Robert McQueen

Creating, Reverting & Manipulating Filesystem Changesets on Linux

Part II in Computer Science

Corpus Christi College

May 19, 2005

Proforma

Name:	Robert McQueen
College:	Corpus Christi College
Project Title:	Creating, Reverting & Manipulating Filesystem Changesets on Linux
Examination:	Part II, Computer Science Tripos, July 2005
Word Count:	11980 (approx)
Project Originator:	Peter Sewell
Supervisor:	Peter Sewell

Original Aims of the Project

To design a format for representing the changes that can be made to the filesystem on a Linux system and to implement a monitoring system to observe other processes and record the filesystem modifications they make in this format. A utility for reverting the changes described in this format or re-apply them to the same or a different system should also be implemented. The completed system should allow the action of installing, upgrading or removing software using a package manager to be completely reverted.

Work Completed

A monitoring system has been implemented which is capable of monitoring any arbitrary process without requiring it or the system's kernel to be modified, and outputs a log of the filesystem changes made in an easily comprehensible format. The monitoring system has almost complete coverage of methods by which the filesystem can be changed. In comparison to the monitoring system, the means for applying or reverting the monitored changes should be reasonably simple. A possible technique has been described but not implemented.

Special Difficulties

Effective and complete monitoring of changes requires detailed knowledge of various relevant Unix system calls. These turned out to be more complex and more numerous than expected. As a result, understanding their behaviour and writing the monitoring software took longer than anticipated.

Declaration

I, Robert McQueen of Corpus Christi 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

Acknowledgements

I'd like to thank Dr. Sewell for his patient support and guidance throughout the implementation of this project and production of this document, and also for the original inspiration for this project. I'd also like to thank the developers of the Linux kernel for making such an amazing (albeit somewhat quirky) system freely available to all.

Chapter 1

Introduction

Most modern Linux distributions manage the software installed on the system by using binary *packages* of pre-compiled software. Packages typically consist of two parts, the files (be it executable code, data or documentation) of the software itself and various metadata such as a description of the software, the version provided and details of other packages which must be installed to make the software usable (called dependencies, such as shared libraries). A package manager (such as `dpkg` in Debian GNU/Linux[1] and its derivatives, and `rpm` in Red Hat Linux[2] and its derivatives) allows you to install, remove and upgrade these packages. Higher level tools such as `apt` for `dpkg` or `yum` for `rpm` exist to automate downloading lists of the packages available from your distribution, installing selected packages and their dependencies or upgrading all of the packages currently installed on your system. As a consequence of this automation, use of these higher-level tools can cause major changes to a large proportion of the software currently installed, introducing bugs and instability, potentially leaving parts of the system unusable.

The original motivation for the project was to provide an undo mechanism for operations with the `dpkg` and `apt` package management tools on a Debian based system. After a failed or unwanted upgrade or package installation, the versions and configuration of the software on the system could be returned to their previous state. A naïve implementation of such a mechanism might simply install older versions of the relevant packages in an attempt to restore the system to its previous state. However, downgrading packages is not officially supported by Debian packages, and there are reasons why it is not practical to rely on being able to do. This will be detailed in the next section.

Naturally an undo mechanism which only sometimes works is undesirable, so a lower-level approach was considered instead. The solution proposed by this

[1]`http://www.debian.org/`
[2]`http://www.redhat.com/`

project is to provide a mechanism by which to monitor the changes made to the filesystem by a particular group processes. This information can be represented as a changeset which records the original status of any files or directories which were modified, and the changes that were made to them. These changesets may then be reverted or applied to an identical or similar filesystem. This effects not only an undo mechanism for the changes made by the package manager but also a useful mechanism for a variety of other purposes such as monitoring the installation of software not available in a packaged form, reverting the effects of executing potentially untrusted code, or the application of the same changeset to a number of similar systems.

There are existing programs, such as `checkinstall`[3], which aim to record the modifications to the filesystem made when a particular piece of software is installed. Although recent versions include support for backing up files which have the potential to be modified, the emphasis is on producing a list of files which are created so that they can be made into a package and removed at a later date, rather than implementing the ability to repeat or reverse arbitrary changes. The mechanism adopted is discussed in the next section, although it has a number of shortcomings that make it unsuitable for this project.

[3] `http://asic-linux.com.mx/~izto/checkinstall/`

Chapter 2

Preparation

In order to better understand the requirements of the overall system I first examined the requirements placed on the monitoring system by the `.deb` package format. I outlined a set of criteria which a monitoring mechanism would have to fulfil. Based on these criteria a method was selected and research into the behaviour of the candidate mechanism was carried out in order to implement a simple proof of concept monitoring framework. Since processes will be tracked at the system call level, those calls which are relevant to filesystem access under Unix-like operating system were examined in detail in order to betted understand how the filesystem may be modified and thus what needs to be recorded to correctly form a changeset as described above.

2.1 Implications of the `.deb` format

Each Debian package is contained within a single `.deb` file. As well as the files which are unpacked on to the system during installation or upgrade of packages, the Debian Policy Manual[3] also outlines a mechanism for the maintainer of packages to provide scripts which are executed before and after the installation, upgrade, or removal of packages. Maintainer scripts are called in many different contexts depending upon the package management operation being carried out at the time; for example the installation of a new package, upgrading a package or the removal of a package from the system. They are frequently used for user interaction in order to configure the software being installed but they may also make arbitrary changes to the system which are not or can not be represented by a file contained in the package. Some examples of this include:

- Most packages contain automatically generated scripts containing snippets of code to do certain tasks in an ordered fashion. For example most include code to manage the transition of system documentation from `/usr/doc` to `/usr/share/doc` and will create a symbolic link from `/usr/doc/packagename` to `/usr/share/doc/packagename` if `/usr/doc` is not already a symbolic link.

- In some cases, scripts must be provided to work around issues caused by design decisions to do with the `dpkg` package manager itself – a common example is that `dpkg` will never replace a directory on the filesystem with a symbolic link (nor the converse) due to potentially undefined behaviour should a directory which was expected to be empty be found to contain files, or in the converse case a symbolic link may be a deliberate attempt to store files elsewhere. Such manipulations require a maintainer script to ensure the correct behaviour for the particular case in question.

- Packages which contain shared libraries must run `ldconfig` after their installation, to update the dynamic linker's cache of available shared object files.

- Programs which manipulate data on the system can change the format that data is stored in between versions, and the maintainer scripts of programs such as the OpenLDAP and PostgreSQL daemons attempt to dump the data stored in the old format and import it into the new version. Less critical examples are minor changes in the format of configuration files, which are often also handled by maintainer scripts.

On the author's system, 650 of the 1000 installed packages provide at least one maintainer script. Of the 1550 scripts, most are POSIX or Bourne-Again shell scripts, with the exception of 25 which are Perl scripts, and one of which is a native executable — the pre-install script for the shell itself. Whilst some of these scripts contain automatically generated fragments, most are hand-written by maintainers. Due to this, and the nature of shell scripts (with regard to invoking numerous external programs), a static analysis of their actions would not be feasible or reliable given that not all are shell scripts.

Design Implications

There is no requirement for maintainer scripts to reverse actions which are carried out on package upgrades, so simply replacing the files of a package with the files provided by an older version is not sufficient to revert all of the filesystem changes. An undo mechanism which attempted to do so could at best cause that package not to function, or at worst break unrelated software or cause loss of data on the system. To correctly revert the actions of maintainer scripts from arbitrary packages it is necessary to observe all of the changes which are made by the package manager and to restore the original state.

2.2 Monitoring Strategies

Requirements Analysis

In light of the above observations, and to provide a system which is flexible enough to be used in a variety of situations, a suitable monitoring mechanism must…

- …capture enough information to fully revert any changes which are made — simply observing that a file has been deleted, modified or overwritten will not allow the file to be restored to its original state. The properties and contents of a file must be preserved before the state is changed.
- …be able to monitor any arbitrary program without modification or any prior knowledge — as already discussed, package maintainer scripts can execute any programs they wish, and the system will be far more useful if it not limited to monitoring the package manager alone — other applications such as third party install systems are good candidates for monitoring in the same way so that their changes can be reverted.
- …be able to monitor all of the changes made by any given process — it should not be possible for a program being monitored to perform actions which are not visible to the monitoring system. Although it is not a goal of the system to make it safe for untrusted code to act on the system, and there may be means that monitoring can be deliberately evaded, monitoring should be sufficiently comprehensive such that there are no circumstances where a legitimate installer or packaging system is not monitored fully.
- …not include changes which were not made by the process or task under observation — unrelated changes to the system such as the activities of other users, system daemons, log files, etc, should not be included in the changesets created, otherwise reverting these changes would cause loss of unrelated data.
- …ideally not require intrusive modifications to the system, such as kernel patches or modules — this is to aid ease of development and deployment of the system.

Several monitoring strategies were evaluated with these criteria in mind.

2.2.1 Strategy 1: Scanning the Filesystem for Recent Modifications

A simple approach to finding the files created during installation of software would be to scan the filesystem for files modified between the start and the end of the installation process. An uninstallation could then be effected by recording this list of files after the installation, and simply removing them to revert this change. Besides being time-consuming on systems with a large number of files this method fails to distinguish between changes made by the installation and changes made by other concurrent processes on the system. Furthermore, this method can only account for files being added — the original state of files which are deleted or modified is already lost by the time that file creations are observed and noted.

2.2.2 Strategy 2: System-Wide Snapshots

Several techniques exist which allow you to take a snapshot of the filesystem on a Linux system, such as the fairly well-established Linux LVM (Logical Volume Management)[1] or `unionfs` [5], which allows several filesystem to be mounted "on top" of another, with any modifications being made to a "higher" filesystem, and the old filesystem state being preserved in the original. Before an intrusive operation with the package manager a snapshot could be taken of the filesystem, and if the change proves to be undesirable the filesystem state rolled back to the original. This is a fairly simplistic approach which suffers many of the drawbacks of the previous filesystem-scanning method (§2.2.1), potentially discarding users data and changes to log files. Use of these methods does not provide a generic solution to the monitoring problem because specific kernel support is required and must be configured ahead of time.

