University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

A Sophisticated Shell Environment

Hüseyin Pehlivan

A thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

June 2000

Abstract

An undo facility is an essential component of most interactive applications. In current operating system shells, whether textual or graphical, such facilities are typically very poor. Algorithms are presented for adding a recovery mechanism to a shell which allows previous commands to be selectively undone and redone, and previous versions of files to be recovered.

The recovery mechanism involves making the shell control resources in a more intelligent way. Programs are run under greater control, with the shell monitoring and analysing their resource requests. This provides better high level information to the shell and, for example, provides techniques to prevent foreign or untrustworthy programs from doing any damage, and to reduce problems with conflicting resource requests from concurrent programs.

A prototype implementation called `brush` has been constructed to investigate the convenience and natural feel of these facilities. It is command-independent and provided entirely at the user level. The idea is based on the interception of a particular class of system calls, using tracing facilities supported by many Unix operating systems. Recovery information required to implement selective undo is then stored according to the behaviour of the intercepted system calls. This doesn't involve any modification either to the kernel or to existing programs.

The recovery mechanism maintains three components in the file system for purposes of selective undo and redo: a history file which consists of user-typed commands, a record file which keeps all file I/O requests made by system calls, and a personal directory in which old versions of files are located. These components enable the system to be brought back to any desired state. Users can reach all past copies of a file, without resorting to system administrator assistance. The system state is kept consistent during recovery, employing a conflict-detection mechanism that checks for the applicability of selective undo/redo.

A shell environment with recovery facilities enables programs' interactions with the file system to be monitored very closely. Using this close monitoring, some Unix programs can be equipped with more sophisticated features. In this way, a *make*-like utility has been designed and implemented which provides automatic facilities in performing compilations of programs.

Dedication

$\mathcal{TO}$

my parents, Osman and Fitnat

$\mathcal{AND}$

my wife, Nermin

$\mathcal{AND}$

my boys, Burhan and Serhan

Acknowledgements

This work was undertaken at the Department of Computer Science at the University of Bristol under sponsorship of Karadeniz Technical University. First of all, I would like to thank the former and current vice-chancellors of Karadeniz Technical University; Prof. Aydın Dumanoğlu, who created such an opportunity and even led my registration with Bristol University in person, and Prof. Türkay Tüdeş, who helped keeping my study running smoothly, respectively.

I would like to express deep gratitude and thanks to my supervisor Dr. Ian Holyer his cooperation, enthusiasm and direction over four years. He has always been ready to deal with all the prototyping problems which are encountered throughout this thesis. Additionally, his ideas and comments on the first draft of the thesis were of great assistance.

I also thank Neil Davies for reading the first draft of my journal paper, Henk Muller who spared the time to answer my programming questions all the time, and John Gallagher for his valuable comments on one of my technical reports which assisted the last refinement of the thesis before its official submission. My thanks also go to the former members of the Bristol Functional Programming Group Eleni Spiliopoulou and Alastair Penney for short talks on programming.

Finally, I would like to thank my family and friends for all the moral support and encouragement, which were really invaluable and contributed to the emergence of this thesis. The unity and solidarity that they showed me are to be appreciated forever. Especially, I owe a lot to my unforgettable and namesake friend, Hüseyin Sağlam, who died two years ago. His humorous conversations will never be able to be forgotten.

Declaration

The work presented in this thesis is independent and original except where explicit reference in the text is indicated. No part of the thesis has been submitted for another degree or qualification of the University of Bristol or any other university or institution of education. The views expressed in the thesis are those of the author and in no way represent those of other authors.

Hüseyin Pehlivan

Contents

List of Figures

Chapter 1

Introduction

In the context of this thesis, a sophisticated shell is one which is equipped with recovery facilities, as well as features of conventional shells. A recovery ability is a crucial feature that many interactive single-user applications provide to allow the user to reverse the effects of previous commands. This capability of applications enables the user to recover from unintentional commands and repair any resulting damage at any point in the interaction. An undo facility, for instance in an editor, encourages a user to act more freely, without the fear of losing useful information.

A *shell* is a program which provides a user interface to an operating system. It may be a text-based command line interpreter as in Unix, or it may be a graphics-based file and process manager. Either way, the ability to repair damage to permanent resources such as files is an important one. The facilities which are typically provided at present are rather primitive, consisting of a "waste bin" directory where old versions of files are stored when explicitly deleted, together with various ad hoc backup mechanisms provided by individual applications. It is ironic that one can always undo the deletion of a single character in an editor, but not the deletion of a permanent file in a shell.

