A General-Purpose Synthetic Filesystem

Stephen Kell

Computer Science Tripos, Part II

Christ's College

16th May 2005

Proforma

Name:	Stephen Kell
College:	Christ's College
Project Title:	A General-Purpose Synthetic Filesystem
Examination:	Computer Science Tripos, Part II, June 2005
Word Count:	approx 11900
Project Originator:	Stephen Kell
Supervisor:	Douglas Santry

Original Aims of the Project

To build a synthetic filesystem exposing a graph of Java objects, with support for method invocation, field accesses, and automatic update of the filesystem to reflect changes in the graph. In so doing, to identify those aspects of a traditional filesystem API that are unsuited to general-purpose implementations. Emphasis on a proof-of-concept rather than high-performance implementation.

Work Completed

Substantially the above. Some less important features were left unimplemented for reasons of time. A networkable, cross-platform means of implementing method call was devised and implemented, though currently only with slow performance.

Special Difficulties

None.

Declaration of Originality

I, Stephen Kell of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

Traditionally, filesystems are thought of as systems for the naming, storage and retrieval of logical collections of data (files) held on a storage device (typically a magnetic disk). However, as we will see, the concept of a filesystem has been applied in many different ways, and not all of these concern storage. In fact, the facilities provided by filesystems are remarkably general: naming, access control, some concurrency control, and the reading and writing of data.

In this chapter we will explain and motivate the idea of a "general-purpose synthetic filesystem".

1.1 The Filesystem as an Abstract Data Type

The Unix operating system popularised many advancements in filesystem design, including the arbitrary-depth directory tree structure and linking. It also pioneered the use of the filesystem's programming interface for performing what would traditionally have been seen as distinct classes of operation.

This latter technique, commonly stated as the "everything is a file" principle, has proved to be a powerful architectural feature, as it permits the use of a single programmatic interface to manipulate diverse entities including files, hardware devices and channels of inter-process communication (IPC) such as pipes and sockets.[1]

This diverse use of the filesystem interface, and corresponding diverse implementations of this interface, lead naturally to the idea that a filesystem interface represents an abstract data type (ADT). Accordingly, most modern operating

[1] These entities may or may not be addressable through the filesystem, but at the very least can be read or written using file descriptors. For example, devices are traditionally named in the `/dev` hierarchy, while pipes originally could only be created using a special system call.

systems are architected with a single internal interface, known as the Virtual File System or VFS interface, upon which all filesystems are implemented. For the remainder of this document we will primarily consider the Unix filesystem interface, or specifically its POSIX standardisation [1].

1.2 Synthetic Filesystems

We may distinguish between filesystems whose purpose is to provide storage, including traditional disk-based filesystems and also network file systems such as NFS [2] and Coda [3], and those whose purpose is to provide an interface into some other domain where the data would not conventionally be thought of as files. We will use the term "synthetic filesystem" to refer to any filesystem backed by something other than a persistent storage medium.[2]

The most well-known of this latter kind of filesystem is the `/proc` filesystem, which exposes process images and metadata as files. Originating [5] as a replacement for the `ptrace()` system call in Unix, `/proc` was incorporated [6] into System V Release 4 and has subsequently been extended by various modern Unices [7]. It is useful as a debugging aid, and as a platform enabling tools such as `ps` to be implemented without directly accessing kernel data structures.

Other synthetic filesystems systems include FreeBSD's `devfs`, a purely synthetic filesystem device filesystem[3], and NetFS [8], which exposes network interface configurations. In a similar vein, Plan 9 [9] generalises the filesystem to an abstract per-process network-transparent namespace used as the basis of nearly all IPC.

1.3 Special Purpose *versus* General Purpose

We observe the following advantages of exposing data through a synthetic filesystem (as opposed to some other IPC scheme).

- Clients need not include any special-purpose logic for primitive accesses (by which we mean operations corresponding intuitively to the operations of the filesystem interface, such as `open()`, `close()`, `read()`, `write()`, `stat()` and so on).

[2] This definition does not always provide a clear division between synthetic and non-synthetic, for example with the `ftp2nfs` system [4], but will be sufficient our purposes.

[3] This contrasts with traditional Unix device files, which are recorded on disk using a special type of inode.

- The filesystem's existing access control semantics can be used.
- Performance gains may be had from the simpler client interface. (Train et al [8] claim that the `ps` tool ran four times faster when coded against the `/proc` filesystem rather than kernel data structures, owing to this simplified interface.)

For many of these, there are also corresponding disadvantages.

- The logic for accesses must still be provided by the server, in its VFS implementation.
- Logic for more complex operations than primitive accesses must still be provided on the client side.
- Existing access control, accounting metadata (e.g. timestamps) or the filesystem's intrinsic structure (i.e. the directed acyclic graph) may be inadequate or redundant for the purposes of the target data structure.
- Conversion of data to and from byte streams for transfer across the filesystem interface may degrade performance.

The first of these problems characterises what we will call a *special-purpose* synthetic filesystem. The logic which maps filesystem operations onto the underlying data structures is specific to the filesystem.

This naturally raises the question of whether, for a given run-time representation of general data structures, it would be possible to implement a filesystem which provides a useful set of operations without knowledge of the intended application – that is, a *general-purpose* synthetic filesystem.

The key advantage of a general-purpose implementation is its reusability: it immediately brings the benefits of filesystem-based IPC to any host application. It would provide a convenient interface to the internals of running services and interactive applications, and could be used for testing, in-situ debugging, obtaining status information or scripting complex interactions.

1.4 Problems with General Purpose

Several problems with the general-purpose approach are apparent. The implementation has no knowledge of the data structure's semantics, except whatever typing information is available at run-time. Accordingly, more responsibility for protecting consistency resides with the client. There is a similar problem with concurrency control. Possible approaches to the type system are:

- Extend a compiler to generate additional code to serve nominated data structures as filesystems, using compile-time type information.
- Describe to the filesystem implementation at run-time how to interpret the data at particular memory addresses.
- Require the data structures to contain run-time type information which can be consumed by the filesystem implementation.
- Serve memory ranges typelessly, and require the client process to be aware of the intended semantics.

Modifying a compiler, though perhaps efficient, would be cumbersome and unportable. The typeless approach is overly trusting of the client, and in most cases incapable of providing a useful inteface to the client without requiring additional logic.

The second approach is taken by the RTA system [10], which exposes C data structures as a filesystem (and as a relational database). However, in the spirit of the general-purpose ideal, we would prefer to minimise the amount of code which must be added to an existing data structure implementation in order to expose it as a filesystem.

For this reason, we have chosen to use the Java runtime system (known as the Java Virtual Machine or JVM [11]), and to implement a filesystem which exposes a graph of Java objects.

1.5 Java Object Graphs

The use of the Java runtime brings several advantages. It provides strong type-safety guarantees, and has a simple object-based type system. Data structures consist of a connected graph of discrete objects. Each object contains of a series of fields, which are either primitively-typed (integers, Booleans, etc) or references to other objects. Arrays are considered a special type of object.

This data model maps conveniently to a filesystem structure. Intuitively, a set of objects can be represented as a set of directories, within which primitive fields are seen as files and object fields as symbolic links.[4]

The object-oriented design of the Java language, in particular its consistent use of encapsulation, also partially solves two problems mentioned above. Firstly,

[4]Here we are exploiting the fact that Unix filesystems permit cycles of symbolic links, while this it not allowed with hard links. This allows the effect of a general graph, while being implemented essentially as a DAG.

consistency of the underlying data structure can be protected by exposing only the public interface of an object, which by convention is at least moderately robust.

Secondly, the possibility of exposing more complex operations than simple reads and writes is raised by the existence of instance methods. Ideally, these methods would be exposed as executables within the filesystem, and could be invoked using the `exec()` system call. However, `exec()` is not part of the filesystem interface, so we can have no way of specifying its behaviour for a particular target file. In other words, the filesystem interface has no clean way of providing a general remote procedure call (RPC).[5] We will investigate various partial workarounds for this problem.

1.6 Adaptability of the Filesystem Interface

Before going further, it is worth questioning whether the filesystem interface is suitable for the diverse uses to which it has been put. Its storage-oriented design sits at odds with the strong typing of modern programming languages like Java, while we have already noted the potential inadequacy or oversufficiency of its metadata, the possibility of high marshalling overheads, and the lack of an RPC facility.

Unix has worked around these problems in various ways, such as the `ioctl()` system call, which is used to perform otherwise-unspecified operations on special files (such as device files or sockets) and extended attributes (a general set of name-value pairs attached to each file). Since these workarounds generally increase complexity and reduce the uniformity of the filesystem interface, there is a case for designing a new interface intended to cater for a more general set of operations and underlying implementations.

Therefore, as an experimental aspect of our work, we will note where the existing interface proves ill-suited to the highly general nature of our filesystem implementation, and draw conclusions from these when evaluating the system.

[5] Here, 'remote' means that the procedure call is performed behind the VFS interface; as always, it may or not actually be handled over a network.

Chapter 2

Preparation

Before starting on the implementation, preparations were made by deciding on feature requirements and success criteria, and on the development platform, tools and techniques to be employed.

2.1 Feature Requirements

The original project proposal mentions several feature requirements, which we restate here with some refinements. Those marked "optional" are not considered essential to the success of the project, but are desirable and will be implemented if time permits.

1. Representation of a Java object graph as a two-level directory hierarchy, with symbolic links connecting the objects.

2. Getting and setting of primitive fields, using the filesystem API such that tools like `cat` and `echo` can be used to read and write field values. We also choose that data is input and output in a textual (rather than binary) form, since this is the convention in Unix for IPC streams that may be used either by a human or by another process (for example, the output of many command-line Unix tools including `grep`, `ps` and others).

   Both textual and binary approaches have their own advantages and problems, as Schrock [7] discusses in the context of the `/proc` filesystem.

3. Following and setting of object reference fields (as symbolic links), such that links can be followed using `cd` and updated using `ln -s`[1]. Trying to

[1] Since we wish to silently replace an existing link rather than dereference it or create a new one, the appropriate command is `ln -snf`.

set the link to an invalid value (perhaps an object of the wrong type, or a path which doesn't map to an object directory) should return a relevant error code.

4. Method invocation on a local system, such that in an object's directory there exist files named after each of the object's instance methods, and that calling `exec()` on one of these should have the effect of executing the method. Again, we use textual rather than binary formatting, here for the parameters and return value.

   As discussed in the Introduction, it is not normally possible for a filesystem implementation to specify the behaviour of `exec()`, so we will instead take the approach of serving each method as an executable file which, when executed, will communicate with the filesystem by some other channel in order to invoke the method.

5. Exposition of array elements, using a further subdirectory within the object directory, whose elements are files (for primitive elements) or symbolic links (for object elements) named by their numerical index and behaving similarly to fields. Optionally, a similar indexing scheme should be applied to objects implementing the Map interface (in the `java.util` package).

6. (Optional.) Exposition of accessor methods (i.e. 'getters' and 'setters', as defined by the JavaBeans[12] standard), as files or symbolic links similarly to how fields are exposed.

7. (Optional.) A mechanism to create new objects. This is a useful piece of functionality which does not neatly correspond to a filesystem operation – `mkdir()` is close, too primitive: we require additional arguments (the object class and any constructor arguments) and create not only a directory but also its contents.

8. (Optional.) Use of the filesystem on a remote system, using NFS. This ought to be possible, but owing to the potential for unanticipated behaviour of the NFS client and server, it is not a requirement.

2.2 Success Criteria

Clearly, the primary success criterion must be to satisfy the feature requirements. In a project of experimental nature such as this one, it is difficult to assign quantitative targets, not least because no closely comparable systems exist. The

benefits of the system are mostly abstract, in that they relate to the versatility and convenience of the IPC system provided rather than any directly-measurable quantity.

As such, we have chosen specifically not to set any hard targets for performance or reliability. Our implementation will not be designed for high execution speed or low memory overhead, with the emphasis placed on producing a working proof-of-concept; however, in our evaluation we will nevertheless provide quantitative comparisons of speed and memory usage with existing systems, as a starting point for future work.

2.3 Base Platform

As we are considering only Unix filesystems, GNU/Linux was chosen as the target operating system. It is a familiar and widely-used Unix-like operating system.

There are two obvious approaches to implementing the filesystem: write a Linux kernel module, or use a user-space filesystem library. The latter approach was chosen, as it avoids the steep learning curve of writing kernel code, and increases the potential for portability of the implementation (should a port of the library be available also).

The `fuse` library [14] was chosen as the base for implementing the filesystem. It presents a simple C API, is widely used and was used for the implementation of the RTA [10] system, perhaps the system most similar to the one being developed here. It works using a kernel module serving a socket under the `/proc` filesystem to the user-space process. All development was done using version 1.9 of `fuse`.

The target Java runtime is version 1.4, which is more mature than the recent 1.5 revision while not lacking any important features.

2.4 Implementation Strategies

Since the aim of the project is to produce a working proof-of-concept rather than high-performance implementation, the overriding philosophy is to keep the code and architecture simple where possible.

Since we are using a C-based library to implement the filesystem, we use the Java Native Interface [15], a standard C interface to the JVM. This interface supports all operations on Java objects supported in the Java language, including run-time 'reflection' on the names, types and other properties of Java-language programmatic entities such as classes, fields and methods. (This support is essential to our filesystem's automatic traversal and naming of data structures and

their elements.)

We therefore have the choice as to how much of our filesystem is implemented in C, and how much (if any) in Java. Although C may offer performance advantages, JNI offers little compile-time checking of Java types, and with considerably clumsier syntax. Combined with the generally less productive and more bug-prone nature of the C language, we chose to bind the `fuse` API into Java at the lowest level, and keep the amount of C code to a minimum.

2.5 Tools and Techniques

Most of the development work was done under Eclipse, a cross-platform general-purpose Integrated Development Environment with available support for Java, C and C++. The toolchains used for Java and C are the Blackdown JDK and GNU `gcc` respectively, both de-facto standards on the GNU/Linux platform. Since debugging of JNI code within Eclipse is not currently possible, GNU `gdb` was used for this (incurring some familiarisation overhead).

Common techniques such as defensive programming and assertions were used throughout.

The strategy was to develop the system feature-by-feature, according to the milestones defined below. In some cases, individual features are interdependent and could not be usefully implemented separately, in which case they have formed part of the same milestone.

1. Successfully mounting the `fuse-hello` example filesystem on the development machine. This filesystem serves a single read-only file, containing the text "Hello, world!".

2. Re-implementation the `fuse-hello` filesystem in Java, on top of a simple Java binding of `libfuse`.

3. Binding of a single Java object into the filesystem, with its fields and methods visible in a directory listing, but without access to them.

4. Object method invocation on the local machine.

5. Primitive fields accessible as regular files.

6. Object-valued fields accessible as symbolic links.

7. Recursive binding of an entire object graph from a single call.

8. Indexer subdirectories.

9. Class (i.e. static) method invocation (locally only).
10. Object creation.
11. Accessor methods.
12. Remote method invocation over NFS.

2.6 Anticipated Problems

Before the start of coding, several potential problems were apparent which necessitated additional planning.