2.2.3 Strategy 3: Wrapping the C Library

Another prevalent mechanism for recording the actions of certain processes is to use the system's dynamic linker to replace certain functions in the C library with different code. The `checkinstall` tool uses this method, which is more promising than the previously strategies because it gives the opportunity for existing state to be preserved before changes are executed — files which are to be modified are

[1] `http://sources.redhat.com/lvm2/`

copied before the C library's real functions are invoked. However, this method is not guaranteed to work with all processes. Programs which are statically linked (such as installation programs from commercial vendors), linked against older versions of the C library or execute as another user (via the `setuid` mechanism) would escape monitoring entirely because the linker will not load our wrapper library.

2.2.4 Strategy 4: Tracing System Calls in the Kernel

Recent versions of the Linux kernel include an audit framework which can be selectively enabled for certain groups of processes. It logs any system calls made along with their arguments and return values. This seems like a good basis for a monitoring system — only relevant processes are monitored, any given process can be monitored, and no filesystem changes can be made by that process which will not be recorded.

However, as well as requiring kernel audit support, there is an additional problem: there is no provision for preserving the state of an entity before a modification is carried out. A process which makes an `unlink` system call to remove a file will cause a log entry to be created, but without a copy of the file before it is removed, it is impossible to revert the action of removing it. There are several techniques which could remedy this, but they all require potentially complex kernel modifications.

2.2.5 Strategy 5: Tracing System Calls from Userspace

The Linux kernel (like most other Unix kernels) provides a debugging extension called `ptrace` to allow one process to examine and change the behaviour of another process, and this is what is used by debugging tools such as `gdb` (the GNU debugger)[2] and `strace` (system call trace tool)[3]. The `ptrace` extension has a mode of operation where the process which is being debugged is stopped upon entry and exit to the kernel when making a system call. This mode of operation is used by `strace` to examine and print the arguments of a system call before it is made, and the results of it after it has finished.

Using this mode of `ptrace` operation as the basis of a monitoring system has all of the benefits of the previous kernel-based method (§2.2.4), such as monitoring only relevant processes, being able monitor any given process without modification, and that all changes made by a given process can be observed. It

[2] `http://www.gnu.org/software/gdb/`
[3] `http://www.liacs.nl/~wichert/strace/`

also requires no kernel modifications (`ptrace` has been in the standard kernel for several years) and allows the tracing process to preserve any information which is about to be destroyed before the system call takes place.

Design Implications

As the method which best satisfies the criteria outlined above, it was decided to implement the monitoring system using the `ptrace` extension. The remainder of this chapter examines the workings of the `ptrace` mechanism, and the wide variety of system calls which can be used to modify the filesystem on a Unix system, all of which must be understood to record all modifications and preserve any filesystem state which is about to be changed. For both, the implications of the available interfaces on the design of the monitoring system are considered.

2.3 `ptrace` Mechanism

Research into the workings of the `ptrace` extension was carried out by reading the manual page for the `ptrace` system call itself, inspecting the source code of the `strace` tool, and occasionally examining other manual pages[2] and the source code of the kernel and the C library to gain a better understanding of the system call interface. To test this knowledge, a simple tool was implemented which intercepts and prints the system call number and return value of each system call made by a process after it was started.

When one process in Linux starts a child process by duplicating itself using the `fork` (or in more recent kernels, the more flexible `clone`) system call, the parent process receives a `SIGCHLD` signal when the child process has completed execution, and can then collect the exit status (or return value) of the child process by using one of the `wait` family of system calls. The status value returned can be examined using provided macros to determine the reason that the child has stopped or exited.

Use of the `ptrace` extension is also based around this parent/child `wait` system call — when one process is tracing another, it effectively becomes its parent process, and the traced process becomes the child. Tracing can be commenced on an existing process with a `PTRACE_ATTACH` call, or a child process can be traced from its outset by setting the `CLONE_PTRACE` flag on a `clone` system call. After a `fork` call, a child process can indicate that it should be traced by its parent with a `PTRACE_TRACEME` call.

When being traced, a child process stops under all of the usual circumstances such as terminating or receiving a `SIGSTOP`, but also whenever it receives any signal. The parent process uses one of the `wait` system calls to determine when a traced process has stopped, and can discover the reason for stopping by inspecting the status value returned. The parent process can continue executing the child under normal tracing, or can make a `PTRACE_SYSCALL` or `PTRACE_SINGLE_STEP` call to indicate that the child should continue but stop when entering or leaving a system call, or after each machine instruction has been executed.

When stopped, a child process can be inspected or modified by the parent process by using operations of the `ptrace` system call to access its registers or memory. In `PTRACE_SYSCALL` mode, the registers of the child process must be examined before a system call takes place to determine the system call and its arguments, and once completed, its return value[4]. If any arguments (such as filenames) are passed as pointers, the memory of the child process must also be inspected to retrieve these values. The parent process can stop monitoring or terminate the child at any time, or continue monitoring until it exits normally.

Design Implications

The monitoring system needs to observe the actions of the monitored process and its children from their outset to the end of their execution, so acts as a wrapper to a certain command which it starts by calling `fork`, and then `ptrace(PTRACE_TRACEME)` and finally `exec` in the child process. The technique adopted to handle child processes is detailed in the implementation section.

At the entry to each system call, the registers of the monitored process should be examined to determine what the system call is, and if it is one that can modify the filesystem. If so, its arguments should also be retrieved, and a copy taken of any entities which are about to be changed. Upon exit from the system call, the return value should be inspected, and the effects of successful calls recorded.

2.4 Unix Filesystem Manipulation

Interpreting the system calls used to manipulate the filesystem requires a detailed knowledge of the relevant system calls and their usage and effects. A brief study of the available means of interacting with the filesystem was carried out, again with the help of the manual pages, and observing the actions of programs using the `strace` tool.

[4]On Intel x86 processors, the system call number is stored in the `eax` register, and the first few arguments in the `ebx`, `ecx`, `edx`, `esi` and `edi` registers. The return value is also held in the `eax` register, and errors are signified by returning small negative values with pre-arranged meanings.

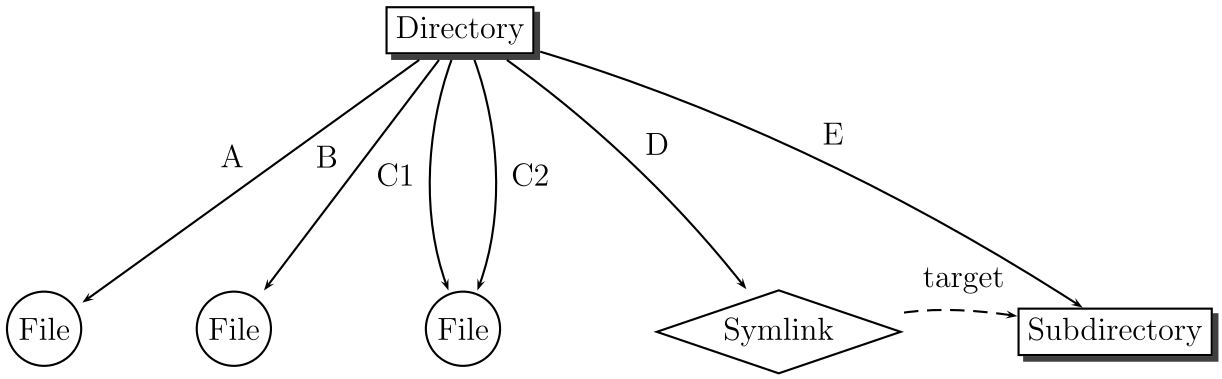

Figure 2.1: Names associated with the link from the directory to each entity

Unix filesystems represent all filesystem entities (be they directories, files, symbolic links, device nodes, named pipes or sockets) with a structure stored on the disk called an inode. Each directory contains a mapping of names to these entities, and any entity which is not itself a directory can be referred to by multiple names from within the same or other directories on the same filesystem — a technique often referred to as hard linking. The alternative strategy is a symbolic link (symlink), which is an inode on the filesystem which contains the path of another file to read instead. See Figure 2.1 — C1 and C2 are links to the same file, and D is a symlink to E. All inodes store a variety of auxiliary information about the entity, such as the permissions (except for symbolic links), the user and group that owns the entity, and three timestamps, `atime` for the time of the last access, `ctime` for the creation or last modification time of the inode, and `mtime` for the last modification time of the entity itself.

When a file is opened by a process for reading or modification, the open file is represented to the process by a file descriptor, which is a number unique to that process that can be used to refer to the file. It is important to note that file descriptors refer to entities by inode and not by pathname, so even if the entity is renamed, system calls that act on file descriptors are still applicable to the entity under its new name. Furthermore, there are a number of system calls which allow the file descriptors themselves to be manipulated — duplicated, shared with child processes, or even passed to another process.

The system calls that are relevant to tracing all filesystem modifications can be broken down into several categories:

- **System calls which directly manipulate only the mapping between directory entries and inodes.** The relevant system calls here are `unlink`, which removes a name for an entity, and causes that entity to be deleted if it is no longer referred to by any names; `link`, which creates an additional name for an existing entity; and `rename`, which atomically replaces one name for an entity with a new name, unlinking the target name if it currently exists.

- **System calls which act only upon directories.** Because they are not allowed to be referred to multiple times with hard links, and cannot be deleted if they have any contents, directories must only be created and deleted with the `mkdir` and `rmdir` system calls respectively.

- **System calls which create or manipulate inodes when given a path to a (potentially not yet existent) entity.** Some system calls such as `symlink` and `mknod` for creating symbolic links, device nodes or other special files, require the given path to not exist and do not follow symbolic links. Others such as `chown`, `chmod` or `utime` for setting inode ownership, permissions or timestamps, follow symbolic links if found. The `open` system call opens an entity for reading or writing, potentially creating a new file as a side effect if desired, and returns a file descriptor.