The aim of this thesis is to describe a way of designing and implementing a more intelligent shell which keeps track of versions of files on behalf of the user, together with information about how they were created or manipulated. This enables it to provide a more uniform and consistent mechanism for undoing the effects of commands, recovering old versions of files, repairing accidental damage, and otherwise managing a user's most permanent and valuable resources in a safe and convenient way.

Besides, such a shell creates a sophisticated working environment in operating systems. A user can run programs under ultimate control, monitoring their interactions with the file system. The control of file system activities of user programs allows many new applications to be developed and even existing ones to be upgraded with new facilities. For example, the Unix *make* program can be enhanced easily, whose design and implementation is the additional scope of the thesis.

1.1 Recovery facilities

This thesis describes a recovery facility for operating system shells which provides undo and redo facilities. To illustrate, suppose a user types the following sequence of commands in a Unix shell:

1) edit data
2) rm data
3) edit program

Now suppose that the user realises that it was a mistake to remove the file called "data", and the user wants to recover the file. We want the user to be able to issue an undo command in some way, e.g.

```
undo 2
```

This reverses the effect of the second command, and retrieves the deleted file. Later, the user may decide to redo the second command, so that it takes effect once more.

The undo command is generally required to support novice users. In an operating system, the invocation of undo can be needed on many kinds of undesired actions that vary from a mistyped command to an application program. In fact, undo presents some way of protecting users against themselves. For example, if the user intends to type

```
rm *.o
```

to remove all the files ending with .o (compiler-generated object files), but accidentally types

```
rm * .o
```

(note the space after the asterisk), *rm* will remove all the files in the current directory and then complain that it cannot find .o. In such a case, undo is a friendly utility which restores the removed files.

The techniques we develop for providing a suitable recovery facility lead to a number of other convenient facilities which can be added to a shell. The primary example of this which we develop in this thesis is a version `bmake` of the “make” utility for keeping versions of files, particularly compiled program components, up to date. Our version of make needs no hand-written description file; instead it monitors compiler commands to establish both the dependencies between files and the commands needed to re-create them on demand. For example, suppose a programmer compiles a source file named “myprog.c” as follows:

```
gcc -g -o myprog myprog.c
```

This compiling command runs under control of `bmake` during which the dependencies of the executable file, “myprog”, are determined. At a later stage, the programmer can issue the following command to produce an up to date version of the executable, probably after the source file is modified:

```
bmake myprog
```

where the dependencies are checked for any modification. If there is any dependency that has been modified since the compilation, `bmake` recompiles the source file. Programs which are made up of many source files can also be updated in a similar way. As will be seen in this thesis, the main advantage of `bmake` is that it can deal with composite programs easily and automatically.

1.2 Problems to be Solved

The provision of undo and redo facilities has been studied before, almost always in the context of editors or other self-contained programs. The literature on such systems is discussed in Chapter 3. However, no substantial work appears to have been done on undoing operating system commands. In an operating system context, there are a number of factors which make the provision of recovery difficult, and which we tackle in this thesis. These factors may also help to explain why recovery facilities of the kind we describe have not been implemented in the past.

Incorporating Recovery into an Existing System

With editors, recovery operations are designed into the system from the beginning. Operating systems do not take account of such facilities in their design. It is not practical to re-design an operating system from scratch. Thus, our task is to add recovery facilities to an operating system retrospectively.

Gaining Control over Commands

In an editor, the action of each user command is carried out interpretively under the direct control of the editor itself, so it is easy to adapt the behaviour of the command, e.g. to take a copy of some text before deleting it. In a shell, commands are usually treated as entirely separate programs, which are run completely independently of the shell. The shell then does not have any obvious direct control over the behaviour of the command. It only has external control, e.g. to pause or terminate it. This makes it difficult to adapt its behaviour, e.g. to take a backup copy of a file that it deletes, for example.

Finding out What Commands Do

In an editor, the intended result of each command is known to the editor program, simply because the commands are directly provided by the editor. In a shell, it is difficult to find out what the intention of a command is. Some information can be gleaned from the command line, which may mention the files which the command is to act on. However, this does not take account of implicit actions of a command, or actions which are determined dynamically by interaction with the user. Shell commands are typically very complex, and what is needed is the ability to monitor the effects of a command on the file system as they happen.

Complex Commands

In addition, the commonly available commands are typically very complex in their actions, and they differ in behaviour from one version of an operating system to another. It would be difficult to keep track of this.