- Documentation on the `fuse` system is sparse, and generally assumes a familiarity with details of the filesystem API and common VFS calls. Considerable preparatory reading was necessary to gain this familiarity, mostly of Linux manual pages [16] and header files.

- The `fuse` library offers single- or multiple-threaded event dispatchers, allowing for one or many filesystem calls to be handled simultaneously by the process. This threading is implemented using the `pthreads` library. Since JVM threads are not necessarily implemented using `pthreads`, this greatly complicates concurrent programming with JNI.

  Since concurrent processing of multiple requests was not considered an essential feature, we elected to use only the single-threaded dispatcher. However, where practical, the relevant Java-side data structures are made thread-safe, to allow for future work on this problem. Other concurrency issues will be covered in the with the rest of the implementation.

- Methods, as mentioned previously, must be served as executable files, which are usually platform-dependent. Java offers one of the most platform-independent executable formats, the JAR, which is a compressed archive of Java bytecode files. Our implementation will serve methods as JAR files, and assume that client systems are configured to support direct execution of these (e.g. with `exec()`).

Chapter 3

Implementation

We begin with an overview of the complete system, followed by a detailed look at each component of the system in a feature-by-feature fashion. Familiarity with the Java and C languages is assumed.

3.1 Overview

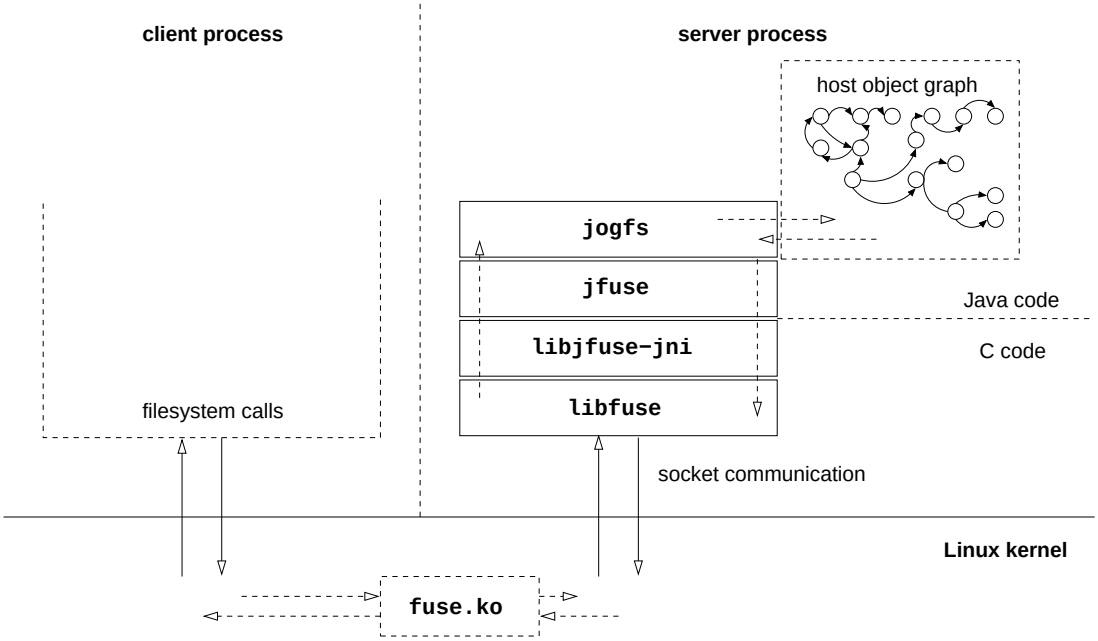

Figure 3.1: Structural Overview of the System

Figure 3.1 shows the high-level structure of the system, with dotted lines representing the path of a request as it travels from the client through the kernel

and into the user process, and of the response (typically a return code representing success or the class of error which occurred) making the reverse journey.

The components labelled `libfuse` and `fuse.ko` are, respectively, the `fuse` library and kernel module. The remaining three components constitute the implementation of the project. A brief explanation of each of these follows.

`libjfuse-jni` is the Java binding of the `libfuse` library. It is responsible only for calling the appropriate Java method, doing any marshalling required, selecting the appropriate return code and doing any low-level error reporting.

`jfuse` is a set of Java language classes abstracting from the low-level interface of `libjfuse-jni` into a familiar set of primitives including files, directories and symbolic links. Common routines such as pathname resolution and permission checking are provided here. This layer is agnostic towards the implementation of the filesystem, containing mostly abstract classes.

`jogfs` provides specialised implementations of `jfuse`'s base classes which implement all the logic to interrogate and update a Java object graph (hence its name).

3.1.1 Application Interface

By design, it is very easy for a Java application to link with our system and bind an object graph into the filesystem. A group of static methods are provided for starting and stopping the filesystem, binding and unbinding an object (and, recursively, the remainder of its graph) and querying the status of the filesystem.

3.1.2 Control Path

Figure 3.2 shows the threading of the system. On start-up of the filesystem, a new JVM thread is spawned which calls into the native `FuseMain.run()` method. This is the main entry point into the `libjfuse-jni` library, and starts up the `fuse` event handler loop. The loop reads queued requests (over the socket interface to the kernel module) and, through the C callbacks defined by `jfuse-jni`, calls into the Java code to perform the filesystem operations.

3.2 The `jfuse-jni` Library

This library interfaces with the `fuse`, and performs the following functions.

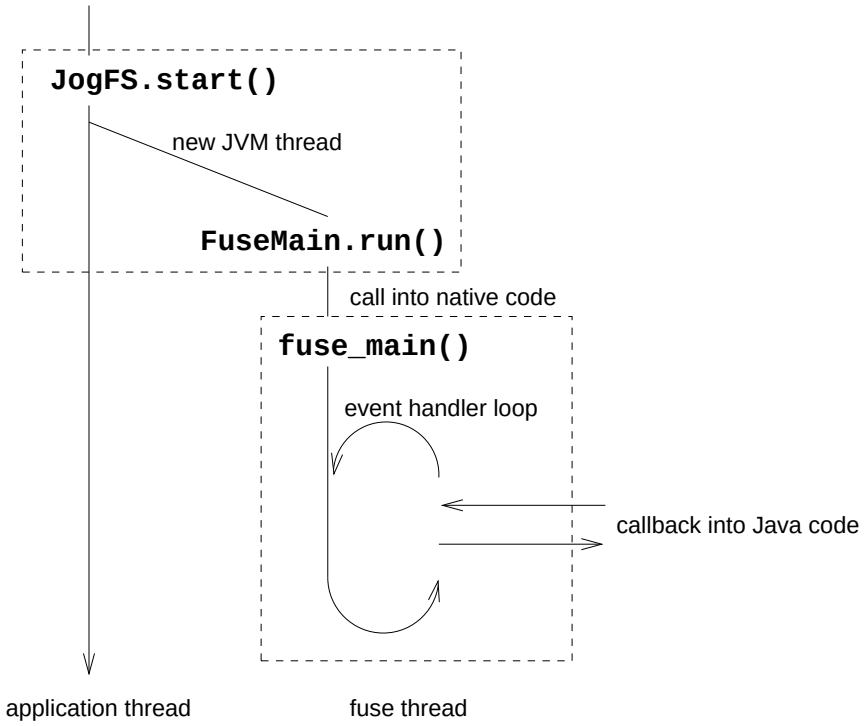

Figure 3.2: Threading and Control Flow

- Start-up and shut-down of the `fuse` event loop.
- Implementing the `fuse` callbacks by calling the appropriate Java method.
- Marshalling and translation of data between Java and C representations.
- Reporting Java exceptions using the appropriate error code.
- Implementing helper methods for the Java code, providing any features not implemented by the Java standard library.

The implementation is mostly straightforward, and is covered in Appendix A.

3.3 The `jfuse` Package

The `jfuse` package is a set of Java classes providing an abstract platform for defining filesystems in Java, independent of their concrete implementation.

We begin with a brief class-by-class summary, then discuss how the various features were implemented.

3.3.1 Class Summary

The classes can be split into two groups: those supporting the `fuse` binding, and those building on it. Most of the supporting classes were described in the previous section, but all are summarised here.

- The central `FuseOperations` interface and `FuseMain` class.
- Those used as arguments or return values to `FuseOperations` methods: `FuseStat`, `FuseStatFs`, `FuseDirectoryIterator`.
- Other classes/interfaces containing only constants or static methods: `FuseOpenFlags`, which defines mode constants for file opening, and `NativeHelpers`.
- Various exception classes.
- `FuseBasicOperations`, a simple no-op implementation of `FuseOperations`, and `FuseHello`, a Java port of `fuse`'s example "Hello, world!" filesystem, implemented as a subclass of `FuseBasicOperations`.

The remaining classes form a hierarchy, as shown in Figure 3.3. Italicised classes are abstract.

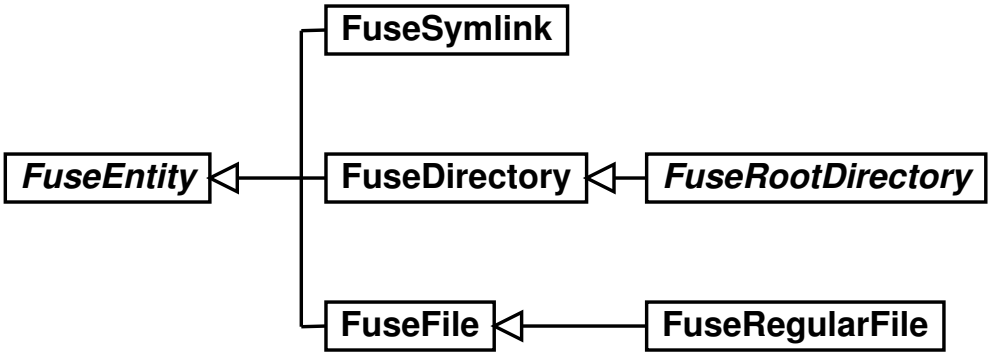

Figure 3.3: The `FuseEntity` hierarchy within `jfuse`

To implement `FuseOperations` methods in an object-oriented fashion, we first resolve the pathname argument to an object representing some part of the filesystem, and then invoke the corresponding operation on that object. Although all `FuseOperations` methods have a pathname as the first argument, the operation's logical 'target' is *either* the pre-existing entity represented by that pathname (e.g. `getattr()` or `read()`, `write()`), *or* the directory containing the implied (possibly nonexistent) entity (e.g. `unlink`, `symlink`, `mkdir`).

We therefore partition the operations into two: those which are applied to an existing target entity (as distinct from its containing directory) are defined by the `FuseEntity` class, while others are defined by `FuseDirectory`. Together, these two classes provide methods closely corresponding to each function of `FuseOperations`, although the method names are generally different to emphasise the distinction. The `FuseDefaultOperations` class is an adapter from the `FuseOperations` interface to that provided by `FuseEntity` and `FuseDirectory`.

An explanation of each remaining class follows.

`FuseEntity` is an abstract base class for any named entity in a filesystem. It provides default implementations for all methods, most of which simply throw an `UnsupportedOperationException` (resulting in an "Operation not permitted" error).

`FuseDirectory` is a base class for directories (i.e. partial mappings from names to entities, with some particular semantics). It can be used as-is, but like `FuseEntity`, most of its methods simply throw an exception.

`FuseRootDirectory` represents a directory with no parent, i.e. the filesystem root. It specifies a `getStatFs` method, returning information on the filesystem's status.

`FuseSymlink` implements a symbolic link (symlink), and can be used as-is to provide expected (POSIX) behaviour.

`FuseFile` is an abstract base class for any entity which is neither directory nor symlink. This might include named pipes, sockets, device special files and regular files. Its subclass `FuseRegularFile` is (currently) identical except that its constructor enforces the 'regular file' type.

3.3.2 Access Control

Access control is implemented by `jfuse` in a mostly straightforward POSIX-like manner. At present, only rudimentary checks for calling process privilege (i.e. to perform 'superuser' operations) are done. More detail on this is found in Appendix B.

3.3.3 Directory Lookup

Directory lookup is implemented by a method on `FuseDirectory` named `findEntity`, which resolves a relative pathname to a `FuseEntity` reference. The implementation has two steps.

The first step is a simple recursive look-up to find the containing directory, with the empty path[1] denoting the current directory (i.e. termination) and is built on the following path-handling primitives.

The second step is to look up the entity by name within the current directory. Every instance of `FuseDirectory` has a reference to an object implementing `java.util.Map` and keyed by entity name. This reference is set by a protected method called from the constructor, so subclasses can instantiate whatever kind of `Map` is suitable. The default implementation creates a `java.util.HashMap`, a simple hash table implementation. Hash tables are generally suitable for directory lookups since they offer $O(1)$ time complexity at a modest cost in space.

Directory look-up must be aware of symbolic links. Where a component of the path is a symlink to a directory, it should be followed, *except* if it is the last component of the path (and not followed by a trailing slash), when the path resolves to the link itself. For more information, see the `path_resolution` page in [16].

3.3.4 POSIX Semantics

No guarantee is provided or assumed, by either `fuse`, Linux's VFS layer or indeed `jfuse`, that a filesystem implementation's semantics conform to the POSIX standard. (For example, early versions of NFS [2] are impossible to implement in a fully POSIX-compliant manner.) On the other hand, a client application may expect compliant behaviour. As such, it is up to the filesystem implementation to provide compliant semantics as far as reasonably possible.

Since it is undesirable to re-implement these with every filesystem, and since `jfuse` is intended to support many different of filesystems, we have endeavoured to implement POSIX semantics as far as possible without compromising the rest of the project.[2]

Together with access control and pathname resolution, we also provide a set of POSIX implementations of the various methods declared by `FuseEntity` and `FuseDirectory`. These methods are declared `final` (non-polymorphic) and have names prefixed by '`std`'. Classes wishing to extend `FuseDirectory`, `FuseEntity` or one of their subclasses can call these methods to gain the expected POSIX behaviour. Clearly, this is limited in that details custom implementation is not

[1] POSIX specifies that the empty path should not be resolved successfully. Our implementation trivially respects this, since it will never be passed an empty pathname by `fuse`. The empty path is simply an internally-used symbol signalling termination.

[2] A notable omission is the `rename` method, which despite its name can perform moves as well as renames. This is by far the most complex of the specified operations in terms of error conditions and special cases, and was not needed by the `jogfs` project.

available to these 'standard' methods: some calls such as `read()` and `write()` have no standard implementation provided.

This latter problem is overcome somewhat in `FuseDirectory` by using the 'factory' design pattern. A protected method called from `FuseDirectory`'s constructor returns up a 'factory' object which provides methods to create a new file, directory and symbolic link. In this way a standard POSIX-like implementation of methods such as `createSymlink` and `createSubdirectory` can be provided without specifying what subclass of `FuseSymlink` or `FuseDirectory` to instantiate.

3.3.5 Symbolic Links

Some subtle problems arise concerning symbolic links. The problem of path resolution was mentioned in Section 3.3.3. Other problems and their solutions are outlined below.