- **System calls which act on the entity represented by a given open file descriptor.** The `write` system call is used for writing data to a file, device, FIFO or socket at the current offset, and a number of calls such as `fchmod` or `fchown` perform the same operations as those mentioned previously, but act upon the entity represented by a file descriptor instead of a path name. Also of particular note is the `mmap` call, which can map ranges of a file into the memory space of a process, either for reading, writing or executing. When a file is mapped writably, no further system calls are necessary to change that file — writing to the pages in memory modifies the file on disk.

- **System calls which manipulate file descriptors alone, but have no actual effect on the filesystem.** Many of these are related to file descriptors which do not represent entities on disk — the `socket`, `pipe` and `accept` system calls return file descriptors representing network connections or unnamed FIFOs. The `read` system call reads data from a file descriptor; `seek` updates the location being read from or written to; and `close` indicates that a file descriptor will no longer be used, and resources associated with it can be freed (this also happens to all of the file descriptors belonging to a process when it terminates).

Each file descriptor has a flag indicating if it will be closed when the `exec` system call is made — by default after a `fork`, the child process has a copy of each of the parent's file descriptors, but after `exec` is called and a new binary is loaded, only the file descriptors flagged not to close on exec (`CLOEXEC`) remain available. The `dup` and `dup2` system calls return a replica of a given file descriptor, which the file position pointer with the original, but does not share the `CLOEXEC` flag. The `fcntl` system call allows the `CLOEXEC` flag to be set and cleared, allowing a process to select which file descriptors remain available to a child process after `exec` is called.[5]

Design Implications

It is obviously essential that all of the system calls which can affect the disk are recorded. The information recorded must be sufficient to determine the state of the entity both before and after the operation, so that the change can be reversed as well as re-applied. All operations which can operate on file descriptors and hence modify a file must also be represented, so the tracking must understand file descriptors which do not refer to files to determine which operations are relevant, and track file descriptors which are passed from a parent to a child process so that its modifications are also represented. Accordingly, almost all of the system calls mentioned here must be acted upon in some way in order to ensure full coverage of the filesystem modifications a process can make.

It is also relevant to note that for each filesystem operation outlined above, it is always possible to make a series of system calls to recreate the state of the entity before the operation was carried out. The one exception to this is an inode's `ctime`, which cannot be set to arbitrary values with the `utime` system call, unlike `atime` and `mtime`. The application or reversion of changes will have to ignore whether or not the `ctime` on the filesystem entity matches the `ctime` observed before or after changes were made, otherwise it will not be possible to revert and then re-apply any given change.

[5] Although rarely used, it is also possible to pass a file descriptor to another process on the same system by sending certain packets over a Unix Domain Socket. Given that this can pass a file descriptor to a process outside of those being monitored, coupled with the added complexity of needing to examine all socket communications, it was decided not to support this in the implementation presented here, although it does provide a means to evade monitoring of certain file modifications.

2.5 Software Engineering

The project was stored in the Darcs[6] revision control system, and developed using the iterative enhancement model[1]. The system was developed in phases, which are echoed in the structure of the next chapter, and testing of each area of functionality carried out before proceeding to the next phase. Development was directed by a list of unimplemented functionality and known defects, and areas of code redesigned when necessary for new functionality to be implemented. A defensive coding style was adopted throughout, with assertions used where appropriate to locate erroneous data within the system, and alert when executing code that should not be reached.

[6] `http://www.darcs.net/`

Chapter 3

Implementation

The implementation of the completed system for creating, applying and reverting filesystem changesets can be broken down into three stages:

1. Designing a format for representing a series of changes to the filesystem.[1]
2. Designing and implementing a monitoring system using the chosen `ptrace` mechanism to observe the filesystem changes made by a particular process, and produce a changeset in the agreed format.
3. Designing and implementing a library to create the inverse of changesets, determine if a changeset is applicable to the current system, and apply the changes if possible, and a simple interface to allow the user to carry out these operations.

This section details the decisions and implementation carried out in each phase, and includes a discussion of the unanticipated limitations encountered in the initially proposed and implemented changeset format. To address these limitations, an alternative changeset format is described, and the monitoring system revised to output this format instead.

The implementation of the monitoring system proved to be considerably more work than anticipated, due to the complexities of the `ptrace` interface and the numerous ways that processes can affect the filesystem on Unix systems. Because of the time taken to understand these factors and implement the monitoring system, and the fact it proved necessary to change the format of how changesets are represented in light of limitations encountered, the final stage of implementing tools for changeset manipulation was not completed. However, details of a possible implementation are described at the end of this chapter, and the monitoring

[1] This is loosely referred to as a "changeset" by this document, although the initially proposed format is a sequence of operations rather than a set of changes.

system is tested and shown to be functional and reasonably complete in the next chapter. Design and implementation of the final stage should be much simpler than that of the monitoring system itself.

3.1 Modelling Changesets

In order to design a format for representing changesets, and better understand the circumstances under which changes are applicable to the filesystem in a given status, a model was constructed of the Unix filesystem in Standard ML, with reference to What's in Unison?[4], which undertakes similar modelling work in OCaml in order to determine the requirements for safe file synchronisation. Appendix A includes the datatypes and functions within the model, which are defined as follows:

- `filesystem` is a datatype which represents a single inode on disk, and because it can represent a directory which recursively contain further inodes, can represent an entire filesystem tree. It contains a few example types for normal files and symbolic links, and models file `properties` and `contents` as strings. Unlike Unix filesystems, it contains an entry for `Nil`, to represent a file which does not exist.

- `path` models the `/`-separated path to a filesystem entity as a list of strings, and given a `path` and the filesystem node at the root of a tree, the `lookupPath` function returns the named entity or `Nil`.

- `condition` is a "fingerprint" of a single entity, but is not a recursive data structure. It stores the type of the entity, and its `properties` if it exists, and the `contents` of a file or `target` of a symlink. The condition of a directory does not directly depend on its contents because the presence of absence of unrelated entities within the directory does not influence its manipulation, except for deletion. Hence the directory `condition` includes only a `requireEmpty` flag to indicate whether or not it is empty.

- `change` represents an operation performed upon a single entity, so contains the `path` of an entity, its `condition` before the operation is performed, and the `operation` along with any arguments. The definition models all changes to entity properties as a `propsOp`, all changes to file contents as a `writeOp`, etc.

- A particular change is considered valid if the condition of the entity in the actual filesystem is the same as that described by the `condition` datatype, as implemented by the `validCond` function.
- Assuming the existence of an `applyOp` function which accepts a `path`, an `operation`, and a `filesystem` datatype representing the root of the filesystem, and returns the root of the filesystem with the operation carried out, the `applyChange` and `applyChanges` functions can be used to apply a single change when valid, or a whole series of changes.

It is worth making a few observations about this model of representing and applying changes, which will become more relevant when implementation is attempted. Rather than representing a changeset, this implements a list of changes, which may refer multiple times to the same entity. This means that the changes must be applied in the order they were made, and reversed in the opposite order, and also that the contents of a file may be included multiple times. In implementation, the latter problem can be addressed by representing file contents as a checksum, and storing one copy of the contents for each checksum. To address the fact that an entity might be changed multiple times during the operation of a monitored process, the application of changes can be made more efficient by determining the initial and final condition of each entity involved, and then making the minimum number of changes to the filesystem to make this the case.

Unfortunately modelling modifications to file contents as a `writeOp` is incompatible with the Unix system calls outlined above (§2.4). When a file is mapped writably into memory with `mmap`, system calls are no longer required to make changes, so this means of representing changes to file contents cannot be applied. Hence it is necessary to inspect the condition of an entity when it is opened, and again when all of its file descriptors are closed, and record an `openOp` and `closeOp` with these conditions.

3.2 Original Monitoring System

The monitoring system is based around the `ptrace` test program mentioned earlier, and accepts another command to execute and monitor as its argument. It is implemented in C due to the requirement for low-level system calls and manipulation of architecture-dependent structures shared with the kernel. The `GLib`[2] utility library was used to provide a variety of useful functions, including various

[2] `http://www.gtk.org/` — GLib is the low-level core library that forms the basis of GTK+ and GNOME.

useful functions including data structures such as hash tables and linked lists, macros to provide functionality such as assertions and convenient boolean types, and a logging framework.

3.2.1 System Call Tracing

The monitoring process allocates a `Proc` structure in memory which contains information about each process being monitored, including its current status — starting, running normally, or entering, during or leaving a system call — current working directory, and the file descriptors currently open by the process. These `Proc` structures are stored in a hash table indexed by the process ID number (`pid`) assigned by the kernel. Appendix B contains example code from the monitoring system.

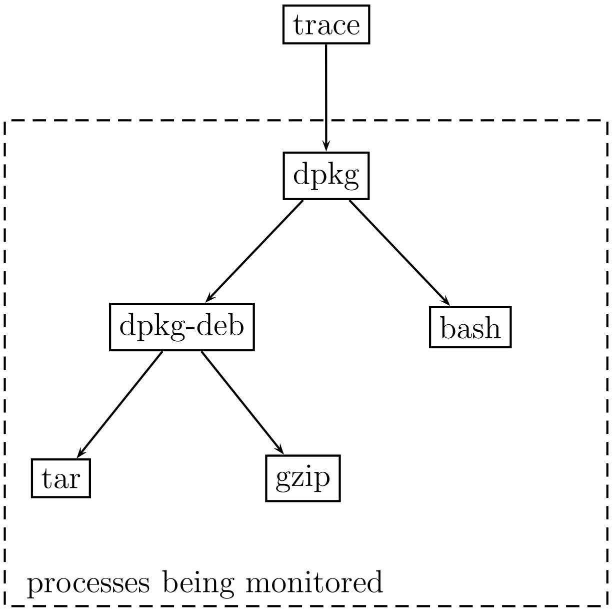

Figure 3.1: Monitoring tree of child processes

As outlined in §2.3, the monitoring system first invokes the command to monitor with `fork`, `PTRACE_TRACEME` then `exec`. The parent process creates a `Proc` structure for the child process, and then then enters a loop which, whilst there are still any active child processes being monitored, calls `wait` to be notified of any change in the status of any of its child processes (see Figure 3.1), looks up the `Proc` structure for the relevant process, determines what the status change was, and acts accordingly. If the child process has terminated, the `Proc` structure is removed from the hash table and freed, causing the monitoring system to exit from its loop if this was the last process being monitored. If the child process has stopped because it has received a signal, the action depends on which signal was received. A `SIGTRAP` signal is received when the child process enters or leaves a system call, where a call-specific handler is invoked. Most other are signals passed to the process when resuming its execution with `PTRACE_SYSCALL`.

