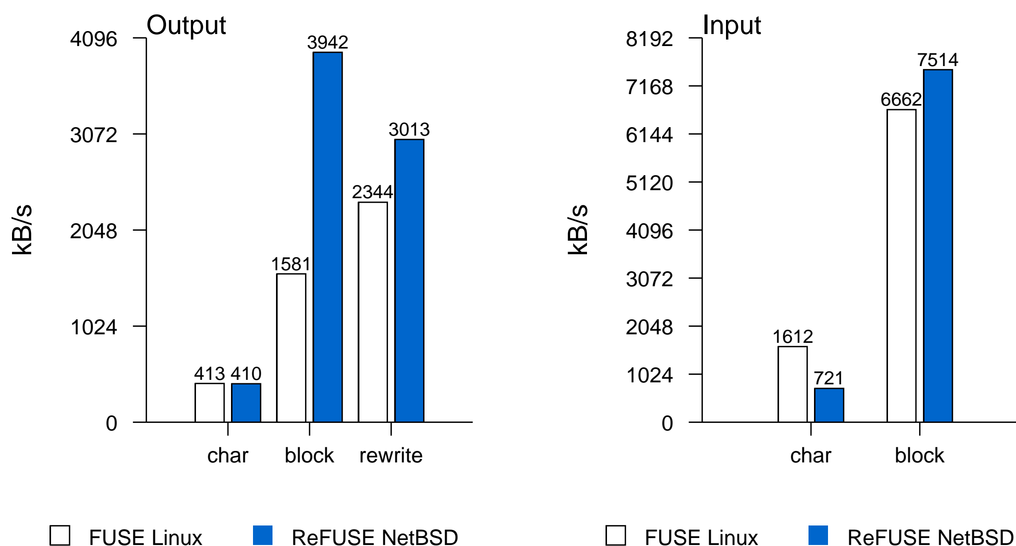
7.3.3 Comparison

There are two main differences in the approaches of the two implementations:

1. The *puffs* version uses a local, per-directory storage for the mappings from lowercase to pathname. This can conveniently be done, because there is the concept of a data structure pointer in the *puffs* interface. The FUSE version, on the other hand, uses a global list, namely the virtual directory database.
2. The *puffs* version does processing in the pathname translation routines as part of the framework and needs to implement only `readdir()` specially. Since the FUSE interface lacks this kind of pathname hook, it must do processing in each individual operation.

These differences result in the fact that the *puffs* version is about a third of the code size of the FUSE version.

[4]The null layer works just like nullfs in the kernel. The only exception is that it is pathname-based instead of vnode-based, since it does the actual operations through system calls, which take pathnames as arguments.



7.4. Performance

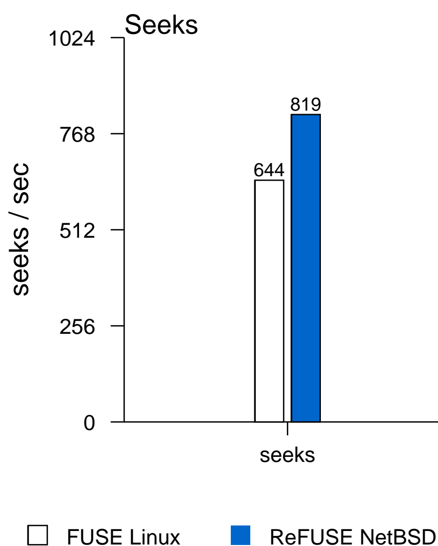
In the course of this work, some very rough performance measurements were made. These measurements were performed inside qemu [13]. The Linux qemu environment was a Knoppix bootable CD, while the NetBSD one was a regular installation to a disk image. The test was running `bonnie++` with a 128MB file while qemu was started with 64MB of RAM. The FUSE file system used for testing was ntfs-3g.