- Circularity: symbolic links can refer to other symbolic links, and may form cycles. In many situations we may wish to get the 'concrete' target of a symlink, i.e. the eventual (non-symlink) target found by repeatedly following links. To avoid potential non-termination we use an iterative (rather than recursive) approach which remembers which links have been visited; if one is reached for a second time, an exception is thrown.

- Duality: where a symbolic link is specified as the target of an operation, whether the operation should be performed on the link itself or its concrete target depends on the operation. For example, a symbolic link has permission bits just like any filesystem entity, but in practice these are never used, since the target's permission bits are always the ones checked. Careful observance of the documented correct behaviour is required.

- External linking: symbolic links (unlike hard links) may refer outside the filesystem containing them, either by absolute path or by containing a sufficiently-long sequence of '`..`' elements. Resolving such a link requires us to resolve an external filesystem entity as a `FuseEntity` object. Our implementation has no specific knowledge of external filesystems, but can query the filesystem like any other user process. One solution is to create a 'shadow' `FuseEntity`, whose implementation simply accesses the filesystem using the standard Java library. This is currently unimplemented; attempts to set external symlink targets return an error.

3.4 The jogfs Package

The jogfs system, or Java Object Graph File System, is a general-purpose synthetic filesystem built on top of jfuse. We begin with a brief summary of the component Java classes, followed by a feature-by-feature description of the implementation.

3.4.1 Class Overview

The complete FuseEntity hierarchy is shown in Figure 3.4.

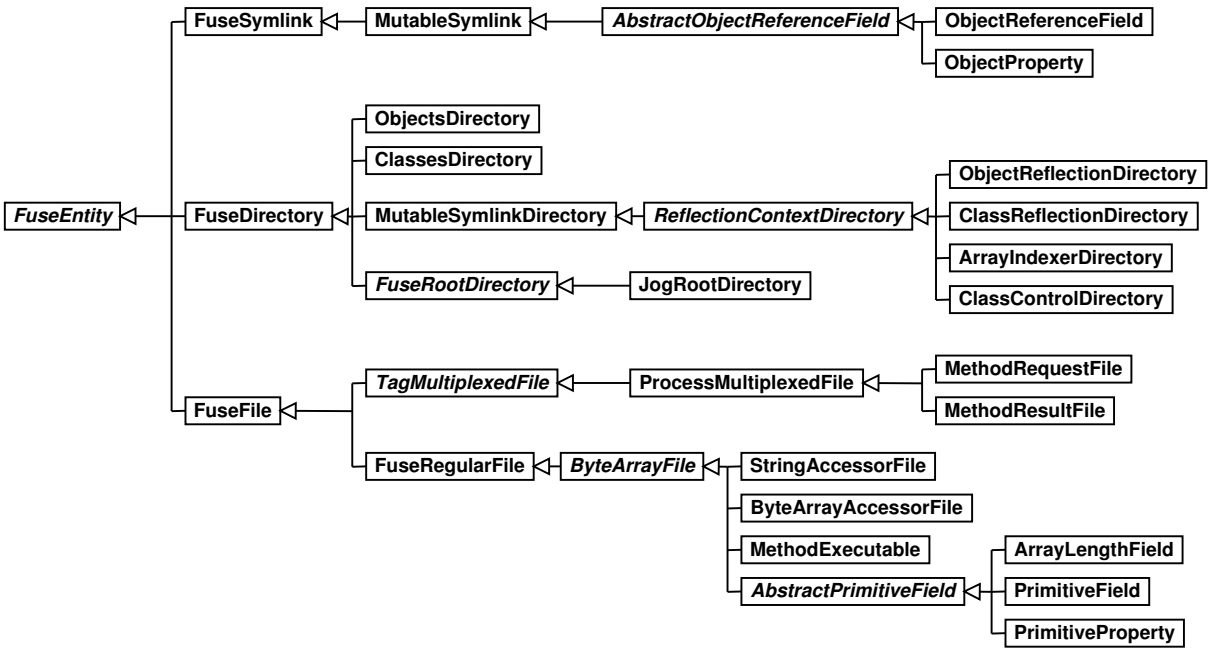

Figure 3.4: Class hierarchy for jogfs, including jfuse base classes

Rather than covering each class individually, all will be covered on a per-feature basis in the sections to follow.

3.4.2 Filesystem Layout

The filesystem has a well-defined directory structure, shown in Figure 3.5.

3.4.3 Design Issues

Before and during implementation, consideration was given to various alternative approaches, and to some potential problems not foreseen at the planning stage.

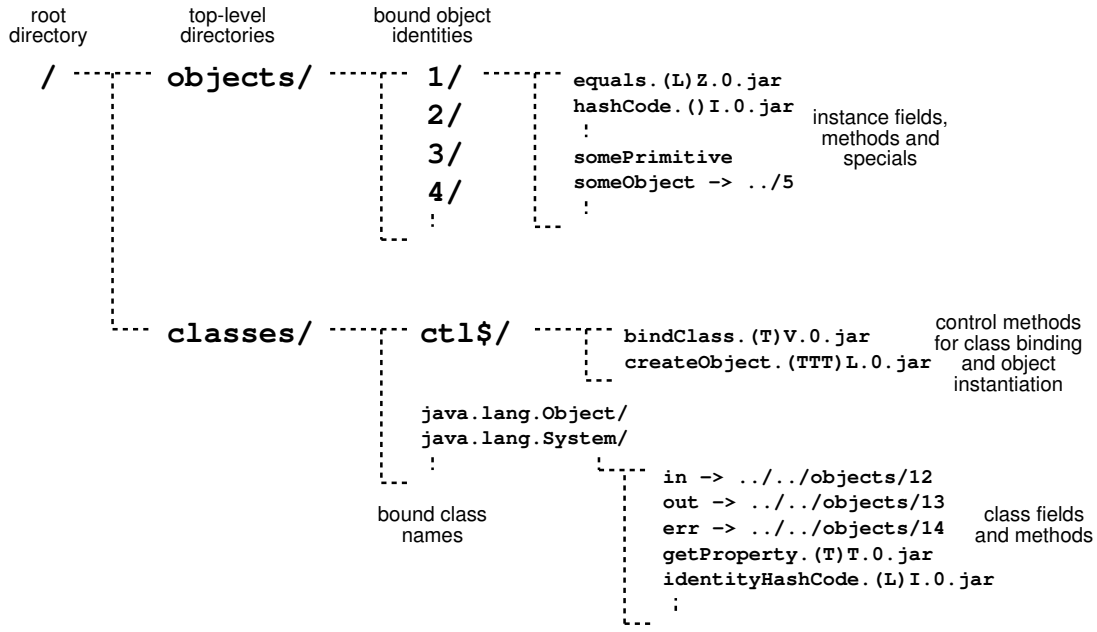

Figure 3.5: Directory structure of the `jogfs` filesystem

Stateful *versus* Stateless

The intrinsic state of our filesystem is, by definition, the identities of the objects and classes which are bound into it at a given time, and the state of those objects and classes. This contrasts with most filesystem implementations, whose role is to manage the storage of state (i.e. data) internally. Consequently, we have two implementation options.

- The "stateful" approach: maintain the filesystem structure as a DAG of objects (instances of `FuseEntity` subclasses), referencing the underlying object graph. This is the simplest approach, with the only apparent difficulty being the task of keeping the parallel data structure consistent with the underlying graph. However, it implies a significant memory overhead.

- The "stateless" approach: maintain minimal state, and generate `FuseEntity` objects as required on a per-call basis. This approach is more complex, particularly when considering synchronization in a concurrent implementation, and potentially slower (owing to frequent object creation, and rebuilding of state with each request). However, it offers lower memory overhead.

For the sake of simplicity, we have mostly taken the stateful approach. However, nothing dictates that only one approach can be used, and some parts of the

system (such as the indexer directories – see Section 3.4.10) use a combination of both approaches.

Concurrency

There are three levels of concurrency to be considered in our filesystem, although none present problems to our current implementation.

Object-level concurrency means performing concurrent method invocations or field accesses on the underlying Java objects, as in any multi-threaded Java program. Every Java program exposing a filesystem will have at least two threads running: at least one application thread, and at least one filesystem thread. In the case of only one application thread, our filesystem allows concurrent accesses to data structures possibly designed only for single-threaded use. It remains the programmer's responsibility to provide appropriate thread safety, or else avoid exposing unsafe objects through the filesystem.

Operation-level concurrency means servicing multiple filesystem requests on the same object graph simultaneously. The usual concurrent programming problems occur: concurrent accesses to data structures (such as directory `Map` instances) must be synchronized, with deadlock a possible problem if simple mutexes are used. This requires the multi-threaded `fuse` event loop, which is not currently used (although a future revision may enable it) so we need not consider this further now.

Process-level concurrency means having multiple processes performing accesses to the filesystem in the same interval of time. These accesses are interleaved differently depending on how the processes are scheduled. Since updates to a file are not atomic (in that they may take more than one `write()` call to accomplish, while the individual writes may or may not be atomic), the usual 'lost update' and 'dirty read' problems arise.

Fortunately, advisory MRSW ("multiple readers [or] single writer") locking of file regions is provided by Linux's `fcntl()` system call, and supported by `fuse`. Client processes are therefore responsible for handling this kind of concurrency.

Interactions with Garbage Collection

If our filesystem were to maintain references to the objects in the exposed object graph, this would have the undesirable consequence of preventing the ob-

jects from being garbage collected, since they will always be reachable through the filesystem. We avoid this problem by generating a unique integer ID for each object bound, and using it to refer to the object. By default, only the `ObjectIdentityMap` class keeps references, and this keeps only a 'weak reference' to the object (see Section 3.4.5).

Conversely, an object which has been garbage collected may still appear in the filesystem, since collection occurs asynchronously and without notification. We therefore check the status of each object on access to its directory or child entities. If the object no longer exists, a "No such file or directory" error is returned and the directory is removed from the filesystem. (See also section 3.4.11.)

Special Objects

Primitive Java data are represented textually for input and output through the filesystem, while objects are referred to by their path. It is also convenient to allow character strings to be represented literally. Since strings in Java are objects (instances of `java.lang.String`), this presents a problem. Strings can now be expressed in two ways: a path, or a literal. These may be ambiguous – for example, a pathname passed as an argument to a method could also be interpreted as a string literal.

We therefore prefix string literals by the '`@`' character, which is unlikely to appear at the start of a path (and is never required to do so: prefixing '`./`' will always yield a usable equivalent). This allows literals to be used in place of an object.

The converse problem also requires a special solution: it may be useful to obtain the literal value of a `String` object bound in the filesystem. Calling the `toString` method works (since methods returning `String`s always return a literal), but to avoid the overhead of a method call, a special read-only '`$`' file is served in each `String` object directory. This is implemented by the `StringFile` class, and contains the UTF-8 bytes of the `String`.

A similar fixed-length '`$` file' is also served for byte arrays, and allows both reads and writes.

3.4.4 Binding an Object

All objects are bound within the `objects` top-level directory, implemented by `ObjectsDirectory`. This class tracks which objects are bound, and handles requests to bind and unbind (as explained in Section 3.4.11).

An object directory is always named after its ID number, though option-

ally a symbolic link may be created alongside providing a 'friendly name'. The `ReflectionContextDirectory` class is a base for both object directories and class directories (see Section 3.4.11), while subclasses `ObjectReflectionDirectory` and `ClassReflectionDirectory` implement anything specific to one or other kind.

3.4.5 Object Identity

Object ID numbers are generated by the `ObjectIdentityMap` class, of which `ObjectsDirectory` maintains an instance. The specification of this class is that it should:

- Provide a unique ID number for any object. If an object is passed in twice, the same ID should result each time.
- Not prevent garbage collection of any object for which an ID has been issued.
- Get the object corresponding to a given ID, or null if the object no longer exists, or throw an exception if that ID has not been issued.

The Java standard library provides two useful classes for this.

- `WeakReference`, which keeps a reference to an object but doesn't prevent it from being garbage collected.
- `System.identityHashCode()`, which generates a hash code dependent only on the object's identity rather than its value. (This cannot be used as an object ID itself because it is not guaranteed to be unique.)

It also provides `IdentityHashMap`, a hash table which tests keys for reference equality (`a == b`) rather than value equality (`a.equals(b)`), and `WeakHashMap`, a hash table in which the keys are weakly referenced. Unfortunately, these cannot be used, since we require reference equality *and* weak referencing at the same time, and also a bi-directional mapping.

Instead, we subclass the `WeakReference` type (creating the `IdentityWeakReference` class) so that its `equals` method tests reference equality of the *targets* of the two references, and that its `hashCode` method returns the identity hash code. This can then be used in a regular `HashMap` instance to provide the required 'weak identity' semantics. The values of this hash table are `Integer` instances containing the object's ID number. The reverse mapping is maintained as another `HashMap`.

A problem arises with the use of `IdentityWeakReference`s as keys. `IdentityWeakReference` instances are mutable, in that when the target object is garbage-collected, the reference will be cleared and the hash code change to zero (the defined identity hash code for a null reference). However, the key will still reside in the hash table bucket corresponding to its old hash code. Fortunately, Java's standard `HashMap` implementation is tolerant of this. When the table is re-sized, the entry will be re-hashed and put into the bucket corresponding to a hash code of zero. To prevent build-up of these stale references, we override the `put` method to remove them before continuing.

3.4.6 Method Invocation

Overview

As described in Chapter 2, named procedure calls cannot be exposed directly by a filesystem. As such, any implementation of method invocation will inevitably be a compromise. Our technique is to serve an executable file which communicates with the process serving the filesystem to perform the method call. This communication itself should occur through the filesystem, so that if the filesystem is served over a network, a remote client can have the method run on the server machine. The executable served must be in a cross-platform format, so that clients on different architectures and operating systems can use the system.

We each method as a JAR file (an archive containing Java bytecode, which can be run on any machine with a Java runtime installed), and uses two special files in the object (or class) directory to perform the communication. Figure 3.6 depicts the system in the general case of the filesystem being served to a remote client wishing to invoke a method.

The Method File

Within an object or class directory, methods are given filenames consisting of the method name and a symbolic string representing the method signature (to allow for overloads), followed by the `.jar` extension. These files are implemented by the `MethodExecutable` class.

The archive contains three Java class files:

- `MethodExecutorMain.class`, the code which is run when the JAR file is executed.
- `MethodCallDescriptor.class`, defining a simple data structure describing the arguments of a method call.