The actions carried out when entering and leaving a system call are defined within the monitoring system in a header file which declares an array of system calls. The allocation of system calls to numbers is architecture specific, so this header file is generated by means a Perl script which reads the kernel's list of the system calls. The array is indexed by system call number, and for each system call records a name for use in debugging output, an action to take for the particular system call, and a pointer to a handler function to be invoked if this is the desired action. The supported actions for each system call are to:

- **Ignore** the given system call, and resume execution of the child process as quickly as possible — useful for system calls which have no relevance to the filesystem and are called very often by monitored processes.

- **Debug** the system call by printing its name when invoked, and return value when completed. This was valuable during implementation to examine the actions of particular monitored processes.

- **Handle** the system call by invoking the given function upon entry and exit to the system call. The handler functions accept a pointer to the `Proc` structure belonging to the process, from which the handler function can determine if a system call is being entered or exited. The structure is also updated with the system call number being entered, as well as the return value or error number when leaving, and provides an area for the handler function to store data between these two invocations.

- **Panic** to stop the monitoring system in the event of the child process performing a system call which the monitoring system is unable to handle for whatever reason.

A number of utility functions were implemented to take care of invoking the different operation modes of the `ptrace` system call. The simplest of these are just to invoke the `PTRACE_SYSCALL` and `PTRACE_TRACEME` operations and halt the monitoring system in the case of encountering any errors, but most of the others perform another valid purpose — they restrict the architecture-dependent code to as few places within the program as possible[3]. There is a single function within the wrapper code which accepts a system call argument number and returns the offset into the kernel structure which represents the register containing that argument. The functions which read and write system call numbers, arguments and return values all use this function, so porting the monitoring system to architectures other than x86 should be reasonably easy.

Testing

After this part of the monitoring system had been implemented, it was possible to trace arbitrary binaries and compare the debugging output with `strace`, and be sure by means of comparison that all of the system calls were being intercepted. Stub handlers were implemented for some system calls to print some of the arguments, demonstrating that the retrieval of arguments was also functioning.

3.2.2 Working Directory Tracking

Any system call which accepts a path to a filesystem entity as an argument can accept either an absolute path, starting with `/` (the root of the filesystem), or a relative path which is resolved from the current working directory. In order to record filesystem changes against the absolute path of the entity being manipulated, and not a symbolic link to it which might not exist when the change is being reverted or applied later, the monitoring system must always canonicalize paths by replacing any elements which are symbolic links with their absolute path, and resolving relative paths from the working directory of the monitored process.

The GNU C library provides a `canonicalize_file_name` function which resolves a path relative to the working directory of the current process, following any symbolic links encountered, and removes any `..` or `.` shorthands within the path provided. However, this is not a complete solution for the monitoring system, which must be able to resolve a path relative to the working directory of another process, resolve a path to a symbolic link itself (without following it), and resolve the absolute path of an entity which does not yet exist (for which the C library function fails).

The monitoring system has a wrapper to this function which is used internally to resolve paths passed as arguments to system calls made by monitored processes,

[3]Even given this layer of abstraction, and generating a system call table for a particular architecture, it is not quite possible to make the rest of the program totally architecture independent because certain system calls are implemented differently on different architectures, so need to be handled separately within the monitoring system.

relative to their working directory and not that of the monitoring system itself. It is hence necessary for the monitoring system to be aware of changes to the current working directory of a process — child processes inherit their working directory from their parent, so the first process which is started by the monitoring system has its working directory initialised to that of the monitoring system itself. The `chdir` and `fchdir` system calls are used by the monitoring system to track changes made by child processes to their working directory and update the `Proc` structure. The former accepts a path (which itself might be a relative path that requires canonicalization), and the latter accepts a file descriptor representing a directory, from which the path can simply be copied.

Testing

Correct tracking of the current working directory, and correct path canonicalization, was confirmed by running a shell within the monitoring system, and changing the working directory to various symbolic links and contrived relative paths. Extensive testing was not necessary because the correct functioning of the rest of the system is highly dependent on path canonicalization, and any errors would quickly have become apparent in further testing.

3.2.3 Process Tracking

The `clone` system call is specific to Linux, and is the most flexible for creating child processes because it can accept various flags. Of particular use to the monitoring system is the `CLONE_PTRACE` flag which instructs the kernel that the new child process should be traced by the same process as is tracing the parent. Upon entry to the `clone` system call, the monitoring system modifies the arguments to set this flag, and upon exit from the system call, inspects the `pid` value returned to the parent, and sets up a new `Proc` structure for the child process. The working directory is inherited from the parent process, and the child also given a copy of the open file descriptors of the parent (see §3.2.4). This has a slight side-effect on the monitored processes — the child process is prohibited from executing until the `clone` system call returns in the parent, and the child `Proc` structure is set up. However, no guarantees are made about which will be scheduled first by the kernel, so this is not a problem in practice, and ensures that the child process is monitored from the outset.

The `fork` system call is common to all Unix systems, and is implemented within the Linux kernel in the same way as `clone`, but has no provision for accepting flags such as `CLONE_PTRACE`. However, because it is implemented within the kernel using the same function, the return values of both calls are interchangeable. The monitoring system can modify the registers of the process before it enters a system call, so it is reasonably easy to preserve a copy of the relevant registers in the data member of the `Proc` structure, and rewrite the `fork` system call into a `clone` system call with the `CLONE_PTRACE` flag. When the system call returns, the monitoring system replaces the backup copy of the modified registers in both the parent process and the child process (if the call is successful).

The `vfork` call is a special variant of the `fork` system call which used to be necessary in old kernel versions for performance reasons. Instead of copying all of the data and other state into a new child process in the kernel, only to have all of this discarded and replaced when the child process makes an `exec` call moments later, when `vfork` is used, the parent is blocked and the child process runs in the parent's memory space until the child calls `exec` or exits. This is difficult to track because the child process ID is by definition only returned to the parent after the child has started executing, so is handled by the monitoring system in the same way as `fork`, rewriting to a `clone` system call with the appropriate arguments.

Testing

Correct tracking of `clone`, `fork` and `vfork` system calls was verified by a pair of test programs which called `fork` or `vfork` in a loop to create a child process every few seconds, which then made a few system calls and then exited. New versions of the C library implement `fork` using the `clone` system call as a consequence of new threading support, but can be made to use `fork` by making the linker assume an older kernel version is running. In this way, correct tracking of all three system calls was verified.

3.2.4 File Descriptor Tracking

In order to support tracking of the file descriptors open by a monitored process, three new data structures were added to the monitoring system. The first, called `ProcFd`, represents a file descriptor belonging to a process, and is stored in a hash table within the `Proc` structure, indexed by file descriptor number. It can be one of three types: a file, an unnamed pipe, or a network socket, and has a boolean to track whether its `CLOEXEC` flag is set (see §2.4). The second, called `File`, represents a currently open file. They are stored in a global hash indexed by the canonical pathname of the file, and includes the `Condition` of the file when it was first opened, a reference count of open file descriptors which refer to it, and booleans indicating whether or not a copy of the file has been preserved, and whether the file has been modified on disk (dirtied) since it was preserved. The `Condition` structure includes the type of the entity, be it absent, file, directory, symbolic link, device node etc, the properties such as timestamps, ownership and permissions from the inode, and various type-specific information such as a SHA1 checksum of the contents for files (rather than actually storing the entire contents in memory), the target of symbolic links, and the device numbers for device nodes.

File descriptors representing sockets or pipes are created whenever a successful `socket`, `accept`, `socketpair` or `pipe` invocation is made. The first child process started by the monitoring system is given three pipe file descriptors as 0, 1 and 2, representing the standard input, output and error file descriptors inherited from outside the monitoring system itself. These socket and pipe file descriptors have their `CLOEXEC` flag monitored so they are correctly duplicated and closed when monitored processes invoke new binaries, but no action is carried out when modifications are made to them.

File descriptors representing files are created by successful `open` or `creat` (a special case of `open`) system calls. If one does not already exist, a `File` structure is created for the canonical path of the file being opened, and a `ProcFd` structure created to record the file descriptor assigned by the kernel. If the `open` system call can modify the disk itself, such as the `O_CREAT` flag being set so that a file is created if it does not exist, or `O_TRUNC` so that the file is truncated to 0 bytes when it is opened, the monitoring system preserves a copy of the file before the call is allowed to proceed.

When made on a file descriptor representing a file, system calls such as `write`, `mmap` (if a writable mapping is created) and `ftruncate` flag the file as dirty. The first time a file is flagged as dirty, a copy of the contents of the file is taken, and the file also flagged as preserved. These copies are stored in a directory created each time the monitoring system is invoked, and are named after the SHA1 checksum of the file. As a consequence, if a file is acted upon multiple times, or multiple files have the same contents, only one copy of the file is retained. When a file descriptor is closed by a process with the `close` system call, or the process exits and all of its file descriptors are closed, if the file descriptor refers to a `File`, the reference count of that `File` structure is decremented before the `ProcFd` is discarded. If the `File` reference count becomes 0 and that file is flagged as dirty, a `close` operation is logged with the condition of the file after it was closed, and an updated copy of the file taken.