There are too many variables in the equation to draw any conclusive results about the performance, as the underlying operating systems have great effect on the results. However, it can clearly be seen that our userspace emulation layer of the FUSE interface has nothing to be ashamed of in terms of performance.

The standard bonnie results for sequential and random access performance are presented in Figure 5 and Figure 6, respectively.

7.5. Stability

The stability of *ReFUSE* has been found to be excellent in approximately the six months it has been in use in NetBSD. We know of a number of people who use *ReFUSE* in daily real-world operation and have never complained about a crash or lost data. Naturally we use it ourselves also.



8. Conclusions and Future Work

The *ReFUSE* functionality presented in this paper builds on the existing *puffs* functionality to provide source code compatibility for FUSE-based file systems. Some observations were made on the levels at which the *puffs* and FUSE interfaces were architected. FUSE-based file systems were examined, and some examples were illustrated. Work continues on *ReFUSE* to provide compatibility with file systems implemented using the FUSE low-level interface. Both *puffs* and *ReFUSE* are being actively maintained and developed in the NetBSD repository.

The excellent thing about FUSE is that the interface is a universal de-facto standard for writing userspace file systems. The interface is also versioned and will support backward compatibility for the foreseeable future. This means that programs once written and compiled against FUSE will continue to function even though new versions of the interface might bring new features. This allows the development of *puffs* to take a more internal route and track the operating system VFS more closely without worrying too much about interface compatibility issues.

The development strategy for *ReFUSE* is considered, in retrospect, to have been optimal. Further work will continue over the next months to refine further the functionality; in particular, the FUSE low-level functionality will be implemented.

Rough performance figures are similar, when measured between FUSE and *ReFUSE*. However, more work needs to be done in this area, so that true comparisons can be made.

There are also plans to extend the scope of the *puffs* framework to support devices as well as file systems. Similar work was done for an earlier version of Linux in FUSD - a Linux Framework for User-Space Devices [1].

Use and Availability

ReFUSE is part of the current NetBSD development branch. It will be included in the NetBSD 5.0 release.

For more information on using and developing *puffs* and *ReFUSE*, please see the NetBSD website: *http://www.NetBSD.org/docs/puffs/*

Acknowledgements

Miklós Szeredi, the author of FUSE, provided help in reviewing this paper. Part of this work was funded by the Finnish Cultural Foundation.

References

[1] FUSD - a Linux Framework for User-Space Devices. *http://www.circlemud.org/ jelson/software/fusd/*.

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

[3] FUSE FAQ. Which method is called on the close() system call?

[4] Fuse for FreeBSD. *http://fuse4bsd.creo.hu/*.

[5] FUSEPod. *http://fusepod.sourceforge.net/*.

[6] Gmail file system. *http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html*.

[7] macfuse: A FUSE-Compliant File System Implementation Mechanism for Mac OS X. *http://code.google.com/p/macfuse/*.

[8] OpenSolaris Project: Fuse on Solaris. *http://www.opensolaris.org/os/project/fuse/*.

[9] The NetBSD Project. *http://www.NetBSD.org*.

[10] The pkgsrc Guide. *http://www.NetBSD.org/docs/pkgsrc*.

[11] thread: ”zfs-fuse 0.4.0 beta1 released”. *Google groups: ZFS-FUSE*.

[12] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *USENIX*, pages 77–90, 1997.

[13] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX*, pages 41–46, 2005.

[14] A. Crooks. Package views: a more flexible infrastructure for third-party software. In *2nd European BSD Conference*, 2002.

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

[16] A. Kantee. Using puffs for implementing client-server distributed file systems. Technical Report TKK-TKO-B157, Helsinki University of Technology, 2007.

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

[18] G. Lehey. *Porting UNIX software: from download to debug*. O'Reilly & Associates, Inc., 1995.

[19] NetBSD Library Functions Manual. *puffs – Pass-to-Userspace Framework File System development interface*, July 2007.