Complex Actions

Commands can do a wide variety of different things, unlike in an editor where most commands can be explained by the replacing of one piece of text by another. Through a shell environment, a user can reach all system resources and modify them to the extent that the system allows. For example, the user has the right to change the appearance and accessibility of the file system considerably, as a result of re-arranging directory contents and access permissions. It would be essential to catch such actions.

Coping with New Commands

In an editor, the set of available commands is more-or-less fixed. In a shell, an arbitrary program can be compiled and installed as a new command at any time. Our recovery facility aims to be able to cope with addition of new commands at any time, without the need for any changes such as adding a description of the new command.

Concurrency & Non-determinism

Commands can be structured as a number of processes, or cooperate with existing daemon processes. It is possible to create processes both statically and dynamically. A process is serviced by the operating system almost independently of other processes, which gives rise to non-determinism. Concurrency may also cause inconsistent changes to the file system to occur. So the execution of concurrent processes must be kept under closer control to reduce usual non-determinism and inconsistency.

Flexibility

Files are large, so cannot be re-created by re-typing, as in an editor where if you delete a small amount of text, you may decide to retype it rather than undoing the delete. Thus an aim of our system is not just to be able to return to any previous state, but also to be able to retrieve any previously existing set of files. For example, suppose you delete one file, then spend a long time constructing a replacement. There needs to be a way to reach a position where both files are available. i.e. the command history must be able to be manipulated appropriately.

This requires selective undo, and the ability to insert new commands into the command sequence at arbitrary places. In a system where selective undo is provided, a command can be undone directly, without dealing with the undoing of any intervening command. With the insert facility, any desired state can be attained smoothly even if it conflicts with other past states. This helps reduce the number of conflicting situations.

Command Times

Commands can take a long time to run, so for efficiency, we want to be able to carry out undo and redo operations without ever re-running the original commands. We want the effects of undo and redo to be equivalent to the re-execution of a particular sequence of commands on the initial state. However, we want to achieve this purely by manipulating the file system. The undoing and redoing of a particular command are equivalent to the reversal and inclusion of its effects on the state, respectively. Therefore, in order to perform undo and redo operations efficiently, it is adequate that we reflect their corresponding effects on the system in some way, rather than executing new commands.

Storage Space

Files can take up a lot of storage, so we want to avoid unnecessary copies of files. For example, we don't want identical copies of files to build up, nor do we want file versions to be stored unnecessarily. For example, there is no need to keep backup copies of generated files, since a knowledge of dependencies allows them to be re-created. Clearly there is a tradeoff between time and space here.

Providing a Mental Model for the User

The design of our recovery facilities needs to be such that using them is as intuitive as possible. There needs to be a simple explanation of what happens when a command is undone, or when it is redone, and this explanation needs to be given in terms of the command sequence itself, relying on the user's knowledge and understanding of the available commands.

There needs to be a strong level of consistency. Some shells and programs keep backup copies of files, but we are looking for a much stronger system in which the relationships between file versions are kept consistent. This system must ensure that undo and redo commands operate on the right versions of files.

To provide sufficient flexibility, we need selective undo, making it easy to choose which commands to undo or redo. If undo and redo are to be possible in all circumstances, a tree-like model of versions is needed, as described in the theoretical models of undo. However, in the shell setting, this is too complicated to grasp easily, and so we propose a type of selective undo based purely on the linear sequence of commands, as normally held in a history list.

One consequence of allowing arbitrary selective undo/redo, but keeping to a linear model of versions, is that undo and redo are not always directly possible. There are situations where conflicts arise because of the dependencies between commands. These conflicts mean that an undo or redo would lead to an inconsistent or confusing state. We characterise these conflicts and show how to make them clear to the user. We also show that, in practice, they do not lead to impossible situations for the user.

On the other hand, the model for recovery which we present is concerned with users' recovery from their own prior commands. We are not concerned with data loss through hardware failure, for example, nor with the recovery of old versions of files from overnight file system dumps, nor with the backups which some application programs keep to guard against system crashes.

1.3 Achievements

To provide recovery facilities, we have developed techniques for tracing the execution of programs, monitoring their effect on the file system, adapting their behaviour as necessary, and establishing dependencies between them.

Operating System Independence

The ideas which we develop could be adapted to any operating system with suitable facilities. The minimum requirements of an operating system to be able to support a system like ours are described in Chapter 5.

Systematic Monitoring

Our demonstration implementation is based on tracing facilities provided in most versions of the Unix operating system. Previous work has used these facilities to provide, e.g., convenient local access to remote file systems, as described in Section 2.2.3. However, there appears to be no previous work which uses the facilities to monitor and record the activities of commands in a systematic way, as we do.