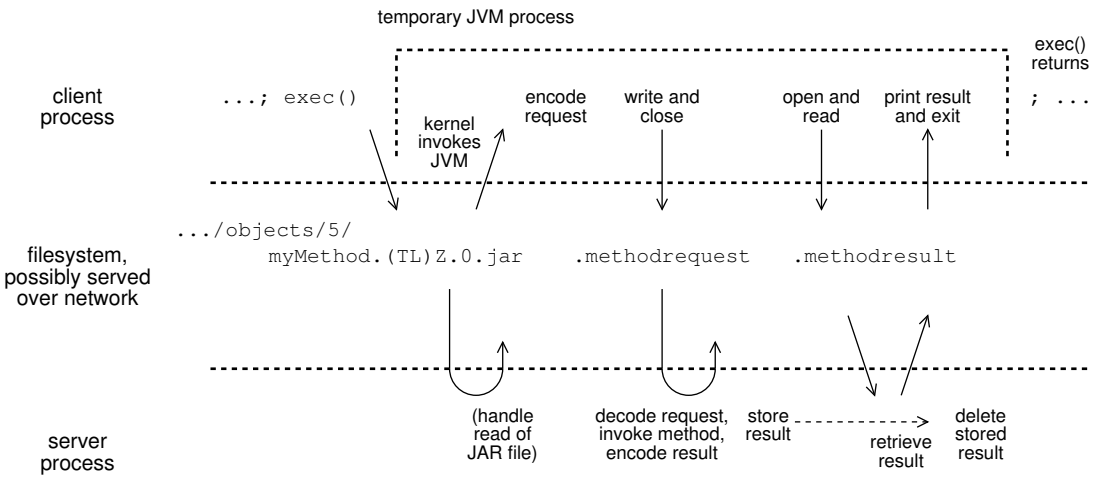

Figure 3.6: Method invocation

- `MethodResultDescriptor.class`, defining a simple data structure describing the result of a method call (either the return value or an error message).

The contents of each JAR file are identical, and are stored in a single shared byte array. The target object and method are encoded entirely within the name and path of the JAR file: the name composed of three parts, each separated by a '.'. First is the method name; second its signature encoded as a String; third a numerical disambiguation string (usually '`0`').

The signature is encoded similarly to the JNI signature string [15], except for two differences:

- Full class names are not specified: instead, the '`L`' character is used alone. This keeps filenames fairly concise, but necessitates the disambiguation number where different overloads take the same number of object-valued parameters.

- The `java.lang.String` class is represented by the token '`T`', since it is commonly-used and often treated differently.

The Executor

The executor is the code within the `main()` method of the `MethodExecutorMain` class. It accepts command-line arguments corresponding to the target method's arguments. Object-valued arguments are encoded as paths, while primitive-valued arguments may be in any format which the relevant class's `parse`*`Type`*`()` method (e.g. `Integer.parseInt()`, `Double.parseDouble()`) can parse. Parsing is done on the server side. String literals may also be passed, prefixed by '`@`'.

To determine the target object and method, the JAR file name and path are obtained using the `Class.getProtectionDomain()` method. If the class is not being executed from a JAR file on the filesystem, the executor exits with an error.

The executor must work over the network as well as on the local filesystem. Object paths will differ in prefix between the client and server side, so 'path translation' must be done. This translates an object path into one relative to the root of the `jogfs` filesystem (using a forward slash as separator regardless of the executing platform's separator). The signature string is used to determine which arguments to translate (i.e. which are object-valued).

The executor opens the `.methodrequest` file in the target class or object directory, and writes out a serialized instance of `MethodCallDescriptor` using standard Java object serialization. This is a simple object containing only the JAR name and the arguments (path-translated where appropriate).

When the file is closed, the method is invoked and the results made available in `.methodresult` as a serialized instance of `MethodResultDescriptor`. Again, this is a simple object containing two string references: the result and an error message. Only one of these will be non-null. For object return values, inverse path translation is done on the result string.

If an error occurred, the message is printed on the standard error; otherwise, the result is output on the standard output.

Request and Result Files

Both `.methodrequest` and `.methodresult` are regular files, rather than sockets or other special files, because it is more likely to be portable and does not imply the existence of any in-kernel data structure (such as a socket or pipe).

The files are implemented as subclasses of `ProcessMultiplexedFile`, a file which may contain different data for each process which makes calls on it. This allows concurrent calls from more than one process to be underway simultaneously. Since each completed method call consists of a sequence of at least six filesystem calls (`open()`, `write()`, `close()`, `open()`, `read()`, `close()`), this is necessary even when using a single-threaded event loop. (In other words, it admits process-level concurrency.)

The data underlying the request file is implemented behind an interface named `ByteSequence`, which defines a small set of file-like operations such as `read()`, `write()` and `truncate()`. The default implementation, called `SimpleByteSequence`, uses the standard `ByteArrayOutputStream` class and resorts to data copying when rewinding or truncating. An implementation of `ByteSequence` is also used for mutable primitive fields (see Section 3.4.7).

More detail on performing a method call is available in Appendix C.

Demultiplexing Issues

A subtlety with request and result files is that multiple networked mounts will often appear as accesses by the same process. Consider an NFS server making accesses on behalf of multiple client machines, each trying to perform a method call.

No solution has yet been implemented for this problem, though a likely approach would be to disallow multiple opens of a request file by the same process, returning a "resource temporarily unavailable" error if this was attempted. The client might then have to make several attempts at opening the request file before succeeding, and other NFS clients would be excluded until the file was closed again.

The disadvantage is that this reduces the available concurrency. Since no client identification (beyond the process ID) is available to demultiplex the calls, a better solution would require changes to the `fuse` interface (and possibly to others). For example, if a multithreaded NFS server were used, the calling thread ID could be used to demultiplex the calls, but the `fuse` interface does not expose this.

Problems with `fuse`

A problem arises when attempting to perform actions on a `close()` call with `fuse`, originating as follows.

- `fuse` does not explicitly accept a `close()` callback.
- Instead, it provides `release()`, which is called exactly once for each `open()` once all handles on a file have been closed and all memory mappings unmapped, and `flush()`, which is called once for every `close()`.
- Both provide no guarantees about when they are called: they may be issued immediately on the relevant `close()` or `munmap()`, or asynchronously some time after that.
- When `flush()` or `release()` are issued asynchronously, the calling process ID shows as zero rather than the process making the final `close()` or `munmap()`.

- Experiments show that a call to `close()` followed by a call to `open()` on another file may generate the corresponding `fuse()` calls in the reverse order, i.e. `open()` then `flush()`.

As a result, `close()` operations cannot be demultiplexed correctly, and therefore cannot be used as a trigger for method invocation. Naively attempting all open requests also fails, because the client process may already have called `open()` on the result file, and found it empty, when the `flush()` call is received.

The solution is to attempt a request after every `write()` operation. These can always be demultiplexed correctly. Since request files are small (rarely more than a few hundred bytes), it is likely that a single `write()` operation will transfer the whole request, so little CPU time is wasted on attempting deserialization of incomplete data. If the request does fail, we simply wait for another write and try again.

Since `close()` cannot be demultiplexed, timeout of stale requests is essential to avoid a memory leak. On every `flush()`, unsubmitted request data older than two minutes is purged, irrespective of the process owning it.

Remaining Problems

Aside from the demultiplexing issue mentioned above, the major problem with this interface is that a large payload[3], i.e. the JAR file, must be delivered (possibly across a network) for every method call. Since the payload appears distinct for each method, despite containing the same data, it is unlikely to be cached on the client side. Furthermore, starting the JVM on the client fintroduces a substantial time and memory overhead. Such a system is adequate for scripting applications, but is far from being an efficient RPC interface.

Of course, there is no obligation to use the JAR file. Any valid request written to the `.methodrequest` file will generate a result as expected. It is perfectly possible to write a high-performance RPC library using this interface. However, since method calls are no longer part of the filesystem namespace, and extra client-side logic is required to perform them, this conflicts with the objectives of a general-purpose synthetic filesystem.

[3]The JAR file served is currently approximately 13KB, uncompressed, but we estimate that compression and code optimisations could reduce this by three-quarters.

3.4.7 Primitive Fields as Files

Overview

Primitive fields represent values of a primitive Java type such as `int`, `double`, `boolean` and others, and appear in the filesystem as regular files. They are implemented as subclasses of the `AbstractPrimitiveField` class. In all cases, the file is represented as an array of bytes containing the current value of the field (regenerated from the underlying object whenever required) and an instance of `PartialByteSequence` (see Appendix D) which records any updates that have been made to the field.

Updating

When the field is closed (or rather a `flush()` call is received), a new byte array is generated by superimposing the updates on the current data. The resulting byte array is turned into a UTF-8 string and parsed into an object corresponding to the underlying primitive type (e.g. `Integer` for an `int`, `Double` for a `double`, etc.).

Subclassing

What happens to the resulting object is left for a subclass to decide, by implementing the `commitObject()` abstract method. In the case of a simple field (as implemented by `PrimitiveField`), this would use the reflection API to set the field value to the object passed in. If type checking fails, an "invalid argument" error is returned. The subclass must also implement the `getBytes()` method, which returns the UTF-8 bytes of the field's current value in string form.

3.4.8 Object Fields as Symbolic Links

Overview

Object fields represent references to Java objects, and appear in the filesystem as symlinks pointing to object directories. Recursive binding (see Section 3.4.11) ensures that referenced objects are always bound into the filesystem, while type-checking is provided as usual by the Java runtime. The `AbstractObjectReferenceField` class, a subclass of `FuseSymlink`, is the base for all object reference field-like entities.

Mutable Symbolic Links

Symbolic links are not updateable – POSIX specifically states for `symlink()` that if the link pathname exists, it is not overwritten. Consequently we are forced to implement a two-stage update: remove the old link, then create a new one.

To avoid creating a new `FuseEntity` instance each time this is done, we subclass `FuseDirectory` and `FuseSymlink`, giving `MutableSymlinkDirectory` and `MutableSymlink`. The directory implementation keeps a cache of `MutableSymlink` entities when they are unlinked, and reinstates them when a 'new' symlink with the same name is created, but updating their target. `AbstractObjectReferenceField` is a subclass of `MutableSymlink`.

Unfortunately, there is nothing to stop a client from removing a symlink and never reinstating it. Any client wishing to read the symlink's value would then find it absent. This could be partially solved by multiplexing the directory on a per-process basis (though victim to the same problems as method calls – see Section 3.4.6), or by restoring links from the cache after a short time. At present, neither of these is implemented.

Setting the Target Path

When created, all `AbstractObjectReferenceField` instances are assumed to be null references. These are given the target path 'null$', which is guaranteed not to point to anything in the current directory (since no field or method may be named this).

Setting the target path requires the following steps.

1. If the new target path exists, then resolve it as an `ObjectReflectionDirectory` instance and get its target object.

2. Otherwise, extract the target object identity (since the path will be of the form `path/to/objects/n` where n is some object ID) and get the object reference from `ObjectIdentityMap`. This is necessary as sometimes the target directory will not be created until later in the recursive binding process.

3. Call `updateUnderlyingReference()` for the new target object.

4. If the update fails (because of type-checking), return to the old target object.

5. Otherwise, update the symlink's size to reflect the new target path length and return.

In the case of a Java-language field, `updateUnderlyingReference()` uses the reflection API to set the underlying reference. In the case of a property setter, the reflection API would be used to call the 'set' method for with the specified argument.

3.4.9 Special Fields

We may use the term 'fields' to refer not only to Java-language fields, but also to a host of field-like entities, including accessor methods ("getters and setters"), array elements and special '`$`' fields of strings and byte arrays.

Like conventional fields, all these entities are implemented as subclasses of `AbstractObjectReferenceField` or `AbstractPrimitiveField`.

Aside from Java-language fields, we identify the following field-like entities.

- Array 'length' fields. These are not supported by the reflection API, but are implemented separately as a simple read-only subclass of `AbstractPrimitiveField`.
- Property 'getters' and 'setters'. These are methods conforming to the JavaBeans conventions [12], and can be thought of as fields exposed through a layer of code rather than directly. At present these are not implemented, but can be added straightforwardly as subclasses of `AbstractObjectReferenceFields` and `AbstractPrimitiveFields`.
- String and byte array accessor files. These are described in Section 3.4.3 and implemented as subclasses of `ByteArrayFile` (also the superclass of `MethodExecutable`).
- Array index accessors. These are explained in the following section.

3.4.10 Indexer Directories

Within the object directories of arrays, a special subdirectory named `index$` is exposed. This has a numbered entry for each array element, where each entry resembles a field (either primitive or object reference depending on the type of the array). These behave exactly like Java-language fields, but are implemented using the reflection API's `Array` class to manipulate array elements. They are also generated on request, following the "stateless" design approach.

3.4.11 Recursive Binding

A key feature of the filesystem is that upon binding an object, all objects accessible through its public fields should also be bound. If an object is unbound, any object no longer reachable should be unbound. (Here 'reachable' means reachable through the filesystem, rather than the garbage-collection sense.)

There are four ways in which an object may be bound into the filesystem.

Direct binding means that the application code has specifically asked for the target object to be bound.

Indirect binding means that the object is bound because it is referenced by an accessible field of another bound object.

Strong binding means that the object is bound and strongly referenced by the filesystem (as opposed to the conventional weak referencing). This is used for objects created through the filesystem, to prevent these from being immediately garbage-collected.

Transient binding means that the object is bound for a limited time only (typically five minutes). This is used for objects returned by method calls, to avoid build-up of temporary objects in the filesystem.

Binding and Unbinding

Each `ObjectReflectionDirectory` instance maintains accounting information on how it is bound. `ObjectsDirectory` provides the interface for binding and unbinding objects, and updates each object directory's accounts accordingly.

- Direct binding is represented by a Boolean flag, true if the object is direct-bound.
- Indirect binding is represented by a set of the referring `AbstractObjectReferenceField` instances.
- Strong binding is represented by a list of object references to the target object. This makes strong binding reentrant: an object may be bound multiple times over, and is only left unreferenced when it has been unbound the corresponding number of times.
- Transient binding is represented by a set of timestamps, kept sorted in a tree so that bindings older than a given timestamp can efficiently be removed.

Recursion

On binding an object, all connected objects must also be bound. Since the resulting graph may have cycles, this must be done carefully to avoid an infinite loop.

Our algorithm uses two alternating phases: creation and population. It is outlined below.

1. If the target object is already bound, return.
2. Otherwise, create the object directory for the target object, but do not populate it with fields.
3. Add the new directory to a global queue of directories awaiting population.
4. While the queue is non-empty, pop the head and populate it.

As object reference fields are created, they recursively bind their target object using the same algorithm, but do not recurse any further except by popping from the queue. Only newly-referenced directories are added to the queue. Since only a finite number of objects may be in the graph, only a finite number of directories may be added to the queue, so the algorithm is guaranteed to terminate.

Unbinding

Unbinding is necessary in three cases.

- Unbinding of a directly- or strongly-bound object is explicitly requested.
- An object reference field is updated such that the former target is no longer part of a bound object graph (i.e. it is no longer 'reachable' from a direct, strong or transient binding).
- Access to an underlying object fails because the object has been garbage-collected. This only happens when an object is not reachable by a Java object reference, so may only happen for direct- or transient-bound objects.

This has clear parallels with garbage collection, and we use a familiar algorithm to solve it. Every time an object reference is updated, we run a "mark and sweep" algorithm on the set of bound objects. This works as follows.

1. Mark each bound object unreachable. (`ObjectReflectionDirectory` defines a flag for this.)

2. For each direct-, transient- or strongly-bound object,
   - Mark its object directory reachable.
   - For each instance of `AbstractObjectReferenceField` in the directory, recursively mark the target object directory reachable.
3. For each bound object, test whether it is marked reachable. If not, remove it.

Considering that unbinding disconnected objects is not crucial for correctness, it may be preferable to run this routine only every n updates or m minutes.