[20] J.-S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-lite. In *USENIX*, 1995.

[21] B. Welch and J. Ousterhout. Pseudo Devices: User-Level Extensions to the Sprite File System. In *USENIX Summer*, pages 37–49, 1988.

[22] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[23] E. Zadok. *FiST: A System for Stackable File System Code Generation*. PhD thesis, Computer Science Department, Columbia University, 2001.

Appendix A *ReFUSE* ***virtdir*** Interface

```
int     virtdir_init(virtdir_t *tp, const char *rootdir, struct stat *d,
                     struct stat *f, struct stat *l);
void    virtdir_drop(virtdir_t *tp);

char    *virtdir_rootdir(virtdir_t *tp);

int     virtdir_add(virtdir_t *tp, const char *name, size_t size,
                    uint8_t type, const char *tgt, size_t tgtlen);
int     virtdir_del(virtdir_t *tp, const char *name, size_t size);

virt_dirent_t   *virtdir_find(virtdir_t *tp, const char *name,
                              size_t namelen);
virt_dirent_t   *virtdir_find_tgt(virtdir_t *tp, const char *tgt,
                                  size_t tgtlen);

VIRTDIR         *openvirtdir(virtdir_t *tp, const char *d);
virt_dirent_t   *readvirtdir(VIRTDIR *dirp);
void            closevirtdir(VIRTDIR *dirp);

int             virtdir_offset(virtdir_t *tp, virt_dirent_t *dp);
```

Appendix B *puffs* Callback Interface

fs type operations:

```
int     puffs_fs_statvfs(struct puffs_cc *pcc, struct statvfs *sbp,
                         const struct puffs_cid *pcid);
int     puffs_fs_sync(struct puffs_cc *pcc, int waitfor,
                      const struct puffs_cred *pcr,
                      const struct puffs_cid *pcid);
int     puffs_fs_fhtonode(struct puffs_cc *pcc, void *fid, size_t fidsize,
                          struct puffs_newinfo *pni);
int     puffs_fs_nodetofh(struct puffs_cc *pcc, void *cookie, void *fid,
                          size_t *fidsize);
void    puffs_fs_suspend(struct puffs_cc *pcc, int status);
int     puffs_fs_unmount(struct puffs_cc *pcc, int flags,
                         const struct puffs_cid *pcid);
```