Flexibility

These techniques allow us to deal with most of the problems outlined in the last section. Our system can be added to an existing operating system without the need to adapt the kernel or involve the system administrator, and it can cope with a large and dynamically changing set of available user commands.

Transaction Based Approach

The essence of the technique is to monitor systematically all the relevant system calls made by a running program. For each system call, just before it is executed, extra calls are made as necessary to make backup copies of files and record information. The system calls made are recorded as a series of *transactions* on the file system, i.e. operations such as reading a file or writing to a file which are treated as indivisible. The effects of a command are reversed by reversing each of its transactions.

Dependency Tracking

The sequence of transactions corresponding to user commands allows us to determine the dependencies between commands, so that we can tell which commands create which versions of which files, and then find out which subsequent commands use those file versions.

Consistency

This ability to track the effects of commands and determine dependencies allows us to describe the consistency requirements necessary to provide a simple model of undo, to calculate the conflicts arising in the use of undo and redo, and to offer the user some support in getting around the conflicts.

Extended Facilities

It also allows us to contemplate a number of further facilities at the shell level. People have striven to meet some requirements for many years; for instance, executing programs under the user's own control, protecting the system against malicious programs, enhancing usability and learnability of the system, and reducing usual non-determinism. In this way, new command interpreters, such as restricted shells and secure shells, were implemented to restrict users' operations, and the integrity of the file system was protected against user actions and system crash via backup tapes. This emphasizes the value and significance of our work. In fact, each requirement does not have to be satisfied by a separate mechanism, although past work has focused on them separately. It is possible to replace all these separate mechanisms by a single one, which can be easily seen in this work.

Automatic Make Facility

The one which we develop furthest in this thesis is bmake. This keeps versions of programs up to date as with `make`, but without a hand-written description file. Instead, dependencies and re-compiling commands are calculated directly from the transactions associated with the command sequence. This means that using `bmake` is particularly easy, and it gives a much greater guarantee of consistency.

In addition, it is possible at any time to write out the current dependencies and re-compiling commands into a conventional description file, e.g. for source-level distribution of a product.

Shell Redesign Ideas

The ideas presented in this thesis emerged from the desire to design an operating system shell using a purely functional language such as Haskell. This involves redesigning the shell to remove as much non-determinism as possible. However, the ideas potentially have a wider application, so they are presented here without reference to functional programming. The aim is to achieve a shell with the following properties:

- Intelligent management of files and programs.
- The ability to undo commands and recover old versions of files.
- The ability to kill rogue programs cleanly.
- The ability to run foreign programs with minimal risk.
- The ability to run untrusted programs in a protected environment, as with Java's sandbox.
- The ability to run programs concurrently with clean sharing of resources.

Many of these points depend on the ability to undo and redo commands, and this is the aspect which we concentrate on in this thesis.

1.4 Overview of Thesis

This thesis basically concentrates on the creation of a more reliable Unix environment. As a whole, it describes the specification, design, and implementation of such an environment. Below, we disclose the scope of each chapter of the thesis.

Chapter 2 is a description of some relevant concepts in operating systems. It describes the mechanisms with which user processes can be monitored under great control, the command interpreters which establish the interaction between a user and the system, and the sophisticated programs which are used in program development environments.

In Chapter 3, we examine interactive undo support, mainly focusing on the user-oriented characterization of undo. This examination includes the kinds of user recovery support, the functionality of undo support in user interfaces, and a review of existing undo support models.

In Chapter 4, recovery commands and task-oriented commands are analysed. Comparing existing undo mechanisms, undo's role in transforming system states are examined. The chapter also conceptualises the undo command and classifies most common Unix commands.

Chapter 5 presents some practical ways in which the Unix system can be provided with recovery facilities. It introduces the recovery-related concepts and gives an analysis of the common Unix commands in terms of recovery, showing what a user's recovery from commands in operating systems is like. It also specifies how the Unix *make* program can be enhanced with automatic utilities, to produce a version which is called `bmake`.

In Chapter 6, we deal with the user-level design of a recovery mechanism that is embedded in a user shell. It defines the domain of undo support which will be provided and develops an efficient design model, resolving problems that it raises. Besides, it covers the design issues of the `bmake` utility.

Chapter 7 describes the implementation of the user shell with the facilities introduced in Chapter 6. It handles a separate implementation of each module of the shell and optimises the components of the recovery mechanism for purposes of efficiency. It also presents the implementation of the `bmake` program.