In the case of a bound object being garbage-collected, its directory may be immediately removed from the filesystem since it is not referred to by any other objects. However, it may itself have been referring to objects which are not otherwise filesystem-reachable, so a mark-and-sweep should be run after this to remove those objects.

Class Binding and Object Creation

As well as binding objects into the filesystem, we may also bind classes, in order to provide access to their static fields and methods. This operates very similarly to object binding, except that there is no recursive binding. The set of classes bound at any one time is defined to be the union of the set of classes of bound objects and the set of classes explicitly bound.

The `classes` Directory The `classes` directory is implemented as a special implementation of `Map` called `UnionMap`. This takes an array of constituent `Map` instances and defines its mappings as the union of the mappings provided by these. In the case of the same key appearing in more than one `Map`, the `Map` at the lowest array index 'wins'. Mappings cannot directly be added to the `Map`, but must be added to one of the constituent maps.

In the case of the `classes` directory, the constituent maps are a `HashMap` of explicitly-bound classes and a `Map` derived from the set of classes of bound objects, as provided by `ObjectsDirectory`. (This 'derivation' uses another kind of `Map` called a `FunctionMap`, which applies a specific function to a key to obtain the corresponding value, here mapping class names to `Class` objects.)

The `ctl$` Directory The `classes` directory contains a special subdirectory called '`ctl$`', which exposes 'control' operations for binding classes and creating

objects. This is implemented essentially as a bound object directory: it exposes a special object containing only the methods to bind classes and instantiate objects.

Classes are bound by executing the `bindClass.(T)V.0.jar` file, with the fully-qualified class name as argument (prefixed by a '`@`', as it is a string literal).

Likewise, objects may be instantiated by executing the `createObject.(TTT)L.0.jar` file. This takes three arguments: the fully-qualified class name, the constructor signature, and the constructor arguments encoded as a single string with each argument enclosed in double-quotes. It returns a reference to the newly-created object, which is strongly-bound.

Class unbinding and strong-unbinding of created objects are not yet implemented, but could be added straightforwardly.

Chapter 4

Evaluation

Many different criteria may be used to evaluate the project. We will break these down into three areas:

- Assessment of the current state of the project, including features implemented and known problems.
- Demonstration of the system using some test cases, and discussion of testing and reliability issues.
- Consideration of the experimental side of the project, discussing problems caused by limitations of the filesystem API.

4.1 Current State

4.1.1 Summary of Features and Problems

The system currently consists of approximately 1,300 lines of C and 8,750 lines of Java. It implements all mandatory functions originally specified (see Section 2.1), and also the object creation (considered optional), but at present lacks the following:

- Exposition of accessor methods similarly to fields. (At present they are available only as methods.) This omission is due to time constraints, though the feature would be straightfoward to implement.
- Indexer directories for objects implementing `java.util.Map`. The semantics of this require further thought, since it is not clear how the entries in the indexer directory would be named.

- Use of the filesystem over NFS. There are no particular barriers to this, since `fuse` filesystems are known to work over NFS, but on the current test system, mounting fails with the NFS server generating a "bogus `i_mode`" message. This could probably be fixed by some subtle changes to filesystem parameters, but there has not been time to experiment further.

There are also some known problems or inadequacies with existing implementations, summarised below. Most of these were detailed in Chapter 3, along with suggested fixes.

Problems in the `jfuse` layers:

- Only one filesystem may be exposed per application.
- Only the single-threaded `fuse` event loop is currently supported.
- No POSIX-compliant implementation of `rename()` is provided.
- There is no well-defined interface for checking for a privileged caller (except testing for user ID zero, i.e. the root user).
- Symbolic links cannot point outside the filesystem being served.

Problems with `jogfs`:

- Method invocation using `exec()` requires a Java runtime on the client system, and has high overheads in time and memory.
- Method requests are attempted for every write to the request file, rather than on closing the file.
- Concurrent method requests can only be demultiplexed on a per-process basis, reducing the concurrency available e.g. among NFS clients.
- Symbolic links (i.e. fields) can be removed from object directories by a client process and subsequently not re-created. (Any client may re-create them, but is unable to read the field's value before doing so.)
- There is currently no way to unbind classes, or remove strong bindings on objects created throug the filesystem, or to make permanent a transient binding on a method result (except by creating a new object that refers to it).
- The expensive mark-and-sweep procedure runs on every update to an object reference.

- Certain programmatic features of Java have no filesystem equivalent, leading to certain functionality to be missing. For instance, there is no way to create a subclass through the filesystem, so interfaces which rely on subclassing (such as protected constructors) are unusable. Similarly, a Java object's mutex cannot be manipulated through the filesystem.

- The filesystem currently cannot be mounted over NFS, so remote method invocation has not yet been tested.

4.1.2 Testing and Reliability

Limitations

A piece of software such as this can only reach stability through prolonged testing in realistic usage scenarios. Clearly, this has not been possible in the development time available, and will not be appropriate until the system has underdone at least one more development iteration.

Nevertheless, the system has proved sufficiently stable to undergo repeated tests of individual operations without problem.

Test Cases

We present a series of output samples from command-line manipulation of a test filesystem. The object graph has been designed to permit illustration of all key features, and its definition shown in Figure 4.1.

First we inspect the top-level directories of the filesystem and the contents of the first object. Initially there are four objects bound – the test object and the three it refers to.

```
stephen@srk31:~/proc/12418/jog$ ls
classes  objects
stephen@srk31:~/proc/12418/jog$ cd objects; ls
1  2  3  4
stephen@srk31:~/proc/12418/jog/objects$ cd 1; ls -l
total 10
lrwxrwxrwx  1 stephen stephen     4 May  9 12:04 author -> ../3
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 checkNumericalEquality.(TI)Z.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 equals.(L)Z.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 getClass.()L.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 getNewObject.()L.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 getSomeInt.()I.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 hashCode.()I.0.jar
lrwxrwxrwx  1 stephen stephen     4 May  9 12:04 magic -> ../4
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 notify.()V.0.jar
```

```
public static class TestClass
{
    public Object objectField = new Object() {
     public Object obj1 = new Object();
     public Object obj2 = obj1;
    };
    public Object testObjectField = null;
    public Object self = this;
    private Object hidden = null;
    public int thisYear = 2005;
    public double pi = 3.1415926;
    public String author = "Stephen Kell";
    public byte[] magic = new byte[] { 0x4b, 0x5e, 0x6e, 0x7f  };
    public int getSomeInt() { return 1984; }
    public Object getNewObject() { return new Object(); }
    public boolean checkNumericalEquality(String s, int i)
    {
     return Integer.parseInt(s) == i;
    }
    public void setTestObjectField(Object o) { testObjectField = o; }
}
Object jogTestGraph = new TestClass();
```

Figure 4.1: Test object graph

```
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 notifyAll.()V.0.jar
lrwxrwxrwx  1 stephen stephen     4 May  9 12:04 objectField -> ../2
-rw-r--r--  1 stephen stephen     9 May  9 12:04 pi
lrwxrwxrwx  1 stephen stephen     4 May  9 12:04 self -> ../1
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 setTestObjectField.(L)V.0.jar
lrwxrwxrwx  1 stephen stephen     5 May  9 12:04 testObjectField -> $null
-rw-r--r--  1 stephen stephen     4 May  9 12:04 thisYear
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 toString.()T.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 wait.()V.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 wait.(J)V.0.jar
-r-xr-xr-x  1 stephen stephen 13373 May  9 12:04 wait.(JI)V.0.jar
```

Next we perform some test method calls, using `Object`'s `equals()` method.

```
stephen@srk31:~/proc/12418/jog/objects/1$ ./equals.\(L\)Z.0.jar .
true
stephen@srk31:~/proc/12418/jog/objects/1$ ./equals.\(L\)Z.0.jar self
true
stephen@srk31:~/proc/12418/jog/objects/1$ ./equals.\(L\)Z.0.jar objectField
false
```

Now we call a method to set the field `testObjectField`, currently null, to point to the object.

```
stephen@srk31:~/proc/12418/jog/objects/1$ ./setTestObjectField.\(L\)V.0.jar .
stephen@srk31:~/proc/12418/jog/objects/1$ ls -l testObjectField
lrwxrwxrwx  1 stephen stephen 4 May  9 12:04 testObjectField -> ../1
```

Next we call a method with two arguments: a string and an integer. First, we use a string literal and the integer value returned by `getSomeInt()` (retrieved by a method call in a subshell). Second, we try with an object reference to a `String` which can't be parsed as an integer, and discover that an exception was thrown.

```
stephen@srk31:~/proc/12418/jog/objects/1$ ./checkNumericalEquality.\(TI\)Z.0.jar
 @1984 `./getSomeInt.\(\)I.0.jar`
true
stephen@srk31:~/proc/12418/jog/objects/1$ ./checkNumericalEquality.\(TI\)Z.0.jar
 author 1984
Invocation threw an exception: java.lang.NumberFormatException: For input string
: "Stephen Kell"
```

Next we call `getClass()`, which binds another object into the filesystem and tells us its path. (It will eventually disappear, as it is only transiently bound.) Then we show the use of subshells to compose two method calls. Notice how the string literal return value is prefixed by '`@`'.

```
stephen@srk31:~/proc/12418/jog/objects/1$ ./getClass.\(\)L.0.jar
/home/stephen/proc/12418/jog/objects/5
stephen@srk31:~/proc/12418/jog/objects/1$ `./getClass.\(\)L.0.jar`/getName.\(\)T
.0.jar
@uk.ac.cam.cl.srk31.jogfs.eval.Features$TestClass
```

After another method call returning an object, there are six objects bound.

```
stephen@srk31:~/proc/12418/jog/objects/1$ ./getNewObject.\(\)L.0.jar
/home/stephen/proc/12418/jog/objects/6
stephen@srk31:~/proc/12418/jog/objects/1$ ls ..
1  2  3  4  5  6
```

Next we set `testObjectField` to point to null again, using the special symbol '`$null`'.

```
stephen@srk31:~/proc/12418/jog/objects/1$ ln -nsf '$null' testObjectField; ls -l
 testObjectField
lrwxrwxrwx  1 stephen stephen 5 May  9 12:04 testObjectField -> $null
```

Next we test primitive fields. We use `echo `cat fieldName`` as a convenience, since it keeps the terminal display tidy by appending a newline. Likewise, `dd` allows us to see any error message from the `write()` call, which plain output redirection does not. Here there is an "Invalid argument" error on the first attempt, since the string written does not parse as an integer.

```
stephen@srk31:~/proc/12418/jog/objects/1$ echo `cat thisYear`
2005
stephen@srk31:~/proc/12418/jog/objects/1$ echo -n "2005.5" | dd of=thisYear
dd: writing `thisYear': Invalid argument
0+1 records in
0+0 records out
```

```
0 bytes transferred in 0.011629 seconds (0 bytes/sec)
stephen@srk31:~/proc/12418/jog/objects/1$ echo -n "2006" > thisYear
stephen@srk31:~/proc/12418/jog/objects/1$ echo `cat thisYear`
2006
```

To follow a link, we may use `cd`, but that will add link-name elements to the current path indefinitely, so we use `readlink` instead. The target object is a byte array, and has a special '`$`' accessor file which we view using the `hd` command.

```
stephen@srk31:~/proc/12418/jog/objects/1$ cd `readlink magic`; ls
$                   hashCode.()I.0.jar  notify.()V.0.jar     wait.()V.0.jar
equals.(L)Z.0.jar   index$              notifyAll.()V.0.jar  wait.(J)V.0.jar
getClass.()L.0.jar  length              toString.()T.0.jar   wait.(JI)V.0.jar
stephen@srk31:~/proc/12418/jog/objects/4$ hd $
00000000  4b 5e 6e 7f                                       |K^n.|
00000004
```

We update one of the elements individually, and see the result reflected in the array as a whole.

```
stephen@srk31:~/proc/12418/jog/objects/4$ cd index$; ls
0  1  2  3
stephen@srk31:~/proc/12418/jog/objects/4/index$$ echo -n "-128" > 1; hd ../$
00000000  4b 80 6e 7f                                       |K.n.|
00000004
```

Now we look at the classes directory. All the classes of the bound objects have been automatically bound. The '`ctl$`' directory is available for binding a new class or creating objects.

```
stephen@srk31:~/proc/12418/jog/objects/4/index$$ cd ../../../classes; ls
[B                java.lang.String
ctl$              uk.ac.cam.cl.srk31.jogfs.eval.Features$1
java.lang.Class   uk.ac.cam.cl.srk31.jogfs.eval.Features$TestClass
java.lang.Object
stephen@srk31:~/proc/12418/jog/classes$ cd ctl$; ls
bindClass.(T)V.jar  createObject.(TTT)L.jar
```

We bind the `java.lang.System` class and examine its contents. The targets of its `in`, `out` and `err` fields have been bound also.

```
stephen@srk31:~/proc/12418/jog/classes/ctl$$ ./bindClass.\(T\)V.jar @java.lang.S
ystem; ls ..
[B                java.lang.String$CaseInsensitiveComparator
ctl$              java.lang.System
java.lang.Class   uk.ac.cam.cl.srk31.jogfs.eval.Features$1
java.lang.Object  uk.ac.cam.cl.srk31.jogfs.eval.Features$TestClass
java.lang.String
stephen@srk31:~/proc/12418/jog/classes/ctl$$ cd ../java.lang.System; ls
```

```
arraycopy.(LILII)V.0.jar      load.(T)V.0.jar
currentTimeMillis.()J.0.jar   loadLibrary.(T)V.0.jar
err                           mapLibraryName.(T)T.0.jar
exit.(I)V.0.jar               out
gc.()V.0.jar                  runFinalization.()V.0.jar
getProperties.()L.0.jar       runFinalizersOnExit.(Z)V.0.jar
getProperty.(T)T.0.jar        setErr.(L)V.0.jar
getProperty.(TT)T.0.jar       setIn.(L)V.0.jar
getSecurityManager.()L.0.jar  setOut.(L)V.0.jar
getenv.(T)T.0.jar             setProperties.(L)V.0.jar
identityHashCode.(L)I.0.jar   setProperty.(TT)T.0.jar
in                            setSecurityManager.(L)V.0.jar
```

Method names are abbreviated such that they don't show the full class name of object-valued arguments, but calling them with no arguments will print the full signature.

```
stephen@srk31:~/proc/12418/jog/classes/java.lang.System$ ./setIn.\(L\)V.0.jar
Invalid arguments supplied. The full signature specification is:
(Ljava.io.InputStream;)V
```

Finally, we create a `java.lang.StringBuffer` object and are given its path in the filesystem.

```
stephen@srk31:~/proc/12744/jog/classes/java.lang.System$ cd ../ctl$
stephen@srk31:~/proc/12744/jog/classes/ctl$$ ./createObject.\(TTT\)L.jar @java.l
ang.StringBuffer @T '@"@foo"'
/home/stephen/proc/12744/jog/objects/11
stephen@srk31:~/proc/12744/jog/classes/ctl$$
```

4.1.3 Applications

Blank Slate

A simple 'blank slate' application has been created. This initially binds no objects, and is terminated by pressing a key in its controlling terminal. While it runs, a user of the filesystem may create and manipulate Java objects, hence opening up the functionality of Java libraries to interactive use, shell scripts and many other programs.