The value of the `CLOEXEC` flag is tracked per file descriptor by handling the `dup`, `dup2` and `fcntl` system calls, updating a flag in each `ProcFd` structure. When file descriptors are duplicated, a duplicate `ProcFd` is created with the `CLOEXEC` flag set to false, and the reference count of the `File` is incremented if it is a file descriptor which refers to a file. When a process successfully invokes `clone`, `fork` or `vfork`, the `ProcFd` structures are copied from the parent to the child process without changing the `CLOEXEC` flags, and the reference count incremented for those which point to `File` structures. Finally, when `exec` completes, any file descriptors that have the `CLOEXEC` flag set are closed, with the same reference counting updates and change logging as described above.

Testing

File descriptor tracking was tested by the use of assertions — in the event of a process successfully manipulating a file descriptor which the monitoring system was unaware of, or a process being given a new file descriptor which the monitoring system considered to already exist, an assertion failure halted the monitoring system to indicate that the tracking of file descriptors had made an error. After tracking down these errors with the aid of debugging output and `strace`, to see where file descriptors are opened and closed, it became possible to run processes such as `xterm`, which starts a shell from which arbitrary processes can be run. This was a good acid test of the tracking because the shell makes extensive use of file descriptor manipulation to set up the standard file descriptors for child processes, and the X terminal itself uses file descriptors to represent its socket connection to the X server and the pseudo-terminal used for exchanging data with the shell process. The monitoring system still contains these assertions, and no further errors were encountered in subsequent development and testing.

3.2.5 Problems with Original Implementation

The original monitoring system maintained a linked list of `Change` structures in memory, each of which echoes the `change` datatype in the model above, containing the path to an entity, its condition when the operation is carried out, the operation, and its arguments. The remaining system calls such as `chown`, `chmod`, `utime`, `truncate`, `lchown`, `unlink`, `mkdir` etc, all of which accept a path to act on as their first argument, were logged directly to this change log. A textual representation of this log was printed to a file upon exit from the monitoring system. For example, the following represents changing the permissions of a file `/tmp/example`:

```
Path: /tmp/example
Type: file
Mode: 0644
User: 1000
Group: 1000
```

```
Mtime: 1116330447

Operation: chmod
Mode: 0755
```

The arguments to the system calls were retrieved by inspecting the registers of the process upon entry to each system call. However, when approaching completion of the monitoring system, it became apparent that there were several limitations to this method of recording changes:

- In order to implement all of the relevant system calls, a large number of very similar handler functions were required, some differing only in very minor ways, such as whether or not to dereference the provided path if it refers to a symbolic link, or which arguments to retrieve from registers and store in the change. As well as being inefficient, this code repetition provides a large scope for bugs such as inconsistent behaviour between the handling of similar system calls.

- The actual result of filesystem operations in the kernel can depend on a large number of factors which would also have to be recorded in the change to allow one to be able to determine the condition of an entity after the operation was carried out. For example, the timestamps when an operation is recorded within the kernel, or the user ID or permissions new entity is created with. In some cases it essentially becomes necessary to examine the condition of a file after a change is made to see what this information was recorded as.

- There is an undesirable dichotomy between the handling of system calls which accept paths and make immediate changes to the filesystem, and those which operate on file descriptors representing open files. When a file that's flagged as dirty is closed, the condition logged at this time will incorporate all of the changes made whilst it was open, rendering superfluous all of the changes logged since it was opened.

- A particular failing of the assumption that a single entity is acted upon by a change is the `rename` system call, which has the capacity to overwrite its destination path if it already exists. To correctly revert this change, a copy of both the source and the destination file is required, and logging changes against a single entity does not allow this to be correctly represented.

In order to address the limitations in the initial implementation of the monitoring system, and the selected format for representing changes, a new format for changes was defined, and the monitoring system revised to use it.

3.3 Revised Monitoring System

Acknowledging that there is no benefit to recording the individual operations performed to entities on the filesystem, and that most operations have side-effects which require the entity to be inspected after each operation, a new model is proposed which discards the concept of a change to a particular entity with a particular operation and arguments.

Returning to the model in Standard ML, the new format could be described as:

```
datatype entity = Entity of path * condition * condition;
```

where the first condition is the state of the entity before the monitored processes made any changes to it, and the second condition is the state of it after the last change has been made. Representing the changes made as a set of these `entity` datatypes has a number of advantages. System calls which influence two entities can be handled simply by recording the condition of both entities before and after the call is made. It is no longer necessary to explicitly record changes to the `umask` of a process, or investigate the timestamp or ownership of a file after operations are carried out, because all of the effects of the environment upon the operations carried out is represented in the condition of each entity.

The implementation of this `entity` tracking was carried out by generalising the existing `File` structure, turning it into a new `Entity` structure, which has the initial and final conditions, a dirty flag, and an reference count of how many open file descriptors refer to this entity. This allowed many system calls to be implemented with the same function, due to the kernel's convention of indicating successful system calls by returning zero. The initial monitoring system numbered 4500 lines of code, which was reduced to 2500 in the revised system, which offered significantly improved functionality and coverage of system calls.

One function was provided for implementing the handling of system calls which operate on an open file descriptor. It ensures that the condition of the entity has been preserved before the system call is allowed to proceed, and then flags the entity as dirty if the system call was successful. This was used to implement the `write`, `ftruncate64`, `fchown32` and `fchmod` system calls, and allowed support to be added for less common or deprecated variants such as `writev`, `ftruncate` and `pwrite64` with only slight changes.

Another function was provided for handling system calls which operate on a given path, taking the argument number of the pathname, and whether the the system call dereferences symbolic links or acts upon them. Before allowing the system call to proceed, it creates a new `Entity` if one did not exist and ensures its condition is preserved. If the system call was successful, the entity is flagged as dirty. Use of this function for handling system calls like `chown32`, `truncate64`, `unlink`, `rmdir` and `mknod` allowed the deletion of hundreds of lines of code, and all of the less common and deprecated variants such as `truncate`, `chown` and `lchown` could be added with very few modifications.

After all of the processes being monitored have terminated, the initial and final condition of every entity which has changed is saved to a log file:

```
Initial: /tmp/test/70
Type: absent

Final: /tmp/test/70
Type: normal
User: 1000
Group: 1000
Mode: 0644
Mtime: 1116410883
Hash: da39a3ee5e6b4b0d3255bfef95601890afd80709
```

Support of the `rename` and `link` system calls requires a little more thought because entities being operated on may be referred to by open file descriptors. If an entity being renamed is currently open, it is necessary that after the `rename` system call, the file descriptors point at the `Entity` associated with the new name — this is achieved by swapping the structures round, so that file descriptors pointing to the original file still point to the point to it under its new name. Whilst the former change format allowed a `link` call to be logged as a seperate action, it is harder to allow `Entity` structures to represent the same file on disk without introducing a layer of indirection involving device numbers and inode numbers (with associated complexities — inode numbers can be re-used). The most common operation that it is necessary to support is that carried out by `dpkg` — "backing up" a file with `link` before atomically replacing it with `rename`, and then unlinking the backup copy if this is successful. Support for this can be included by simply having updates to an entity structure update all of the others to which it has been linked. This will not record the link action in the log, but will ensure that the initial and final states of any entities linked together during the monitoring period are correctly reported.

3.4 Changeset Manipulation

The application of changes was originally going to entail reading the log of each change in turn, and creating a set of entities in memory representing the initial condition of each entity before any changes were performed (a pre-condition), and then its updated condition after each change in the log was considered in turn (a post-condition). The applicability of a changeset could then be determined by comparing the pre-condition of an entity to the current state of the filesystem, and the application of a changeset effected by making whatever changes are necessary to update the entity to its post-condition.

In light of the revised monitoring system, which outputs a pre- and post-condition for each entity manipulated in a given changeset, much of the implementation of the simulation, application and reversal of changes is made considerably easier — the application of changes no longer requires simulating the pre- and post-conditions of each entity in memory; these can simply be read from the log produced by the monitoring system. The reversal of a changeset can be accomplished by simply swapping the pre- and post-conditions for each entity involved.

The only remaining matter for consideration is the order in which changes are applied. Because the output of the monitoring system is a set, no order is guaranteed, but there are some constraints imposed on the order of change application by the Unix filesystem. The main requirement is that directories cannot be deleted if they still contain any entities, and entities cannot be created if their parent directory does not yet exist. A further desirable property is that changes to the system are as near to atomic as possible, so that other processes do not see some files which have had their state changed, when other related files have not yet been updated.

After it has been confirmed that all of the pre-conditions match the current state of the filesystem, changes could be applied in the following order to give the best chance of success, similar to the approach adopted by `dpkg`:

1. Create any new directories with `mkdir` and set their properties with `chmod`, `chown` and `utime`.

2. For any entities which have been newly created, or changed from one entity to another, produce a replacement entity with a temporary name and the appropriate properties alongside the existing one.

3. Rename all of the temporary entities which are not symbolic links to their final names, overwriting the originals if they exist and are not directories (which cannot be overwritten with a `rename` call).

4. Rename all of the temporary symbolic links to their final names — this is done seperately to ensure that symbolic links are not left pointing to entities which do not yet exist.

5. Unlink any entities which are to be removed and are not directories.

6. Now they will be empty, `rmdir` any directories which are be removed or replaced with any other entity.

7. Finally, rename any temporary entities which are to replace directories.

Chapter 4

Evaluation

The monitoring system allows arbitrary processes to be monitored, and records (with a few exceptions, see §4.3) all of the changes made to the filesystem. Excerpts of the log produced by monitoring `dpkg` when installing an example package (the `unison` file synchroniser) are included in Appendix C. The entities included in the log can be broken down into several categories, as shown in Table 4.1. The log not only includes all of the files unpacked from the `.deb` itself (the files created in `/usr`), but also the updates that `dpkg` makes to its database in `/var/lib/dpkg`. Of particular note are the changes made in `/etc/alternatives`, which are made by the `update-alternatives` program which is called by the post-install script of the package, demonstrating that the effects of these scripts have also been recorded, as was deemed essential in the planning of this project.