In Chapter 8, the current work concerning multi-user undo/redo is examined and a few models that propose such undo/redo are described. In addition, it points out how our shell is to be extended in order to cope with problems which are caused by concurrency and multiple users.

We conclude with Chapter 9 by summarising the current status of both the user shell and the `bmake` utility provided, and by giving some additional work required for them.

Chapter 2

Background

This chapter describes some facilities and their variations related to operating systems. In shell-based operating systems, users are most commonly served by shells. Shells create very convenient working environments in which users can reach all system resources. These environments are also suitable for writing and maintaining sophisticated programs. Modern operating systems not only support the services that a shell needs to act as a command interpreter, but they also provide tracing facilities to monitor processes started by the shell.

The purpose of the chapter is to discuss the following topics:

- tracing and controlling the execution of processes,
- the importance of shells and presenting existing shells,
- maintaining sophisticated programs.

This discussion is required to show later the implementation of a shell that is equipped with recovery commands. Besides, it helps prove that there are more functionalities in the environment created by the shell with the undo facility than in one created by an ordinary shell.

2.1 Process Tracing Facilities

The UNIX system provides two schemes for tracing processes: the *ptrace* system call and the */proc* file system. These schemes allow one process to monitor the execution of another process. The monitored process can be run step by step and its memory can be read and modified. Data can also be read from the process table. Old versions of the debuggers such as `gdb` and `dbx` are based on the *ptrace* system call. Some programs such as `ps` and `top` cooperate with the */proc* file system to control and inspect already-running processes.

2.1.1 Tracing with *ptrace* system call

The *ptrace* system call is used to obtain information from the kernel about a process [6]. A debugger process spawns a process to be traced and controls its execution with the *ptrace* system call, setting and clearing break points, and reading and writing data in its virtual address space. Process tracing thus consists of synchronization of the debugger process and the traced process and controlling the execution of the traced process.

```
if ((pid = fork()) == 0) {
    /* child - traced process */
    ptrace(0, 0, 0, 0);
    exec("name of traced process here");
}
/* debugger process continues here */
for (;;) {
    wait((int *) 0);
    read(input for tracing instructions);
    ptrace(cmd, pid, ...);
    if (quitting trace)
        break;
}
```

Figure 2.1: Structure of debugging process

The pseudo-code in Figure 2.1 shows the typical structure of a debugger program. The debugger spawns a child process, which invokes the *ptrace* system call and, as a result, the kernel sets a trace bit in the child process table entry. The child now *execs* the program being traced. For example, if a user is debugging the program *a.out*, the child would *exec a.out*. The kernel executes the *exec* call as usual, but at the end notes that the trace bit is set and sends the child a "trap" signal. The kernel checks for signals when returning from the *exec* system call, just as it checks for signals after any system call, finds the "trap" signal it had just sent itself, and executes code for process tracing as a special case for handling signals. Noting that the trace bit is set in its process table entry, the child awakens the parent from its sleep in the *wait* system call, enters a special trace state and does a context switch.

Typically, the parent (debugger) process would have meanwhile entered a user-level loop, *wait*ing to be awakened by the traced process. When the traced process awakens the debugger, the debugger returns from *wait*, *read*s user input commands, and converts them to a series of *ptrace* calls to control the child (traced) process. The syntax of the *ptrace* system call is

```
ptrace(cmd, pid, addr, data);
```

where *cmd* specifies various commands such as reading data, writing data, resuming execution and so on, *pid* is the process ID of the traced process, *addr* is the virtual address to be read or written in the child process, and *data* is an integer value to be written.

When executing the *ptrace* system call, the kernel verifies that the debugger has a child whose ID is *pid* and that the child is in the traced state and then uses a global trace data structure to transfer data between the two processes. It locks the trace data structure to prevent other tracing processes from overwriting it, copies *cmd*, *addr*, *data* into the data structure, wakes up the child process and puts it into the "ready-to-run" state, then sleeps until the child responds. On resuming execution (in kernel mode), the child does the appropriate trace command, writes its reply into the trace data structure, then awakens the debugger. Depending on the command type, the child may reenter the trace state and wait for a new command or return from handling signals and resume execution. When the debugger resumes execution, the kernel saves the "return value" supplied by the traced process, unlocks the trace data structure, and returns to the user.

2.1.2 Tracing with */proc* file system

The process file system provides access to the image of each process in the system. It represents all processes created in the system as pseudo files in a directory conventionally named */proc*. The name of each entry in the */proc* directory is a decimal number corresponding to the process id. The owner of each file is determined by the process's real user-id, but permission to open the file is more restrictive than traditional file system permissions.