The usefulness of this application is slightly limited by memory and performance concerns (see Section 4.2), and by the few pieces of missing functionality mentioned in Section 4.1.1.

Interactive Application

Scripting of an interactive application is one immediate use of the system. Proof-of-concept was achieved by taking an example Swing application[1] and trivially modifying the source to start `JogFS` and bind a button (object 1) and label (object 2) into the filesystem. Success is demonstrated by Figures 4.2 and 4.3.

```
stephen@srk31:~/proc/23207/jog/objects/2$ ./getText.\(\)T.0.jar
@Number of button clicks: 0
stephen@srk31:~/proc/23207/jog/objects/2$ ../1/doClick.\(\)V.0.jar
stephen@srk31:~/proc/23207/jog/objects/2$ ./getText.\(\)T.0.jar
@Number of button clicks: 1
```

Figure 4.2: Interaction with a graphical application from the shell

Figure 4.3: Graphical application after shell interaction

Clearly, the system will be more useful once performance has been improved and certain other features implemented (notably "accessor methods as fields").

IPC Interface

Usefulness for the system as a general IPC interface is dependent on four factors.

- Performance: this is the primary limitation at present, and is discussed in Section 4.2.2.
- Correctness: the operations exposed should not unduly risk the consistency of the underlying data structure. We rely on the public interface of exposed objects to be robust and thread-safe, as discussed in Section 3.4.3.

[1] `http://java.sun.com/docs/books/tutorial/uiswing/learn/example2.html`

- Convenience: aside from consideration of correctness issues, use of the system requires minimal effort, making it highly convenient. Only small code changes to start-up, shut-down and bind objects into the filesystem are required.

4.2 Analysis

Here we analyse the memory overhead, performance and features of the system.

4.2.1 Memory Overhead

Factors affecting memory usage include:

- number of objects bound
- number of classes bound
- number of members (fields or methods) of each bound object or class
- degree of connectedness in the object graph

The total number of objects bound is most significant, since it is this which defines the ultimate scalability of the system. Conversely, the other factors either do not vary over much more than one order of magnitude (e.g. number of members), or are inherently subordinate to the number of objects (e.g. the number of classes).

A test program was created to discover the memory overhead of the system. It instantiates and binds a specific number of instances of a chosen class. It measures the size of the Java heap at start-up, after object creation, and after binding (running the garbage collector before each measurement). The size of the objects ('graph size') is estimated by the difference between the first two heap measurements, and the overhead of the binding data ('binding size') as the different between the second and third.

The program was run for three different classes, for various numbers of objects. The results are shown in Figure 4.4.

From the data, we conclude the following.

- The estimate of object graph size is poor. The increase is precisely linear with the number of objects, but has a fixed positive offset of around 6KB. This suggests a similar systematic error in the binding size estimate.

Class	number	est. graph size / b	binding size / b	est. per-object / b
`Object`	50	6584	372952	7460
	100	7184	646456	6460
	150	7784	894456	5960
	200	8384	1144520	5720
	250	8984	1392456	5570
`byte[100]`	50	10856	1778664	35600
	100	16656	3455424	35600
	150	22456	5133192	34200
	200	28256	6808576	34000
	250	34056	8481424	33900
`StringBuffer(100)`	50	18384	1114864	22300
	100	30584	2026616	20200
	150	42784	2991992	19900
	200	54984	3954248	19800
	250	67184	4929416	19700

Figure 4.4: Results of the memory overhead tests

- Assuming that the systematic error becomes insignificant as the number of objects tends to infinity, we can quantify the actual per-object overhead as the limit of the estimated overhead. This is shown in Figure 4.5.

The graph shows that the per-object overhead varies from around 5KB up to (and possibly beyond) 33KB. Since the byte array has the highest, we conclude that the large indexer directory is more costly than the `StringBuffer`'s large number of methods.

The overriding conclusion is that memory overhead is very high. Considering that a single `Object` occupies only twelve bytes of heap space, several kilobytes of binding overhead is not acceptable. Possible solutions are:

- Use a less stateful implementation. This option has the most potential for savings, at some cost in performance.
- Use memory-conserving data structures where possible (e.g. `ArrayLists`) in place of `HashMaps`).
- Rewrite the system in a language permitting more precise control of storage, such as C++.
- Redesign the system to offer less fine-grained access than at the level of objects, fields and methods.

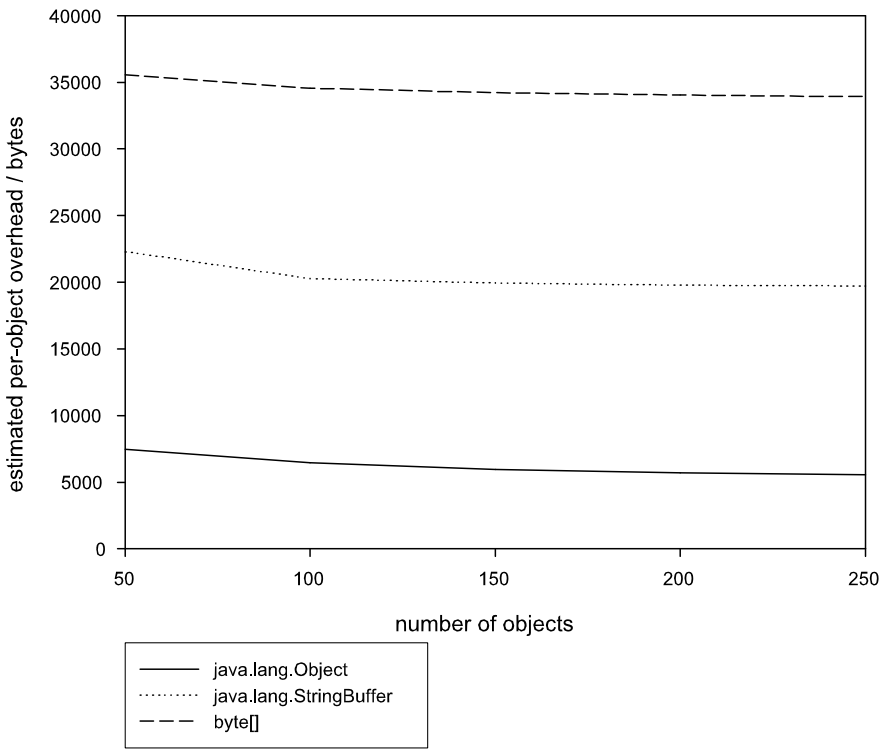

Figure 4.5: Graph of per-object overhead estimates with number of objects

Use of `jmp`, a Java memory profiler, on the test program in the `StringBuffer(100)` case shows that of the 6MB heap size, nearly 2MB is accounted for by character strings (of which no more than 500KB may be within the `StringBuffer` instances), while there is also 800KB of `FuseStat` instances and 700KB of `java.lang.reflect.Method` instances.

A less stateful approach would help with all the first two of these: by not storing directory listings or entity status across calls, but instead generating them from the object graph when required, far fewer character strings and `FuseStat`s would exist at any one time.

The large number of `Method` instances indicates that, unlike `Class` instances, a new copy of the object is generated at each reflection call. This has led to equivalent `MethodExecutable` entities in each object directory referencing its own copy of the `Method`. A cache would solve this problem.

4.2.2 Performance

Although performance was not a success criterion of the system, we briefly discuss some performance-related issues as a starting point for future work.

Read and Write Performance

Clearly, a conventional filesystem benchmark is not suitable for performance testing our system. Comparison to other systems, such as `/proc` or CORBA[17], would be instructive after further development. The potential maximum speed is likely to be limited by the `fuse`, JNI and Java reflection interfaces rather than the design of `jogfs` itself. Rewriting `jfuse` and `jogfs` in C++would enable gains from ahead-of-time compilation, less costly heap management and fewer cross-JNI calls.

Method Invocation

As described in Section 3.4.6, method invocation currently has poor performance owing to JVM start-up costs. Possible solutions are:

- Use a less resource-intensive (but more platform-dependent) executable format, such as a Unix shell script. This requires the file to be served or installed on a per-client-architecture basis.
- Use the request and result files directly, or redefine the semantics of method files to avoid the need for an executable (e.g. write arguments direct to the method file as a string, then read the results back from the same file). Since this is incompatible with the normal `exec()` interface, it is more difficult to interface with existing programs.

4.2.3 Comparison to Other Systems

The only system bearing useful comparison to `jogfs` is `rtafs`, the filesystem component of RTA [10]. We briefly compare the features, API and performance of the two systems.

Features

Unlike `jogfs`, `rtafs` does not expose procedure calls or recurse across data structures. It does, however, verify the validity of memory addresses (cf. Java garbage collection and type checking) using a memory debugging library. There is no `rtafs` feature without an analogue in `jogfs`.

API

- `rtafs` imposes a relational (rather than object-based) data model on the filesystem. To fit program data structures to this model requires a degree of

homogeneity (e.g. arrays of structures, forming tables), while `jogfs` works well with a heterogeneous collection of objects.

- `rta` requires each field and object to be specified at run-time by name, offset, length, type, base address, etc., while `jogfs` determines these automatically. The latter is only made possible by JVM support for reflection, which incurs memory overhead.

Performance

Although no rigorous testing has been done, the sample `rtafs` program exposes several tables while keeping total memory usage under 2MB. This is far better than the current `jogfs`, and most likely better even than an optimised version. Likewise, since accesses are done direct to C data structures, they are inevitably faster than Java reflection calls.

Once `jogfs` has underdone further development, a thorough comparison between the two systems will be warranted.

Conclusion

In conclusion, `rta` has a more complex API and fewer features, but currently offers better performance for the tasks it supports, and has an inherently lower memory overhead.

4.3 Observations on the Filesystem API

One motivation of the project was to expose ways in which the filesystem API is ill-suited to general-purpose applications (as opposed to storage only). We summarise these in three categories.

Storage-specific artifacts

Block counts, block sizes and other storage-derived quantities are used throughout the interface. These have little or no relevance to non-storage applications. Even to storage applications, it is wrong to consider free space constant within the same mount point: consider network-mounted hierarchies spanning multiple disks on the server.

These can be given dummy values – for example, `df` (to detail the usage of storage) shows running `jogfs` instances with spurious counts. However, dummy values can be problematic: for example, a program might suffer buffer overflow

if it reads more data than a dummy size indicates, but it may be expensive (or meaningless) to compute the size of the data ahead-of-time.

Special-Purpose Metadata

Away from storage applications, metadata such as creation timestamps and link counts is often meaningless, while meaningful metadata has no outlet. Even within storage applications, they may be inadequate: for instance, the standard access control metadata is limited to one owner and one group.

Extended attributes and access control lists [18] have been added to many filesystems to overcome this, but require an expanded interface. Preferable would be to incorporate metadata into the same namespace as filesystem entities themselves, as suggested by Miner [19] and others.

Complex Operations Not Supported

Symbolic links cannot be atomically updated (see Section 3.4.8); likewise, we cannot use `rmdir()` to unbind an object or `mkdir()` to create one. There is no support for general named procedures, and opaque catch-alls such as `ioctl()` are inadequate. A better interface might support named procedures together with a transaction-like system for complex operations.

Chapter 5

Conclusion

We conclude with summaries of successes, failures, and identified future work.

5.1 Successes and Failures

Most features of the project work acceptably well, particularly considering that the filesystem interface was not designed for anything like such an application. With a little work, the system will be extremely usable for scripting, debugging and reporting applications, as a very convenient user-space equivalent of `/proc`.

A notable problem is the poor performance of method invocation. No potential solution is known which doesn't compromise networkability or abandon the `exec()` call. This problem will certainly place strong direction on future work.

5.2 Future Work

Throughout the evaluation of the system, many avenues for future work were identified. These fall into three categories, each successively more difficult: adding missing features to `jogfs`, improving its performance, and designing an entirely new general-purpose filesystem interface.

It bears remembering that we have only considered Unix-like filesystems. Other operating systems may provide more generally-applicable filesystem abstractions, and any newly-designed system must learn from these. Since method invocation, a key feature of `jogfs`, is essentially limited in performance by omissions to the filesystem API, future work must unquestionably re-evaluate it.

The realisation of a unified general-purpose interface applicable to the hugely heterogeneous collection of inter-process interactions is a challenging but highly promising goal for the future.

Bibliography

[1] The Open Group, *Single UNIX Specification*, Version 3, 2004.

[2] B. Pawlowski et al, *NFS Version 3 Design and Implementation*, Proceedings of the Summer USENIX Conference, 1994.

[3] M. Satyanarayanan et al, *Coda: A Highly Available File System for a Distributed Workstation Environment*, IEEE Transactions on Computers, 1990.

[4] M. Gschwind, *FTP Access as a User-Defined File System*, ACM SIGOPS Operating Systems Review, 1994.

[5] T.J. Killian, *Processes as Files*, USENIX Summer Conference Proceedings, 1984.

[6] R. Faulkner and R. Gomes, *The Process File System and Process Model in UNIX System V*, USENIX Conference Proceedings, January 1991.

[7] E. Schrock, *A Brief History of* `/proc`, Weblog entry, available at `http://www.unixville.com/?q=node/view/46`, 2004.

[8] J. Train et al, *NetFS: Networking through the File System*, Tech. Rep. 2003-579, University of Southern California Information Sciences Institute, 2003.

[9] R. Pike et al, *Plan 9 from Bell Labs*, UKUUG Proceedings of the Summer 1990 Conference, London, July 1990.

[10] R.W. Smith, Run Time Access Introduction and API Specification, available at `http://www.runtimeaccess.com/`.

[11] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification, Second Edition*, Sun Microsystems, 1999.

[12] G. Hamilton et al, *JavaBeans API Specification*, Version 1.01, Sun Microsystems, 1997.

[13] Sun Microsystems, *Java Remote Method Invocation Specification*, version 1.4.2, available at `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[14] M. Szeredi, *Fuse: Filesystem in Userspace* (introduction, examples and source code), available at `http://fuse.sourceforge.net/`.

[15] Sun Microsystems, *Java Native Interface Specification*, version 1.3, available at `http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html`.

[16] Linux Programmer's Manual. Unix `man` pages, available from `ftp://ftp.win.tue.nl/pub/linux-local/manpages/`.

[17] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, available from `http://www.omg.org/`.

[18] A. Gruenbacher, *POSIX Access Control Lists on Linux*, USENIX Annual Technical Conference, 2003.

[19] G. Miner, *Everything is a File*, manuscript, UMM Morris, 2004.

Appendix A

Implementation of `libjfuse-jni`

We begin by outlining the `fuse` API, then discuss the individual features and the problems they present.

The `fuse` Interface

Figure A.1 shows the C API above which our filesystem is implemented. Central is the `fuse_operations` structure, which consists of a set of callbacks reminiscent of a Unix VFS interface. Most of the callbacks correspond directly to a Unix system call, although always operating on pathnames rather than file descriptors. (File descriptors are issued internally by `libfuse`; the pathnames passed in are always relative to the root of the `fuse` filesystem.) The other differences are detailed in `fuse.h` [14].