node type operations:

```
int     puffs_node_lookup(struct puffs_cc *pcc, void *opc,
                          struct puffs_newinfo *pni,
                          const struct puffs_cn *pcn);
int     puffs_node_create(struct puffs_cc *pcc, void *opc,
                          struct puffs_newinfo *pni,
                          const struct puffs_cn *pcn,
                          const struct vattr *vap);
int     puffs_node_mknod(struct puffs_cc *pcc, void *opc,
                         struct puffs_newinfo *pni,
                         const struct puffs_cn *pcn,
                         const struct vattr *vap);
int     puffs_node_open(struct puffs_cc *pcc, void *opc, int mode,
                        const struct puffs_cred *pcr,
                        const struct puffs_cid *pcid);
int     puffs_node_close(struct puffs_cc *pcc, void *opc, int flags,
                         const struct puffs_cred *pcr,
                         const struct puffs_cid *pcid);
int     puffs_node_access(struct puffs_cc *pcc, void *opc, int mode,
                          const struct puffs_cred *pcr,
                          const struct puffs_cid *pcid);
int     puffs_node_getattr(struct puffs_cc *pcc, void *opc, struct vattr *vap,
                           const struct puffs_cred *pcr,
                           const struct puffs_cid *pcid);
int     puffs_node_setattr(struct puffs_cc *pcc, void *opc,
                           const struct vattr *vap,
                           const struct puffs_cred *pcr,
                           const struct puffs_cid *pcid);
int     puffs_node_poll(struct puffs_cc *pcc, void *opc, int *events,
                        const struct puffs_cid *pcid);
int     puffs_node_mmap(struct puffs_cc *pcc, void *opc, int flags,
                        const struct puffs_cred *pcr,
                        const struct puffs_cid *pcid);
```

```
int     puffs_node_fsync(struct puffs_cc *pcc, void *opc,
                         const struct puffs_cred *pcr, int flags,
                         off_t offlo, off_t offhi,
                         const struct puffs_cid *pcid);
int     puffs_node_seek(struct puffs_cc *pcc, void *opc, off_t oldoff,
                        off_t newoff, const struct puffs_cred *pcr);
int     puffs_node_remove(struct puffs_cc *pcc, void *opc, void *targ,
                          const struct puffs_cn *pcn);
int     puffs_node_link(struct puffs_cc *pcc, void *opc, void *targ,
                        const struct puffs_cn *pcn);
int     puffs_node_rename(struct puffs_cc *pcc, void *opc, void *src,
                          const struct puffs_cn *pcn_src, void *targ_dir,
                          void *targ, const struct puffs_cn *pcn_targ);
int     puffs_node_mkdir(struct puffs_cc *pcc, void *opc,
                         struct puffs_newinfo *pni,
                         const struct puffs_cn *pcn,
                         const struct vattr *vap);
int     puffs_node_rmdir(struct puffs_cc *pcc, void *opc, void *targ,
                         const struct puffs_cn *pcn);
int     puffs_node_readdir(struct puffs_cc *pcc, void *opc, struct dirent *dent,
                           off_t *readoff, size_t *reslen,
                           const struct puffs_cred *pcr,
                           int *eofflag, off_t *cookies, size_t *ncookies);
int     puffs_node_symlink(struct puffs_cc *pcc, void *opc,
                           struct puffs_newinfo *pni,
                           const struct puffs_cn *pcn_src,
                           const struct vattr *vap, const char *link_target);
int     puffs_node_readlink(struct puffs_cc *pcc, void *opc,
                            const struct puffs_cred *pcr,
                            char *link, size_t *linklen);
int     puffs_node_read(struct puffs_cc *pcc, void *opc, uint8_t *buf,
                        off_t offset, size_t *resid,
                        const struct puffs_cred *pcr, int ioflag);
int     puffs_node_write(struct puffs_cc *pcc, void *opc, uint8_t *buf,
                         off_t offset, size_t *resid,
                         const struct puffs_cred *pcr, int ioflag);
int     puffs_node_print(struct puffs_cc *pcc, void *opc);
int     puffs_node_reclaim(struct puffs_cc *pcc, void *opc,
                           const struct puffs_cid *pcid);
int     puffs_node_inactive(struct puffs_cc *pcc, void *opc,
                            const struct puffs_cid *pcid);
```

Appendix C   FUSE Regular Interface

```
int (*getattr) (const char *, struct stat *);
int (*readlink) (const char *, char *, size_t);
int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
int (*mknod) (const char *, mode_t, dev_t);
int (*mkdir) (const char *, mode_t);
int (*unlink) (const char *);
int (*rmdir) (const char *);
int (*symlink) (const char *, const char *);
int (*rename) (const char *, const char *);
int (*link) (const char *, const char *);
int (*chmod) (const char *, mode_t);
int (*chown) (const char *, uid_t, gid_t);
int (*truncate) (const char *, off_t);
int (*utime) (const char *, struct utimbuf *);
int (*open) (const char *, struct fuse_file_info *);
int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
int (*write) (const char *, const char *, size_t, off_t,
      struct fuse_file_info *);
int (*statfs) (const char *, struct statvfs *);
int (*flush) (const char *, struct fuse_file_info *);
int (*release) (const char *, struct fuse_file_info *);
int (*fsync) (const char *, int, struct fuse_file_info *);
int (*setxattr) (const char *, const char *, const char *, size_t, int);
int (*getxattr) (const char *, const char *, char *, size_t);
int (*listxattr) (const char *, char *, size_t);
int (*removexattr) (const char *, const char *);
int (*opendir) (const char *, struct fuse_file_info *);
int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t,
        struct fuse_file_info *);
int (*releasedir) (const char *, struct fuse_file_info *);
int (*fsyncdir) (const char *, int, struct fuse_file_info *);
void *(*init) (struct fuse_conn_info *conn);
void (*destroy) (void *);
int (*access) (const char *, int);
int (*create) (const char *, mode_t, struct fuse_file_info *);
int (*ftruncate) (const char *, off_t, struct fuse_file_info *);
int (*fgetattr) (const char *, struct stat *, struct fuse_file_info *);
int (*lock) (const char *, struct fuse_file_info *, int cmd,
     struct flock *);
int (*utimens) (const char *, const struct timespec tv[2]);
int (*bmap) (const char *, size_t blocksize, uint64_t *idx);
```