Standard system call interfaces are used to access */proc* files: *open*(), *close*(), *lseek*(), *read*(), *write*(), and *ioctl*(). Data may be transferred from or to any valid locations in the process's address space by applying *lseek* to position the file at the virtual address of interest followed by *read* or *write*. A process file contains data only at file offsets that match valid virtual addresses in the process. I/O operations with a file offset in an unmapped area fail. I/O operations that extend into unmapped areas do not fail, but are truncated at the boundary. This includes writes as well as reads.

PIOCSTATUS	Get process status.
PIOCSTOP	Direct process to stop.
PIOCWSTOP	Wait for process to stop.
PIOCRUN	Make stopped process runnable.
PIOCSTRACE	Define set of traced signals.
PIOCSFAULT	Define set of traced machine faults.
PIOCSENTRY	Define set of traced syscall entries.
PIOCSEXIT	Define set of traced syscall exits.
PIOCGREG	Get values of process registers.
PIOCSREG	Set values of process registers.
PIOCSET	Set operation modes for process.

Figure 2.2: A list of *ioctl* operations

Information and control operations are provided through *ioctl*. A few of the *ioctl* operations are shown in Figure 2.2. This list is not exhaustive. Certain of these operations can be used only if the process file descriptor is open for writing. The SVR4 *proc*(4) manual page provides complete details. Also, a concise description of how to use the */proc* file system interface for tracing purposes is given by Faulkner et. al. [34].

A process executes in an environment established by and enforced by the Unix kernel. Natural points of control for a process are where it enters and leaves the kernel, specifically, system call entry and exit, machine faults, and receipt of signals. Events of interest are specified through the */proc* interface using sets of flags. Signals are specified using the POSIX signal set type, `sigset_t`. Machine faults and system calls are specified using analogous set types `fltset_t` and `sysset_t`.

The execution context of a process is stored in a data structure called `prstatus_t`, which a controlling process can request with an *ioctl* at any time. The structure is returned by the **PIOCSTATUS** requests or as an optional side-effect of the process-stop requests **PIOCSTOP** and **PIOCWSTOP**. The most significant fields of the structure are described below.

```
struct prstatus {
    short         pr_why;                   Reason for stop
    short         pr_syscall;               System call number
    short         pr_nsysarg;               Number of syscall arguments
    long          pr_sysarg[PRSYSARGS];     Arguments to this syscall
    pid_t         pr_pid;                   Process id
    prgregset_t   pr_reg;                   General registers
} prstatus_t;
```

A traced process stops when it encounters an event of interest. A stop on system call entry occurs before the system has fetched the system call arguments from the process. A stop on system call exit occurs after the system has stored all return values in the traced process's data and saved registers. This gives programs such as a debugger the opportunity to change the system call arguments and return values.

To take control of new processes spawned by the controlled process, a debugger can set the *inherit-on-fork* flag in the original process and arrange to trace exit from *fork*() and *vfork*() system calls. When the controlled process *fork*s, the child inherits all of the parent's tracing flags and both parent and child stop on exit from the *fork*. The debugger sees the parent's stop on exit from *fork* and uses the return value (the pid of the child) to open the child's */proc* file. Because the child stopped before executing any user-level code, the debugger can maintain complete control.

2.1.3 Comparison of *ptrace* and */proc*

The tracing facilities allow monitoring of a program's use of system services in an easily customizable manner. However, these facilities suffer several drawbacks. In particular, the use of *ptrace* for process tracing is quite primitive as depicted below.

- The kernel must do four context switches to transfer a word of data between a debugger and a traced process: The kernel switches context in the debugger in the *ptrace* call until the traced process replies to a query, switches context to and from the traced process, and switches context back to the debugger process with the answer to the *ptrace* call. This is the only way for the debugger to gain access to the virtual address space of the traced process. Consequently process tracing is slow and affects performance considerably.

- A debugger can trace several child processes simultaneously, although this feature is rarely used in practice. More critically, a debugger can only trace direct child processes: If a traced child *fork*s, the debugger has no control over the grandchild, a severe handicap when debugging sophisticated programs. If a traced process *exec*s, the later *exec*ed images are still being traced because of the original *ptrace*, but the debugger may not know the name of the *exec*ed image, making symbolic debugging difficult.

- A debugger cannot trace a process that is already executing if the debugged process had not called *ptrace* to let the kernel know that it consents to be traced. This is inconvenient, because a process that needs debugging must be killed and restarted in trace mode.