Our Java binding defines the following analogues for the various parts of the interface.

- `FuseOperations`, an interface specifying a method for each of the `fuse_operations` callbacks.
- `FuseMain`, a class implementing `Runnable` and corresponding roughly to the `fuse_main()` function. It also provides static helper methods wrapping `fuse_get_context()` and `fuse_exit()`.
- `FuseStat` and `FuseStatFs`, classes corresponding to the `structs` `stat` and `statfs`.
- `FuseDirectoryIterator`, an intermediary class representing the contents of a directory, used to implement the `getdir()` callback.
- Various Java constants analogous to the standard C symbolic constants for file types, open modes, permission bitmasks and similar.

```
struct fuse_operations {
    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 *, int);
    int (*read)        (const char *, char *, size_t, off_t);
    int (*write)       (const char *, const char *, size_t, off_t);
    int (*statfs)      (const char *, struct statfs *);
    int (*flush)       (const char *);
    int (*release)     (const char *, int);
    int (*fsync)       (const char *, int);
    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 *);
};
struct fuse_context {
    struct fuse *fuse;
    uid_t uid;
    gid_t gid;
    pid_t pid;
};
void fuse_main(int argc, char *argv[], const struct fuse_operations *op);
void fuse_exit(struct fuse *f);
struct fuse_context *fuse_get_context(void);
```

Figure A.1: The `fuse` interface (relevant parts only)

- Various Java exception classes analogous to the standard C error values which may be returned by the callbacks.
- `NativeHelpers`, a class defining some helper methods (see section A).

Restrictions of the Interface

Since our filesystems will be implemented in Java, we would like a single set of C callback implementations to be able to interface to any number of Java filesystem implementations. Unfortunately, as the C functions' signatures do not provide any way of specifying which Java object to invoke methods on, a global variable

or function call is the only way for these functions to get a reference to the correct target object.

To avoid the complexities of demultiplexing calls in this way, instead the number of simultaneously running filesystems is limited to one (per process). A global pointer to the target object is set immediately before `fuse_run()` is called, and retains the same value until the filesystem is stopped.

Start-up and Shut-down

The filesystem is started by calling the `run()` method on an instance of `FuseMain`. A lock is used to ensure that no more than one `FuseMain` is running at one time.

The event loop is started by calling `fuse_main` function, which also mounts the filesystem and installs signal handlers to unmount it on termination. It takes textual arguments, as it is intended to be wrapped by a `main()` function, but we call it by composing argument strings based on settings held in the `FuseMain` object.

Shut-down is also provided by a native method of `FuseMain`, `stop()`, which calls `fuse_exit()` .

Callback Implementations

The code for each callback has the same general form:

1. Convert the pathname argument to a Java String object.
2. Convert any other arguments as appropriate.
3. Invoke the corresponding method on the `FuseOperations` object.
4. Test whether an exception was thrown by the method.
5. If so,
    - Set the callback's return value appropriately for the class of exception which occurred.
6. Otherwise,
    - Convert the method's return value as appropriate.
    - Set the return value either to zero or to whatever the callback specification dictates. (This is usually zero, but for `read()` or `write()` is the number of bytes read or written.)

7. Deallocate any memory allocated (including the String object created) and return.

Marshalling and Translation

Marshalling data between Java and C code is a major role of `libjfuse-jni`. There are several cases to consider.

- Primitive C and Java types. Since each primitive Java type has a C definition (with clear correspondence to a native C type), this is straightforward. Wider integer types are often used on the Java side, to work around the lack of unsigned types.

- Special C types such as `mode_t`, `uid_t` and `time_t`. These are aliases for a primitive C type and are mapped to a suitable Java primitive type similarly.

- C structures. Simple and/or infrequently-used structures are disassembled into components (e.g. `struct utimbuf` into two `jlong`s). Others (`struct stat` and `struct statfs`) have an analogous Java class defined and are copied field-by-field.

  This copying incurs some performance penalty, but given that the structures are accessed more frequently on the Java side than on the C side, the penalty is far less than that of the alternative approach (using a C structure throughout, with Java accesses implemented by native methods).

- Strings and buffers. C buffers and character strings can be passed by using a JNI call to create a Java object (a byte array or `String`) from a char pointer. These calls avoid unnecessary copying. The object then is freed after the method call. In the other direction, JNI provides a similar call to get a pointer to the object's underlying data; we must copy this data into the preallocated buffer passed as argument (in calls such as `write()` or `readlink()`).

Error Reporting

In Java, error conditions are signalled using exceptions, whereas in C it is conventional to return error codes. Since our Java implementation will throw exceptions if an error occurs, when faced with an exception we must select the appropriate error code.

The `fuse` library uses the standard C `errno` error codes, except that they are negated and used as the return value. This allows a positive return value to indicate something useful about success (e.g. the number of bytes read).

If the Java method throws an exception, we determine the return code solely from the exception's class. After each method call, we check for an exception. If one was thrown, we test the exception's type against a list of expected types, and look up the error code corresponding to the first match. (The 'test' uses a JNI call to match subclasses.) If none of these matches, a generic error code is selected (conventionally `EPERM`) and a more detailed description is written on the standard error for debugging.

Native Helper Methods

The `fuse` system currently runs only on Linux. Although it could easily be ported to another Unix, it is unlikely that it would be ported to a non-Unix-based OS. As a result, Unix-specific system information useful when defining `fuse` filesystem is not available from the standard Java libraries. Consequently. the `NativeHelpers` class provides a set of native methods to get the current process ID, effective user ID and effective group ID, and for testing group membership.

Appendix B

Access Control in `jfuse`

Here we describe the implementation of access control in `jfuse`, referring back to Section 3.3.

Partitioning

As with the operations themselves, we partition permissions for the various operations into two sets: one which applies to any named object (e.g. `READ`, `WRITE`, `CHMOD`, `SET_TIMESTAMP`, and one which applies only to directories (e.g. `DIR_LIST`, `DIR_REMOVE_ENT`). The `FuseActionConsts` class defines bitmask-style constants for both of these abstract sets.

Parameters

In POSIX, permission may depend not only on the calling user, the operation and its target entity, but on an argument value: for example, to change the group of a file, a user must not only own the file but be a member of the target group. Similarly, removing an entry from a 'sticky' directory is possible only for a user who owns either the file or the directory.

Accordingly, there are two methods for checking permissions – one for 'entity' operations and one for 'directory' operations. Both have the same form: they take a bitmask of the permissions required, and an extra argument specifying the 'parameter' of the operation. If permission is granted, the method returns; otherwise, a `PermissionDeniedException` is thrown.

For entity operations such as `CHGRP`, the parameter is an integer (e.g. the intended new group ID). For 'directory' operations such as `DIR_REMOVE_ENT` the parameter is the name of the target entry. The argument is ignored if not needed for the checks requested; if more than one check requiring an argument is requested (by ORing of their bitmask constants), then the argument must be the

same for both (or else the check must be split into two successive calls), but this is rare.

Privileged Callers

We have so far ignored the issue of privileged processes, such as those whose effective user ID is that of the `root` user. Calls from these should always be granted the necessary permission. Privilege is highly platform-dependent. Linux uses 'capabilities', a set of fine-grained privileges which can be individually granted to processes, as a generalisation of the root user ID.[1] This fine-grained approach matches up well with our abstract set of operations, but currently capability-checking is not implemented. However, the root user (`uid` 0) is always granted permission.

[1]These should not be confused with the unforgeable resource names called 'capabilities' used by some access control systems.

Appendix C

Details of Method Invocation

Here we cover some detail of method invocation omitted from Section 3.4.6 for the sake of brevity.

Request Processing

When the client process closes the method request file, the data written is deserialized into a `MethodCallDescriptor` and passed to the `request()` method of `MethodResultDescriptor`. It is this method which performs the method invocation. Execution proceeds as follows.

1. Locate the `MethodExecutable` instance corresponding to the method being invoked. This is named in the `MethodCallDescriptor`.

2. Get a reference to the `Method` instance underlying the `MethodExecutable` instance. The `Method` class is part of the Java reflection API.

3. Convert argument strings into `Objects`, using the argument types specified by the `Method`. The `StringMarshaller` class provides a convenient interface to the necessary conversions.

4. Call `invoke()` on the `Method`, using the containing directory's target object (which may be null, for a class directory).

5. If any errors occurred, encode these as a string; otherwise, encode the result as a string. If the result is an object, it will be bound 'transiently' into the filesystem (see Section 3.4.11). In any case, create a `MethodResultDescriptor` to represent the outcome.

6. Serialize the descriptor and save the bytes locally (under the calling process ID) for the client process to read.

Reading the Response

The client may then `open()` the result file and read the serialized `MethodResultDescriptor`. If no request has been made by the calling process, the file appears empty. Once the result file is closed, that process's results are discarded. Every request always generates a result, even if the request is garbage and cannot be deserialized, though the result may encode only an error message. Results are only kept for a finite time – if the client does not open the result file, the result will be discarded after two minutes.

Appendix D

The `PartialByteSequence` Class

Here we describe `PartialByteSequence`, a utility class used in the implementati of primitive fields (see Section 3.4.7).

Overview

The `PartialByteSequence` class represents a numbered sequence of bytes, similar to an array, but with some ranges undefined. These represent parts of the 'file' not yet modified by any `write()` operation. The structure is 'flattened' into a byte array by providing an array of 'backing' data, which fills in the undefined values but is overridden by any ranges stored in the `PartialByteSequence`.

Implementation

Internally, `PartialByteSequence` is implemented as a red-black binary tree (an instance of the `TreeSet` standard class). Each node in the tree is called a 'span', is labelled with an offset within the byte sequence, and contains an array of bytes. Updates to the structure are coded so that adjacent or overlapping spans are merged. It is often necessary to get all spans starting before or after a known offset, which a red-black tree does in $O(\log n)$ time (where n is the number of spans in the tree), comparing well with the $O(n)$ cost of an unsorted list or hash table.

Appendix E

Sample Source Code

E.1 MethodResultFile.java

```
/* MethodResultFile.java
 *
 * Defines the class representing the method result file and performing
 * method invocations.
 *
 * Created on Mar 20, 2005
 */
package uk.ac.cam.cl.srk31.jogfs;

import java.lang.reflect.*;
import java.util.*;

import uk.ac.cam.cl.srk31.jfuse.*;
import uk.ac.cam.cl.srk31.util.*;

/**
 * Defines the file used to communicate method results to clients,
 * and is also responsible for invoking the method.
 * @author Stephen Kell
 */
public class MethodResultFile extends ProcessMultiplexedFile
{
    // the object directory containing us
    private ReflectionContextDirectory objDir;

    // map process IDs to pending results
    private Map pendingData;

    /**
     * Constructs a method result file in the given object context
     * (parent).
     * @param objDir Reference to the containing ReflectionContextDirectory.
     */
    public MethodResultFile(ReflectionContextDirectory objDir)
    {
        super(FuseStat.REG | 0440, NativeHelpers.getUserId(),
            NativeHelpers.getGroupId());
        this.objDir = objDir;
        pendingData = new HashMap();
    }
```

```
void request(MethodCallDescriptor mcd, int pid)
{
    /* Performs a method invocation.
     *
     * This is called by MethodRequestFile.release() with the
     * lock held on mrf, so we needn't worry about concurrent
     * invocations. */

    byte[] resultSerial; // the result data

    // locate the method, by looking up the MethodExecutable in the directory
    FuseEntity ent = objDir.getEntry(mcd.getMethodName());
    MethodExecutable methodFile;

    if (ent instanceof FuseSymlink)
    {
        // follow symbolic links
        methodFile = (MethodExecutable)
            ((FuseSymlink) ent).getConcreteTarget();
    }
    else methodFile = (MethodExecutable) ent;

    Method m = methodFile.getMethod();

    try
    {
        // decode the arguments
        Object[] argObjs = decodeStringArguments(
            mcd.getArgs(), m.getParameterTypes());

        // execute the method
        Object result = m.invoke(objDir.getTarget(), argObjs);

        // encode the result
        resultSerial = encodeResult(result);
    }
    catch (NoSuchFileOrDirectoryException ex)
    {
        resultSerial = encodeError(
            "Invalid argument: one or more "
            + "specified object paths was invalid.");
    }
    catch (InvocationTargetException ex)
    {
        resultSerial = encodeError(
            "Invocation threw an exception: "
            + ex.getTargetException().toString());
    }
    catch (IllegalAccessException ex)
    {
        resultSerial = encodeError("Method is inaccessible.");
    }
    catch (IllegalArgumentException ex)
    {
        // arguments invalid -- try to be helpful
        String errorString;
        if (mcd != null)
        {
            errorString = ReflectionContextDirectory
                .getSignatureString(m, true);
        }
        else errorString = "(signature not available)";

        resultSerial = encodeError(
```

```
                "Invalid arguments supplied. "
                + "The full signature specification is:\n"
                    + errorString
                    );
        }
        catch (ExceptionInInitializerError ex)
        {
            resultSerial = encodeError(
                "An exception was triggered in "
                + "an initialization called by the invocation.");
        }
        catch (NullPointerException ex)
        {
            resultSerial = encodeError("Object reference is no longer valid.");
        }

        // put the result in the result map
        postResult(pid, resultSerial);
    }

    void postErrorResult(String message, int pid)
    {
        postResult(pid, encodeError(message));
    }

    private void postResult(int pid, byte[] serial)
    {
        pendingData.put(new Integer(pid), new SimpleByteSequence(serial));
    }

    /* (non-Javadoc)
     * @see uk.ac.cam.cl.srk31.jogfs.ProcessMultiplexedFile#initData(int)
     */
    protected ByteSequence initData(int pid)
    {
        return getPendingData(pid);
    }

    private ByteSequence getPendingData(int pid)
    {
        // if data is pending for us, grab it
        ByteSequence pending = peekPendingData(pid);
        if (pending == null)
        {
            // appear as an empty file
            pending = new SimpleByteSequence();
        }
        else
        {
            pendingData.remove(new Integer(pid));
        }
        return pending;
    }

    private ByteSequence peekPendingData(int pid)
    {
        ByteSequence pending = (ByteSequence)
            pendingData.get(new Integer(pid));
        if (pending != null)
        {
            System.err.println("Found data pending for pid " + pid);
        }
        else
        {
```

```
            System.err.println("No data pending for pid " + pid);
        }
        return pending;
    }

    /* (non-Javadoc)
     * @see uk.ac.cam.cl.srk31.jogfs.TagMultiplexedFile#getStatus(int)
     */
    public FuseStat getStatus(int tag)
    {
        FuseStat stat = super.getStatus(tag);
        if (stat.getSize() == 0)
        {
            // check for pending data, and set size accordingly
            ByteSequence pending = peekPendingData(tag);
            if (pending != null)
            {
                int size = pending.getSize();
                if (size > 0) stat.setSize(size);
            }
        }
        return stat;
    }

    /* (non-Javadoc)
     * @see uk.ac.cam.cl.srk31.jogfs.TagMultiplexedFile#read(int, long, long)
     */
    public byte[] read(int tag, long count, long off)
    {
        checkPermission(READ);
        ByteSequence data = getData(tag);
        int returnLength = Math.min((int) count, data.getSize() - (int) off);
        return data.read((int) off, returnLength);
    }

    private Object[] decodeStringArguments(String[] args, Class[] cs)
    {
        if (args.length != cs.length)
        {
            throw new IllegalArgumentException("Expected " + cs.length
                + " arguments, " + args.length + " given.");
        }
        Object[] objs = new Object[args.length];
        for (int i = 0; i < args.length; i++)
        {
            objs[i] = StringMarshaller.decode(args[i], cs[i]);
        }
        return objs;
    }

    private byte[] encodeResult(Object o)
    {
        MethodResultDescriptor desc = new MethodResultDescriptor(
            StringMarshaller.encode(o), "");
        return desc.getSerializedRepresentation();
    }

    private byte[] encodeError(String s)
    {
        return (new MethodResultDescriptor("", s))
            .getSerializedRepresentation();
    }
}
```