The system is usable for monitoring most processes, although occasional bugs and assertion failures can still be encountered when monitoring certain programs. The performance is a little under par as shown in the following section, but approaches for remedying the performance are suggested below.

4.1 Completeness

Although examining the output is sufficient for one to be convinced that the changes made to some entities are correctly recorded, it is also necessary to determine whether the monitoring system has recorded all operations performed and preserved the state of every single file which has been changed. To confirm this, the contents of several large `.deb` packages (such as TEX, icon themes for GNOME, and the OpenOffice.Org desktop suite), were unpacked onto the system with the monitoring system observing `dpkg`, and the monitoring system used again to observe the files being removed. Each package contains several thousand

Reason for Change	**Affected Paths**
Files contained within the package	`/usr/bin/unison2.10.2`, `/usr/share/doc/unison/changelog.Debian.gz`, `/usr/share/doc/unison/README.Debian`, ...
Directories contained within the package	`/usr/share/doc/unison`, ...
Symbolic links created by the maintainer script	`/etc/alternatives/unison`, `/usr/bin/unison`, ...
Updates to `dpkg`'s database	`/var/lib/dpkg/info/unison.list`, `/var/lib/dpkg/status`, `/var/lib/dpkg/info/unison.postinst`, ...

Table 4.1: Changes observed when installing `unison` with `dpkg`

files, and a significant number of directories and symbolic links, and the number of entities created by `dpkg` can be determined by using verbose extraction. This number of expected and observed entity changes was compared, and was found to match in all cases.

4.2 Performance

In order to consider the overhead of the monitoring system upon various relevant workloads, the following test workloads were selected from various parts of the Debian package management system:

- **CPU-intensive** — To represent a process which makes heavy use of the processor, but few system calls, the `apt-get` tool is asked to build a dependency tree of installed packages, and check it for consistency.

- **read-intensive** — To represent a process which makes frequent system calls to read the filesystem, but makes few modifications, `dpkg` is used to search its database of installed packages for files matching a certain pattern, causing 1000 files to be read on the author's system.

- **write-intensive** — To represent the installation of a new package, `dpkg` is used to unpack the files contained within a large package (OpenOffice.Org), which contains some 2000 files totalling approximately 30MB.

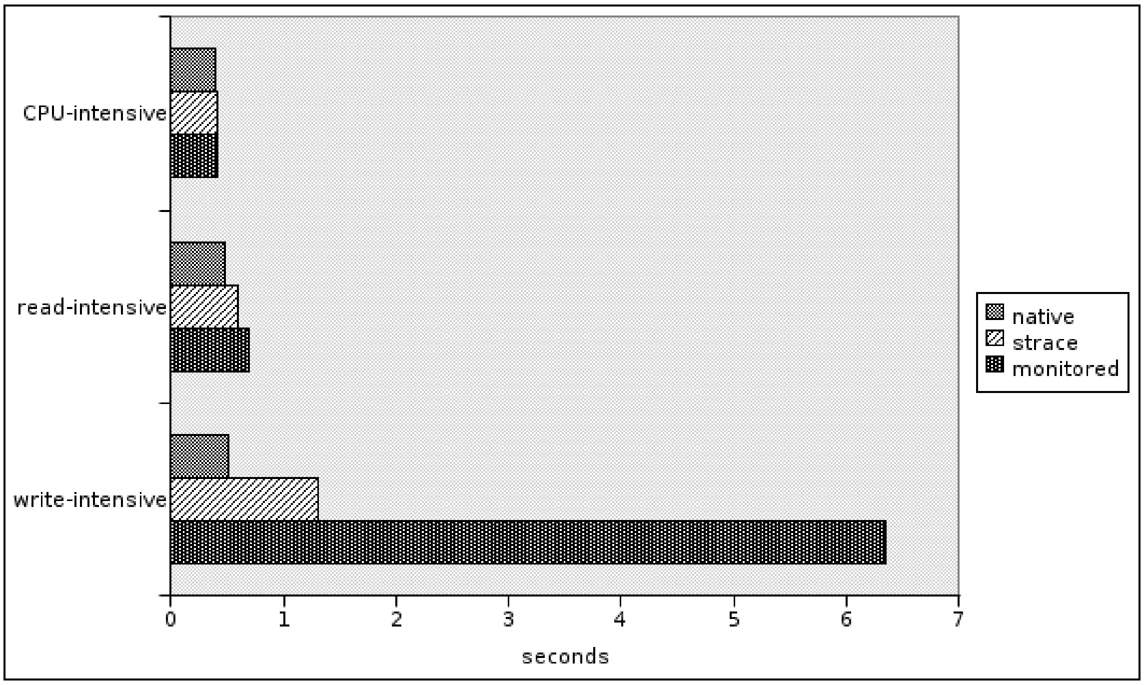

Figure 4.1: Graph of comparative benchmark results

Workload	**Native**	`strace`	**Monitored**
CPU-intensive	0.395s	0.413s	0.411s
read-intensive	0.481s	0.587s	0.685s
write-intensive	0.513s	1.308s	6.345s

Table 4.2: Results of comparative performance benchmarks

The read and write tests are performed within directories on a RAM-based `/tmp` filesystem to exclude variation based on disk access times. Each workload was benchmarked 10 times in three different circumstances to determine the average execution time: running natively under no monitoring system, under `strace` — to consider the overhead of intercepting system calls with `ptrace`, and under the monitoring system itself — to include the overhead of preserving filesystem modifications. The standard output and standard error of all commands was directed to `/dev/null` to eliminate any delays writing to the console.

The average times obtained are presented in Table 4.2 and Figure 4.1. The CPU-intensive workload makes essentially no system calls, so the overhead of monitoring it with `strace` or the monitoring system is negligible. The read-intensive workload demonstrates that there is a slowdown associated with tracing the system calls of a process, which applies equally to `strace` and the monitoring system. Every time a system call is entered or exited, the monitored process must be descheduled, the monitoring system must be executed, and must make several system calls of its own to determine if the system call is relevant, and to retrieve its arguments. The write-intensive operation shows a similar difference between the times for unmonitored execution and monitoring with `strace`, but the monitoring system proves to be significantly slower — taking approximately 12 times longer than the unmonitored process.

The reason for this significant loss in performance was investigated using a profiling tool. The monitoring system spent 96% of this time within functions responsible for computing SHA1 checksums. These are computed for the contents of a file whenever the condition of an entity is inspected, which happens before the first change to a file is allowed to proceed, after every subsequent change, and whilst a file's contents are being backed up (to guarantee it has not been externally modified). The checksumming is necessary to determine if files are in the same state as they were when a change was made, because as detailed in §2.4, timestamps are not sufficient to guarantee this, and also to prevent multiple copies of files being taken. However, performance could be improved by using a less CPU-intensive checksumming algorithm such as MD5, at risk of slightly increasing the chance of hash collisions. A distinction could also be made within the monitoring system between operations which change only the inode, and those which can change file contents, and taking checksums only in the latter case.

4.3 Limitations

There are certain circumstances where the output of the monitoring system does not represent all of the possible modifications made to the filesystem. Some of these are due to system calls which are not understood by the implementation of the monitoring system presented here. System calls for removing and setting POSIX extended attributes on files, which are also used for implementing POSIX access control lists on Linux, are not handled, and the monitoring system is set to panic when they are encountered. To support these, the definition of an entity's condition could be updated to contain any extended attributes set on a file, and the system calls which modify them handled in a similar way to existing functions which modify file properties. Also on a related note, attributes which are specific to certain filesystem types are not recorded — the Linux `ext2` filesystem has attributes such as immutable files which can be modified using the `ioctl` system call, which is currently ignored.

The other notable unhandled system call is `chroot`, which changes the root directory of a process to a specified directory to restrict access to parts of the filesystem. Support for this can be added in a similar way to the existing tracking of the working directories, with a few caveats. Working directories can be outside of the root directory of a process, so a simple implementation could just add the current root directory of a process to the start of absolute paths when canonicalizing. Implementation of this was not considered necessary because this system call is used rarely during software installation, because processes which run as the superuser can trivially escape from their root directory, so it does not enhance security.

The monitoring system has some architectural limitations which currently make it difficult to support threaded processes. When creating threads, passing certain flags to the `clone` system call allows a newly created process to *share* its file descriptors (rather than a one-off copy) and current working directory with its parent process. Implementing reference counted structures to hold this information would allow them to be shared between processes in the monitoring system. Similarly the concept of filesystem entities with more than one name, ie hard links, is poorly implemented. Currently the existence of links before monitoring starts is not considered — only linking and unlinking done by the monitored process is represented by grouping entities together. Adding a concept of inode and device numbers is the strictly correct way of doing this, so that any entities which are the same node on disk are represented only once in the monitoring system. This would require a change in the format of the output, and also requires care to implement because once an entity is deleted, its inode number can be reused by the filesystem for an entirely different entity.

Any access to device nodes is not monitored — any process which risks filesystem corruption by subverting the kernel's filesystem layer entirely (particularly because it can contain buffers which have not yet been written to disk) and writing directly to the disk will be able to avoid monitoring. If the monitoring system was intended to safeguard against potentially malicious processes such as this, it could discard the capability for raw device access before starting the child process.

A process can pass its file descriptors to another process on the same system using a Unix domain socket — this is not caught by the monitoring system but is exceedingly rare, and would rapidly result in assertion errors due to missing or mysteriously appeared file descriptors if it did occur. It was not implemented because it would require a fairly costly inspection of every packet being sent or received on file descriptors representing these type of sockets, so represents a trade-off between a totally comprehensive monitoring system, and one which is sufficiently lightweight to use with everyday tasks.

Chapter 5

Conclusions

Despite not having the opportunity to implement the latter part of the completed system for applying and reverting changesets, the majority of the work has been completed in the implementation of the monitoring system. The monitoring framework presented is almost complete, confirms the validity of the approach suggested, and forms a solid foundation for a completed and enhanced system. Ideally it would have been possible to complete the stated goals and implement the ability to apply changes, but the monitoring system still represents a significant achievement as a stand-alone piece of work.