- The *ptrace* system call offers only very coarse-grained all-or-nothing tracing. This means that there is no way to trace a particular set of system calls. A debugger cannot trace a few system calls without tracing all the rest as well. As a result, having to trace all the system calls issued by the traced process gives rise to further loss of performance.

The */proc* file system removes all four disadvantages of *ptrace*. First, it is faster, because a debugger process can transfer more data per system call than it can with *ptrace*. Second, a debugger can trace arbitrary processes, not necessarily a child process. Third, the traced process does not have to make prior arrangement to allow tracing; a debugger can trace existing processes. Finally, */proc* allows fine-grained control of the traced process's memory, offering individual tracing of the system calls issued.

As part of the regular file system protection mechanism, only a superuser can debug processes that are *setuid* to root. Consequently, it is impossible to trace *setuid* programs, because users could violate security by writing their address space via the process tracing facilities and doing illegal operations. For example, suppose a *setuid* program calls *exec* with file name "privatefile". A clever user could use either *ptrace* or */proc* to overwrite the file name with "/bin/sh", executing the shell (and all programs executed by the shell) with unauthorized permission. *Exec* ignores the *setuid* bit if the process is traced to prevent a user from overwriting the address space of a *setuid* program.

2.2 Mechanisms Using Tracing Facilities

The approach of tracing system calls performed by commands or application programs emerged after standard operating systems were equipped with process tracing facilities. Currently, a few mechanisms have adopted this approach to implement new functionalities in favour of the system and its users. These mechanisms implemented at the user level manipulate the interface between user programs and the operating system services based on special system calls.

2.2.1 Interposition Agents

Although many contemporary operating systems provide low-level mechanisms for intercepting and handling system calls, it seems quite difficult to utilize these mechanisms effectively. To get around this problem, a user-level toolkit is presented by Jones [50].

The toolkit substantially increases the ease of interposing user code between clients and instances of the system call interface, allowing users to write such code in terms of the high-level objects provided by this interface, rather than in terms of the intercepted system calls themselves. This toolkit helps enable new interposition agents to be written, many of which would not otherwise have been attempted. An agent is a program that both uses and implements the system interface.

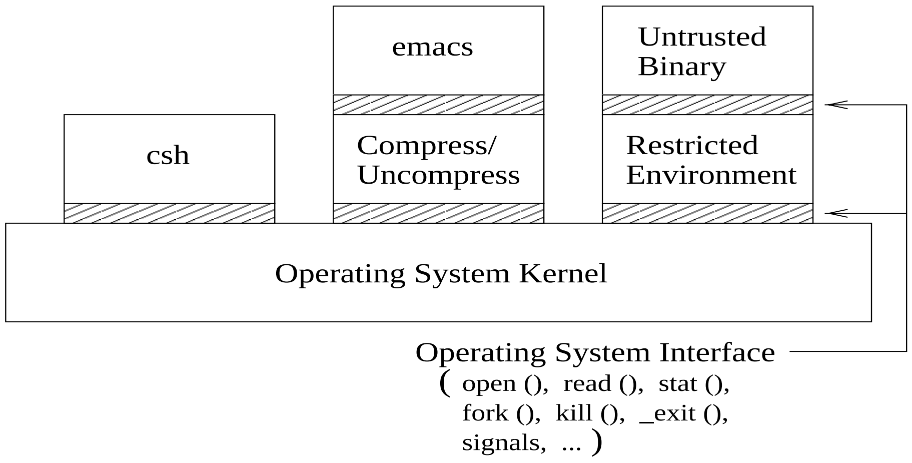
emacs
Untrusted Binary
csh
Compress/ Uncompress
Restricted Environment
Operating System Kernel
Operating System Interface
(open (), read (), stat (), fork (), kill (), _exit (), signals, ...)

Figure 2.3: Kernel and agents provide instances of system interface

The toolkit is implemented in C++ with small amounts of C and assembly language. Thus, it presents as objects in an object-oriented manner the system interface abstractions that include pathnames, directories, open objects referenced by descriptors, and signals, as well as the system calls themselves. It also support the substantial commonalities present in different agents through code reuse.

Figure 2.3 (adapted from [50]) depicts uses of the system interface with interposition, where both the kernel and interposition agents provide instances of the operating system interface. Using the toolkit objects, it is possible to transparently interpose user code (an agent) that handles, for example, compression between an editor such as `emacs` and the kernel. The code can allow users to compress or uncompress files automatically, when they are to be saved or called back to the editor.