E.2 AbstractObjectReferenceField.java

```
/* AbstractObjectReferenceField.java
 *
 * Contains the main logic used to define object-reference-like
 * symbolic links.
 *
 * Created on 02-Apr-2005
 */
package uk.ac.cam.cl.srk31.jogfs;

import uk.ac.cam.cl.srk31.jfuse.*;

/**
 * Defines a featureful abstract base for symbolic links
 * representing an object reference.
 *
 * @author Stephen Kell
 */
public abstract class AbstractObjectReferenceField extends MutableSymlink
{
    // the target path used to denote a null reference
    protected static final String NULL_REFERENCE_PATH = "$null";

    // the directory from which the link target is looked up
    private final FuseDirectory anchor;

    // the context directory containing us
    protected final ReflectionContextDirectory context;

    // the object directory we point to
    private ObjectReflectionDirectory targetContext;

    // the objects directory
    protected final ObjectsDirectory objects;

    // relative path of the objects directory
    private final String objectsPrefix;

    // whether to strongly-bind the target object
    private boolean strong;

    /**
     * Constructs an instance for a given link anchor and
     * reflection context, optionally with strong target binding.
     * @param anchor the directory from which the link path will
     * be evaluated
     * @param context the directory of the link's logical
     * containing reflection context (e.g. the defining object
     * or class)
     * @param strong whether objects referred to by this link
     * should be strongly-bound, i.e. have a strong reference
     * kept to them
     */
    public AbstractObjectReferenceField(FuseDirectory anchor,
            ReflectionContextDirectory context, boolean strong)
    {
        super(NativeHelpers.getUserId(), NativeHelpers.getGroupId(),
                anchor,
                NULL_REFERENCE_PATH); // creates a dangling link
        this.anchor = anchor;
        this.context = context;
        this.strong = strong;
```

```
        objects = JogFS.getRoot().objects();
        objectsPrefix = getObjectsPrefix(anchor);

        // we have a null target for now
        targetContext = null;
    }

    /* Short-cut to this functionality, as we use it often. */
    protected static int getObjectIdentity(Object o)
    {
        return JogFS.getRoot().objects().identities().getIdentity(o);
    }

    /* Get the implied path of an object. Trivial, but convenient. */
    protected final String getRelativePathByIdentity(int ident)
    {
        return objectsPrefix + Integer.toString(ident);
    }

    /**
     * Gets the relative path, from a given anchor directory,
     * to the filesystem's objects directory.
     * @param anchor the directory to which the path returned should
     * be relative
     * @return the relative path to the objects directory from the specified
     * anchor
     */
    public static String getObjectsPrefix(FuseDirectory anchor)
    {
        ObjectsDirectory objects = JogFS.getRoot().objects();
        String objectsPrefix = "";
        FuseEntity ent = null;
        JogRootDirectory root = JogFS.getRoot();

        // iterate up to the root, or objects dir, whichever comes first
        do
        {
            objectsPrefix = objectsPrefix + ".." + FuseDirectory.SEPARATOR;
            ent = anchor.findEntity(objectsPrefix);

        } while (ent != objects && ent != root);

        // if we hit the objects dir, return the path directly,
        // otherwise append the path to objects dir from the root
        if (ent == objects) return objectsPrefix;
        else return objectsPrefix + JogRootDirectory.OBJECTS_DIR_NAME
            + FuseDirectory.SEPARATOR;
    }

    /**
     * Update the link target to the new path specified.
     * @param newTargetPath the new link target
     */
    protected final void updateTargetPath(String newTargetPath)
    {
        // save the old path and context, in case update fails
        String oldTargetPath = targetPath;
        Object oldTargetObject;
        try
        {
            oldTargetObject = getTargetObject();
        }
        catch (NoSuchFileOrDirectoryException ex)
        {
```

```
            // old target wasn't valid
            oldTargetObject = null;
        }
        ObjectReflectionDirectory oldTargetContext = targetContext;

        Object newTargetObject; // the new target object
        ObjectReflectionDirectory newTargetContext; // its context

        // if we're setting the reference to null, new object/context are trivial
        if (newTargetPath.equals(NULL_REFERENCE_PATH))
        {
            newTargetObject = null;
            newTargetContext = null;
        }
        else
        {
            try // looking for target object directory
            {
                newTargetObject = ((ObjectReflectionDirectory)
                    anchor.findEntity(newTargetPath))
                        .getTarget();
            }
            catch (NoSuchFileOrDirectoryException ex)
            {
                // target directory doesn't exist yet -- get object ID from path
                int lastSep = newTargetPath.lastIndexOf(FuseDirectory.SEPARATOR);
                String identityString = newTargetPath.substring(lastSep + 1);
                int ident;
                try
                {
                    ident = Integer.parseInt(identityString);
                }
                catch (NumberFormatException newEx)
                {
                    throw ex;
                }
                newTargetObject = objects.identities().getObject(ident);
                // newTargetObject now definitely isn't null
            }

            // get the context through ObjectsDirectory, so it can do the binding
            newTargetContext =
                objects.getObjectContext(
                        newTargetObject,
                        this
                    );

            // optionally create strong reference
            if (strong) objects.bindStrong(newTargetObject);
        }

        try
        {
            // swap over
            synchronized(this)
            {
                updateUnderlyingReference(newTargetObject);
                targetPath = newTargetPath;
                targetContext = newTargetContext;
                getStatus().setSize(targetPath.length());
            }
        }
        catch (RuntimeException ex)
        {
```

```
            /* If anything goes wrong, forget about it. There's no
             * need to revert the underlying reference, since it's the
             * only statement above that can throw an exception, and
             * if it does throw, it does so before making any update. */

            targetPath = oldTargetPath;
            targetContext = oldTargetContext;
            objects.releaseObjectContext(newTargetObject, this);
            if (strong) objects.unbindStrong(newTargetObject);

            // re-throw the exception, as it still needs handling
            throw ex;
        }

        // if we had a target previously, release our bindings on it
        if (oldTargetObject != null)
        {
            objects.releaseObjectContext(oldTargetObject, this);
            if (strong) objects.unbindStrong(oldTargetObject);
        }
    }

    /**
     * Update the underlying object reference underlying to the
     * specified value. Any exception must be thrown before the
     * update.
     * @param o the new value of the object reference
     */
    protected abstract void updateUnderlyingReference(Object o);

    /**
     * Get the underlying object reference.
     * @return the underlying object reference
     */
    protected abstract Object getUnderlyingReference();

    /* (non-Javadoc)
     * @see uk.ac.cam.cl.srk31.jfuse.FuseEntity#getLinkTarget()
     */
    public String getLinkTarget()
    {
        /* Here we must override the readlink() behaviour to
         * check for updates to the underlying object reference
         * (i.e. updates done in Java code, not through the
         * filesystem). */
        int oldTargetIdent;
        int newTargetIdent;
        Object newTarget = getUnderlyingReference();

        // can't give null to getIdentity, so test specially
        if (newTarget != null) newTargetIdent =
            objects.identities().getIdentity(newTarget);
        else newTargetIdent = 0;

        if (targetContext != null) oldTargetIdent =
            targetContext.getTargetIdentity();
        else oldTargetIdent = 0;

        // if there's been a change,
        if (newTargetIdent != oldTargetIdent)
        {
            // update the target path
            updateTargetPath(getPathFromObject(newTarget));
        }
```

```
        // now proceed as before
        return super.getLinkTarget();
    }

    /**
     * Returns the path from the current directory to the
     * specified object, assuming it (or will be) bound
     * into the filesystem.
     * @param o the object whose path to get
     * @return the relative path of the object
     */
    protected final String getPathFromObject(Object o)
    {
        if (o == null) return NULL_REFERENCE_PATH;
        else
        {
            int ident = getObjectIdentity(o);
            return getRelativePathByIdentity(ident);
        }
    }

    /**
     * Gets a reference to the target object, if there
     * is one.
     * @return the target object
     * @throws NoSuchFileOrDirectoryException if the target
     * path is invalid (or represents the null reference)
     */
    public final Object getTargetObject()
    {
        FuseEntity ent = this.target();

        // hopefully the target is an *object* directory
        return ((ObjectReflectionDirectory) ent).getTarget();
    }

    /* (non-Javadoc)
     * @see uk.ac.cam.cl.srk31.jfuse.FuseEntity#finished()
     */
    public void finished()
    {
        if (targetContext != null)
        {
            // release our binding on the target object
            objects.releaseObjectContext(targetContext.getTarget(), this);
        }
        super.finished();
    }
}
```

Appendix F

Project Proposal

Computer Science Tripos Part II Project Proposal

A General-Purpose Synthetic Filesystem

Stephen Kell, Christ's College

Originator: Stephen Kell

20 October 2004

Special Resources Required

PWF account with 250MB file space
Use of personal Linux machine

Project Supervisor: Douglas Santry

Director of Studies: Dr M. P. Fiore

Project Overseers: Dr S. W. Moore & Dr P. Liò

Introduction

A synthetic filesystem differs from a conventional filesystem in that its data is based not on a disk, but in some other space. This space might be the kernel (as with Linux's *procfs*), a user process, or maybe something even more abstract, made available through a network protocol or other I/O mechanisms.

These synthetic filesystems make use of the filesystem API as a general-purpose interface for accessing data. However, only a few synthetic filesystems (such as *procfs*) are widely used. This project will develop a general-purpose synthetic filesystem for Java programs. This filesystem will be capable of exposing an arbitrary Java object graph to the filesystem under Linux, while enabling all or most conventional programmatic features of Java objects to be invoked using the standard Unix filesystem API.

Required Resources

1. PWF account with 250MB file space
2. The use of my own Linux machine (450MHz K6-3, 128MB memory, 6GB disk) for development tools and (possibly) temporary storage

Back-ups will be provided by the existing PWF back-up regime. Should my personal Linux machine fail, a PWF Linux machine would suffice *except* that privileged access would be required, for the insertion of a special kernel module.

Starting Point

A pre-existing user-space filesystem library such as *fuse* will be used to provide the low-level filesystem implementation and kernel interface.

I already have a reasonable knowledge of C and Java, but familiarisation with the libraries and interfaces being used will be required.

Project Description

The aim of the project is to demonstrate the use of the Unix filesystem interface as a general-purpose interface for accessing structured data and code. It will then be possible to identify some of the limitations and weaknesses it has in this capacity.

Taking a Java object graph as a general representation of structured data with accompanying code, a synthetic filesystem will be implemented to expose this. Java programs will be able to bind an object to a directory in the filesystem, within which the object's fields and methods will be exposed as data files, symlinks to other object-directories, and executables. (Special consideration will be given to particular fields and methods such as arrays, get- and set-methods and collection indexer methods, in order to represent them in as natural and usable a way as possible.)

The essential components of the filesystem implementation are

- C code to interface between the user-space filesystem library and all Java-language parts of the implementation, making heavy use of the Java Native Interface (JNI).
- The basic implementation of object graph traversal and reading/writing of data. This could be implemented in C using JNI, or in Java using reflection. The latter option is preferred for maintainability reasons, particularly since performance is not considered important for the purposes of this project.
- A marshalling layer to perform conversions between the filesystem's untyped byte-based representation of data and Java's type system.
- Code to support method invocation. Since conventionally, executing a file is not part of the filesystem interface, we have no facility to define how to invoke executables on our synthetic filesystem. Rather, the host OS will attempt to read the executable as data in a known executable file format, and then execute it according to that format. This behaviour will need to be worked around.
- A Java application-level interface to allow applications to bind objects to directories. Few or no other operations should need to be supported.

In addition to these core components, it would be helpful to provide

- A useful demonstration application, from which an object graph is bound to the filesystem and can be usefully manipulated using the filesystem API (perhaps from a shell) while the application continues to run. An interactive graphical application would be well-suited to this. The application need not be written specially, only modified very slightly to expose relevant objects to the filesystem.

Success Criteria

On successful completion, the filesystem will support

- Representation of an object graph as a directory tree with symbolic links.
- Getting and setting of primitive-type field values using the filesystem API. This could be tested by getting and setting values from the shell using the *cat* and *echo* commands.
- Following and setting of object-reference field values using the filesystem API. This could be tested by following references (links) using the *cd* command and updating references using the *ln -s* command.
- Method invocation on a local system. It should be possible for a local process to call *exec()* on a method in some object's directory, supplied with any parameters in a plain-text format, and have that method execute on that object in a new JVM thread.
- Exposition of get- and set- methods as non-executable files, readable and/or writable as appropriate.
- Exposition of indexing methods of arrays and collections using subdirectories.
- Preservation of thread-safety. All object accesses made through the filesystem API should be thread-safe if the Java interface to that object is thread-safe. Since the filesystem interface will access the Java objects through Java code, this should be enforced by the JVM as usual.

Support for the following functions is desirable but should not be considered a primary success criterion:

- Object creation. The filesystem interface clearly is not adequate here, since when creating a directory in our synthetic filesystem there is no way to specify or infer the class of the object which the directory is to represent. There may still be a way to construct an object, either in multiple steps or using a method call (mapping to the relevant constructor).
- Object reclamation. As always, unreachable objects should be reclaimed by the JVM's garbage collector, though we note that reachability testing cannot include account for symbolic links into the filesystem from outside.

- Method invocation from a remote system, e.g. over NFS. This presents special problems because of the platform-dependent nature of executables, but some limited solutions may be possible.

Plan of Work

The following plan assumes roughly twenty weeks of active work on the project and dissertation, spanning from late October to the end of April and including breaks for Christmas and Easter. This conservative schedule should allow room in case of unforeseen difficulties.

1. Familiarisation period, initial organisation and design work. [2 weeks]
2. Initial implementation, allowing binding of a single object and listing of its fields and methods (only). [1 week]
3. Basic implementations of marshalling layer, and reading/writing of fields including object references. [4 weeks]
4. Support for local method invocation. [2 weeks]
5. Support for get- and set- methods and indexing subdirectories. [2 weeks]
6. Demonstration application. [1 week]
7. Testing and review of success criteria. [2 weeks]
8. Design, implementation and testing of additional features, if on schedule. [2-3 weeks]
9. Conclusions and writing up. [3-4 weeks].