5.1 Future Work

Most of the points outlined in §4.3 can be addressed to solve the known shortcomings with the monitoring system as implemented, both in terms of performance and completeness of monitoring. Some changes are reasonably straightforward, but some such as the sharing of structures to represent threading or hard linking might be made easier by implementing parts of the monitoring system in a higher-level language with reference counted objects, and wrapping the necessary low-level `ptrace` functionality for use in this language. Code or at least relevant data structures could then be shared between the implementation of the monitoring system and the subsequent code for manipulation of changes.

Another area for consideration is support for stopping monitoring of processes which appear to be daemons. Package maintainer scripts for daemons often restart the daemon when it is upgraded, and under the current monitoring system, the daemon will be monitored indefinitely and monitoring will never terminate. Handling certain system calls involved in changing process groups will allow daemonisation to be detected, and the necessary processes released from monitoring. Obviously this is at odds with the idea of the monitoring system as a sandbox, but is necessary to support general usage of `dpkg`, so could be made optional and decided on a per-invocation basis.

As detailed in the project proposal, potential extensions to the project beyond implementing changeset application could include enhancing the applicability of changes by considering the contents of certain files as separate entities. Each package installed on the system has an entry within the `/var/lib/dpkg/status` file — considering each as a separate entity would allow the reversion of a changeset involving that file even when subsequent changes have been made to other parts of it. This would render the system significantly more robust against filesystem changes, providing a more flexible utility.

Bibliography

[1] Victor R. Basili and Albert J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Trans. Software Eng.*, 1(4):390–396, 1975.

[2] Michael Kerrisk et al. Linux and posix manual pages. `ftp://ftp.kernel.org/pub/linux/docs/manpages/`.

[3] Ian Jackson and Christian Schwarz et al. Debian Policy Manual. `http://www.debian.org/doc/debian-policy/`.

[4] Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[5] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Computer Science Department, Stony Brook University, October 2004. `http://www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf`.

Appendix A

Filesystem Changes in ML

This is the model of filesystem changes in Standard ML, presented in §3.1.

```
type name = string;
type contents = string;
type target = string;
type properties = string;

datatype filesystem = File of properties * contents
                    | Dir of properties * (name * filesystem) list
                    | Symlink of properties * target
                    | Nil;

exception InvalidPath;

(* val lookupName = fn : name * (name * filesystem) list -> filesystem *)
fun lookupName(n, (c, f)::dir) = if n = c then f
                                            else lookupName(n, dir)
  | lookupName(n, [])          = Nil;

(* val lookupPath = fn : path * filesystem -> filesystem *)
fun lookupPath([],    f) = f
  | lookupPath(p::ps, f) = case f of Dir(_, c) => lookupPath(ps, lookupName(p,c))
                                   | _         => raise InvalidPath;

type requireEmpty = bool;

datatype condition = fileCond of properties * contents
                   | dirCond of properties * requireEmpty
                   | linkCond of properties * target
                   | absentCond;

datatype operation = deleteOp
                   | writeOp of contents
                   | propsOp of properties
                   | createFileOp of properties * contents
                   | createLinkOp of properties * contents
                   | createDirOp of properties;

datatype change = Change of path * condition * operation;

(* val validCond = fn : filesystem * condition -> bool *)
fun validCond(File(p, c),    fileCond(sp, sc)) = p = sp andalso c = sc
  | validCond(Dir(p, c),     dirCond(sp, re))  = p = sp andalso (case c of [] => true
                                                                          | _  => not(re))
  | validCond(Symlink(p, c), linkCond(sp, sc)) = p = sp andalso c = sc
  | validCond(Nil,           absentCond)       = true
  | validCond(_,             _)                = false;

exception InvalidCond;
```

```
(* val applyOp = fn : path * operation * filesystem -> filesystem *)

(* val applyChange = fn : change * filesystem -> filesystem *)
fun applyChange(Change(path, cond, oper), root) =
      if validCond(lookupPath(path, root), cond) then applyOp(path, oper, root)
                                                 else raise InvalidCond;

(* val applyChanges = fn : change list * filesystem -> filesystem *)
fun applyChanges(c::cs, root) = applyChanges(cs, applyChange(c, root))
  | applyChanges([],    root) = root;
```

Appendix B

Code Sample

This is the `main.c` of the monitoring system, including the `main` function and the `trace` routine it calls in the main loop.

```
#include <errno.h>
#include <glib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>

#include "entity.h"
#include "proc.h"
#include "ptrace.h"
#include "syscall.h"

static GSList *pending;

typedef struct {
        pid_t pid;
        int status;
} PendingProc;

void trace() {
        Proc *proc = NULL;
        pid_t pid;
        int status;

        /* if we have any pending procs, see if they've got Proc structures yet */
        if (pending != NULL) {
                GSList *tmp;
                for (tmp = pending; tmp != NULL; tmp = tmp->next) {
                        PendingProc *pp = (PendingProc *) tmp->data;
                        proc = proc_get(pp->pid);
                        if (proc != NULL) {
                                pending = g_slist_remove(pending, pp);
                                pid = pp->pid;
                                status = pp->status;
                                g_free(pp);
                                g_debug("%u: handling pending process", pid);
                                break;
                        }
                }
        }

        /* if we havn't recovered a pending proc, get a new event */
        if (proc == NULL) {
                if ((pid = wait(&status)) < 0)
                        g_error("trace(): wait failed: %s", g_strerror(errno));
```

```
                proc = proc_get(pid);
                if (proc == NULL) {
                        PendingProc *pp = g_new(PendingProc, 1);
                        pp->pid = pid;
                        pp->status = status;
                        pending = g_slist_prepend(pending, pp);
                        g_debug("%u: adding pending process", pid);
                        return;
                }
        }

        if (WIFEXITED(status)) {
                g_debug("%u: child exited with status %d", pid, WEXITSTATUS(status));
                proc_exit(proc);
        }

        else if (WIFSTOPPED(status)) {
                int signal = WSTOPSIG(status);

                switch (signal) {
                case SIGSTOP:
                        /* new child processes are SIGSTOPped, so we kick them off */
                        if (proc->state == PROC_STATE_STARTING) {
                                ptrace_cont_syscall(proc, 0);
                                proc->state = PROC_STATE_RUNNING;
                                g_debug("%u: child process started from SIGSTOP", pid);
                        } else {
                                g_debug("%u: not delivering SIGSTOP, already stopped!", pid);
                        }
                        break;
                case SIGTRAP:
                        switch (proc->state) {
                                case PROC_STATE_STARTING:
                                        /* new child processes get trapped immediately */
                                        proc->state = PROC_STATE_RUNNING;
                                        g_debug("%u: child process started from SIGTRAP", pid);
                                        break;
                                case PROC_STATE_RUNNING:
                                        proc->state = PROC_STATE_ENTERING;
                                        syscall_enter(proc);
                                        break;
                                case PROC_STATE_DURING:
                                        proc->state = PROC_STATE_LEAVING;
                                        syscall_leave(proc);
                                        break;
                                default:
                                        g_error("%u: received unexpected SIGTRAP!", pid);
                        }

                        ptrace_cont_syscall(proc, 0);

                        switch (proc->state) {
                                case PROC_STATE_ENTERING:
                                        proc->state = PROC_STATE_DURING;
                                        break;
                                case PROC_STATE_LEAVING:
                                        proc->state = PROC_STATE_RUNNING;
                                        break;
                                case PROC_STATE_DURING:
                                case PROC_STATE_RUNNING:
                                case PROC_STATE_STARTING:
                                        break;
                                default:
                                        g_error("%u: reached unexpected state after invoking handlers", pid);
                        }

                        break;
                default:
                        ptrace_cont_syscall(proc, signal);
                        g_debug("%u: delivered signal %u", pid, signal);
                        break;
```

```
                }
        }

        else {
                g_error("%u: process stopped for unknown reason!", pid);
        }
}

int main(int argc, char* argv[]) {
        pid_t pid;
        char *cmd;
        char *cwd;

        g_set_prgname("trace");

        if (argc < 1)
                g_error("Usage: %s [program name] [arguments]", argv[0]);

        cmd = g_find_program_in_path(argv[1]);
        if (cmd == NULL)
                g_error("main(): unable to find command %s", argv[1]);

        proc_init();
        entity_init();

        switch (pid = fork()) {
        case -1: /* error */
                g_error("main(): fork failed: %s", g_strerror(errno));
                break;
        case 0: /* child */
                ptrace_traceme();
                execv(cmd, argv+1);
                g_error("main(): execv failed: %s", g_strerror(errno));
                _exit(1);
                break;
        default: /* parent */
                cwd = g_get_current_dir();
                proc_new(pid, cmd, cwd);
                proc_fd_std(proc_get(pid));
                g_free(cmd);
                g_free(cwd);

                break;
        }

        while (proc_count() > 0) trace();

        entity_log();

        return 0;
}
```