Appendix D FUSE Low Level Interface

```
void (*init) (void *userdata, struct fuse_conn_info *conn);
void (*destroy) (void *userdata);
void (*lookup) (fuse_req_t req, fuse_ino_t parent, const char *name);
void (*forget) (fuse_req_t req, fuse_ino_t ino, unsigned long nlookup);
void (*getattr) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi);
void (*setattr) (fuse_req_t req, fuse_ino_t ino, struct stat *attr,
        int to_set, struct fuse_file_info *fi);
void (*readlink) (fuse_req_t req, fuse_ino_t ino);
void (*mknod) (fuse_req_t req, fuse_ino_t parent, const char *name,
       mode_t mode, dev_t rdev);
void (*mkdir) (fuse_req_t req, fuse_ino_t parent, const char *name,
       mode_t mode);
void (*unlink) (fuse_req_t req, fuse_ino_t parent, const char *name);
void (*rmdir) (fuse_req_t req, fuse_ino_t parent, const char *name);
void (*symlink) (fuse_req_t req, const char *link, fuse_ino_t parent,
         const char *name);
void (*rename) (fuse_req_t req, fuse_ino_t parent, const char *name,
        fuse_ino_t newparent, const char *newname);
void (*link) (fuse_req_t req, fuse_ino_t ino, fuse_ino_t newparent,
      const char *newname);
void (*open) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi);
void (*read) (fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
      struct fuse_file_info *fi);
void (*write) (fuse_req_t req, fuse_ino_t ino, const char *buf,
       size_t size, off_t off, struct fuse_file_info *fi);
void (*flush) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi);
void (*release) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi);
void (*fsync) (fuse_req_t req, fuse_ino_t ino, int datasync,
       struct fuse_file_info *fi);
void (*opendir) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi);
void (*readdir) (fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
         struct fuse_file_info *fi);
void (*releasedir) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi);
void (*fsyncdir) (fuse_req_t req, fuse_ino_t ino, int datasync,
          struct fuse_file_info *fi);
void (*statfs) (fuse_req_t req, fuse_ino_t ino);
void (*setxattr) (fuse_req_t req, fuse_ino_t ino, const char *name,
          const char *value, size_t size, int flags);
void (*getxattr) (fuse_req_t req, fuse_ino_t ino, const char *name,
          size_t size);
void (*listxattr) (fuse_req_t req, fuse_ino_t ino, size_t size);
void (*removexattr) (fuse_req_t req, fuse_ino_t ino, const char *name);
void (*access) (fuse_req_t req, fuse_ino_t ino, int mask);
void (*create) (fuse_req_t req, fuse_ino_t parent, const char *name,
        mode_t mode, struct fuse_file_info *fi);
void (*getlk) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi,
       struct flock *lock);
void (*setlk) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi,
       struct flock *lock, int sleep);
void (*bmap) (fuse_req_t req, fuse_ino_t ino, size_t blocksize,
             uint64_t idx);
```