Appendix C

Example Output

This is an example log file from monitoring `dpkg` installing an example package. For every hash recorded in an initial or final condition, a copy of the file is included alongside the log in the directory produced by the monitoring system. For brevity, some changes observed have been removed.

```
Initial: /var/lib/dpkg/info/unison.prerm
Type: absent

Final: /var/lib/dpkg/info/unison.prerm
Type: normal
User: 0
Group: 0
Mode: 0755
Mtime: 1115595074
Hash: bd3527cd979ae52d3bec524cbd0e48a46981cdb6

Initial: /usr/bin/unison2.10.2
Type: absent

Final: /usr/bin/unison2.10.2
Type: normal
User: 0
Group: 0
Mode: 0755
Mtime: 1115595074
Hash: 45d29ef21934a7b4b7835fb72d13c26162586d26

Initial: /etc/alternatives/unison
Type: absent

Final: /etc/alternatives/unison
Type: symlink
User: 0
Group: 0
Mode: 0777
Mtime: 1116413501
Link: /usr/bin/unison2.10.2

Initial: /var/lib/dpkg/status
Type: normal
User: 0
Group: 0
Mode: 0644
Mtime: 1116413331
Hash: 344ec5e483998c6af074f712a82ea0ccbe4a4442

Final: /var/lib/dpkg/status
```

```
Type: normal
User: 0
Group: 0
Mode: 0644
Mtime: 1116413502
Hash: 90c2ea0938815584523e03ab661257fa533f08fa

Initial: /usr/share/doc/unison/changelog.Debian.gz
Type: absent

Final: /usr/share/doc/unison/changelog.Debian.gz
Type: normal
User: 0
Group: 0
Mode: 0644
Mtime: 1115595034
Hash: 48861ad2efcf8a18e3e34efccdc3109ffb82bc39

Initial: /var/lib/dpkg/info/unison.postinst
Type: absent

Final: /var/lib/dpkg/info/unison.postinst
Type: normal
User: 0
Group: 0
Mode: 0755
Mtime: 1115595074
Hash: ec032fa8ad5bba22599a22300ce720ee5d46bc03

Initial: /usr/share/doc/unison/README.Debian
Type: absent

Final: /usr/share/doc/unison/README.Debian
Type: normal
User: 0
Group: 0
Mode: 0644
Mtime: 1115595034
Hash: ebf1ab39690957a009c3f2ae01a1bb9bd6c06794

Initial: /var/lib/dpkg/info/unison.list
Type: absent

Final: /var/lib/dpkg/info/unison.list
Type: normal
User: 0
Group: 0
Mode: 0644
Mtime: 1116413500
Hash: 145e1b45eac08d3b8b53198cf39d6dd59bb0b889

Initial: /usr/share/doc/unison/changelog.gz
Type: absent

Final: /usr/share/doc/unison/changelog.gz
Type: normal
User: 0
Group: 0
Mode: 0644
Mtime: 1094498146
Hash: 2bd6bcf7218f85e2a5d74b1f18925eb6de04b060

Initial: /usr/share/doc/unison
Type: absent

Final: /usr/share/doc/unison
Type: directory
User: 0
Group: 0
Mode: 0755
Mtime: 1116413500
Empty: false
```

```
Initial: /usr/bin/unison
Type: absent

Final: /usr/bin/unison
Type: symlink
User: 0
Group: 0
Mode: 0777
Mtime: 1116413501
Link: /etc/alternatives/unison
```

Appendix D

Project Proposal

Robert McQueen
Corpus Christi College
ram48@cam.ac.uk

Part II Computer Science Project Proposal

Creating, Reverting & Manipulating Filesystem Changesets on Linux

Monday 18th October 2004

Originator: Peter Sewell
Special Resources: Superuser access to Linux machines for testing, including own desktop (AMD Athlon 2500+, 1GB RAM, 2x 60GB HDD) and laptop (Intel Pentium M 1.2GHz, 512MB RAM, 40GB HDD).

Project Supervisor: Peter Sewell
Signature:

Director of Studies: David Greaves
Signature:

Project Overseers: Richard Gibbens & Alan Mycroft

Introduction

System administrators of Linux systems often make changes to many parts of the filesystem which can be logically grouped together to represent a single task, such as installing or removing a particular package or piece of software, or making configuration changes to make a certain piece of hardware work with the system. This project intends to provide a system which can monitor the filesystem changes made by a given process, and compile these changes into a changeset, recording the state of files and directories before (files which are deleted or modified) and after (those which are created or modified) the changes are made.

As well as the monitoring mechanism itself, a set of utilities will be implemented to allow these changesets to be inspected, providing a log of changes that were made by the administrator, reverted, restoring the changed files to their original state, and later reapplied where appropriate, or sent to another system and applied there (beneficial in a cluster where the same change needs to be made repeatedly to similar machines).

A user interface will be provided for these operations, and could also include a mechanism to mask unrelated changes from a changeset before applying it or sending it to another system, or displaying in advance which changesets can currently successfully be applied on the system.

Giving the changeset applicator more understanding about the meaning of certain files on the system will allow changes to be applied even when the current system state doesn't match the state of the relevant files before the changes were made. For example, changes to text files could be represented as diffs and applied to the current versions of the files where possible, or some files which represent databases of items (such as the password file, or the package system database) can have changes applied to one entity regardless of changes to the others in the file.

Work to be Done

Planning

- Investigate the best way to monitor changes that are made to the filesystem by a process (and its children). Initial investigations will centre around the `strace` utility, and other programs (such as GNU `gdb`) that use the kernel's `ptrace` interface, to observe all of the system calls which are made, and hence record those system calls which modify the filesystem. I am focussing on this mechanism because it allows only relevant changes to be included in the changeset, and avoids time-consuming comparisons of the entire system. An alternative method such as an `LD_PRELOAD` library would not function correctly with setuid, setgid or statically linked binaries, so certain changes could be missed.

Furthermore, using `ptrace`, the system calls which modify the filesystem can be intercepted before they take place, and so files which are about to be deleted or overwritten can be preserved before the change is allowed to take place. Although there is a performance penalty with the `ptrace` interface (every syscall is intercepted whether or not it is relevant), and perhaps some level of complexity in the implementation, it is an existing mechanism which can be used to record all filesystem changes, without the need to modify the kernel.

- Examine packages from the Debian GNU/Linux distribution to investigate the necessity, and potential benefits, of a file changeset (versus package) based approach.
- Define a simple set of semantics to represent a UNIX filesystem, and use them to specify the criteria in which a filesystem changeset is applicable to a system that follows these semantics.

Monitoring System & Changeset Format

- Design an on-disk format to represent changes that have been made to the filesystem, containing files which have been deleted, created and modified, and sufficient metadata to efficiently determine the applicability of the contained changes to a given filesystem.
- Implement the monitoring system to create these changesets from observing the actions of running processes. This is likely to be implemented in C due to the necessity of interacting with the kernel directly for system calls and the `ptrace` interface.

Library and Tools to Manipulate & Apply Changesets

- Create a library using a higher-level language[1], which can load the metadata from the changesets, compare the changes to the current system to determine their applicability, and use the preserved files to apply and revert the changes. A utility for comparing filesystems (such as `rsync` or `unison`) can be used (or depending on requirements, implemented) to test & demonstrate the correct application and reversion of changesets.

- Implement utilities which use this library to inspect, apply and revert changesets at the command line.

- Implement a user interface to do these tasks interactively.

Further Extensions

The above sections are considered essential to a successful implementation of this project. However, as time permits, I hope to implement one or more of these extensions to enhance the utility of the software where possible:

- Implement an improved understanding of certain files on the system so that changes can be applied even if they are no longer in the same state that they were at the time the changeset was created. For example, textual configuration files can have their changes represented and applied as a diff from the GNU `patch` utility, even if they have changed, and certain files such as `/etc/passwd` or `dpkg`'s package database represent lists of discrete objects, so operations on them can be performed even if other entries in the file have changed.

- Improve the user interface to allow changes to be edited to remove irrelevant changes, creating new changesets, or to allow the user to selectively apply parts of an existing changeset.

- Implement code in the user interface to decide in which order changesets can be applied to the system, and which of the current changesets are applicable at the current time, and show this to the user, allowing them to group changes and apply them in order.

[1] Perhaps C# because of its similarity to Java, but unlike Java the Linux implementation (Mono) is free software, already provides bindings to UNIX-specific system calls which are essential for examining and modifying the filesystem in this project, and also has bindings to native user interfaces such as Gtk+ which are considerably simpler to use than Swing.

Special Resources

Superuser access is required to a Linux system to test the recording and application of changes. Several such systems are available to me including my own desktop and laptop PCs, and I may use User-Mode Linux, VMware or Xen to create sandboxed test systems. Working copies of code and dissertation will be stored on PWF and periodically saved to Pelican.

Starting Point

Existing code which monitors running processes (such as `gdb` and `strace`) will be investigated, as well as the Linux kernel source code, to determine the best way to use the `ptrace` interface to monitor the changes that are being made to the filesystem. Existing `LD_PRELOAD` programs such as `fakeroot` will be examined to evaluate this as an alternative monitoring strategy (although statically-linked programs would escape analysis) or mechanism of preserving files before changes are made. I am already familiar with the C programming language and the workings of Linux systems, and able to program Java and similar languages for higher-level functionality such as the user interface.

Success Criteria

A successful implementation of this system will:

- define a format for representing changes made to a filesystem,
- allow the filesystem changes made by a given process to be monitored to create filesystem changesets,
- provide a utility to inspect the contents and applicability of these changesets,
- provide a utility to apply and revert these changesets when it is valid to do so, returning the relevant files in the system to their state before or after the changeset was created, and
- implement a user interface for doing this interactively.

Time permitting, some of the above-mentioned extensions will be implemented to enhance the functionality of the system. By the progress report submission in February 2005, I aim to have implemented the core components, namely a mechanism to monitor changes, and utilities to inspect, apply and revert the resultant changesets, and have made a start on the user interface.

Plan of Work

Task (as described above)	**Deadline**
Explore monitoring strategies & decide on most practical method	November 5th 2004
Define filesystem semantics & applicability of changesets, define changeset format	November 19th 2004
Implement & test monitoring system	December 3rd 2004
Implement & test library to manipulate changesets	December 17th 2004
Implement & test tools for inspection & application of changesets	January 14th 2005
Implement & test initial user interface	January 21st 2005
Prepare progress report	February 4th 2005
Prepare & give presentation, extend user interface with extra functionality	February 18th 2005
Implement conflict resolution, complete outstanding coding	March 4th 2005
Complete introduction & planning sections	March 18th 2005
Complete implementation section	April 1st 2005
Complete evaluation & conclusions sections	April 15th 2005
Complete final draft of dissertation	April 29th 2005
Print & submit dissertation	May 13th 2005