The toolkit is built on top of the Mach 2.5 system call interception mechanism. Since the interception mechanism redirects system calls to handler routines in the same address space, interposition agents reside in the same address space as the applications running on them. This makes every agent reimplement some system calls (e.g. *execve*() and *fork*()) to allow the application's portion of the address space to be reloaded, while maintaining the agent's portion.

2.2.2 Detecting Intrusions

Intrusion detection systems rely on a wide variety of observable data such as system-call data. These systems prevent intrusions from either local or remote users, evaluating the legitimacy of their activities. Many solutions to detecting and preventing intrusions are based on the interception of system calls [33, 40, 46, 55, 95]. Below, an application that monitors the execution of network programs is described.

Network programs often rely on helper applications to retrieve and process various kinds of information from the Internet. These helper applications become security-critical, particular when they handle untrusted data. Considering all helper applications untrusted, a secure environment to contain them, named `Janus`, is implemented by Goldberg et. al. [40].

`Janus` is intended to protect all resources on a user's system from malicious data that subverts the document viewer. For example, when a user downloads a Postscript document from a remote network site, it may be automatically handled by *ghostview*, which effectively allows almost arbitrary 'code' in a document because PS is a language. Since that downloaded data or programs could be under adversarial control, it is completely untrustworthy. Therefore, `Janus` confines the untrusted software and data by restricting the program's access to the operating system. This is achieved by intercepting and filtering dangerous system calls via the Solaris process tracing facility. In this way, the risk of a security breach is reduced.

In order to protect the system against an untrusted program, the program is not allowed to access any part of the system or network for which `Janus` has not granted it permission. Initially, the helper application is placed in a particular directory; it cannot *chdir* out of this directory, however it has full access to files in or below this directory. It also has read access to certain carefully controlled files referenced by absolute pathnames, such shared libraries and global configuration files. When `Janus` encounters an application that attempts to make a system call denied by the security policy, the application is entirely killed.

Helper applications are able to *fork*() children. *Janus* recursively traces them. That is, each child that the application creates is traced by creating a new child which then detaches the application and begins tracing the application's child. This method safely allows the traced application to spawn a child by ensuring that all children of untrusted applications are traced as well.

2.2.3 Extending operating systems

Extensibility is a very important issue in modern operating systems. Extendible operating systems can easily adapt to support new protocols or satisfy the requirements of new emerging applications. Alexandrov et. al. used the process tracing facilities as a new method of extending existing operating systems, building `Ufo` (User-level File Organizer) [2].

`Ufo` implements a global file system which extends a wide range of the functionality of a standard Unix operating system (Solaris). It is originally implemented for users who have access to multiple computers that are geographically distributed. The sources from the large number of existing HTTP and anonymous FTP servers are presented to users as if they were local files. This enables all local applications to access remote files transparently.

In `Ufo`, system calls that an application executes are intercepted by the *Catcher*, a user-level layer inserted between the application and the operating system. This layer allows the user's environment to be controlled by modifying the behaviour of the system calls that operate on remote files. To provide new functionality, the layer also issues additional service requests. For a system call that needs a remote file, `Ufo` itself downloads the file from the remote site.

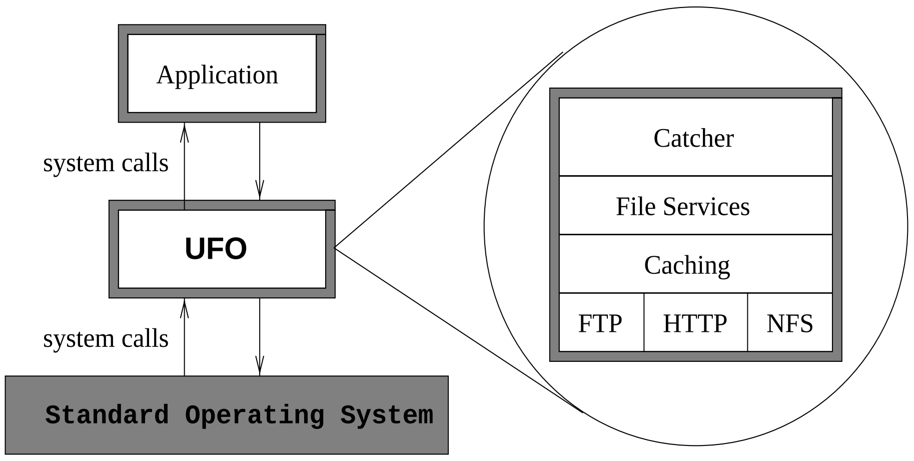

Figure 2.4: General architecture of Ufo

Remote machines are connected via authenticated and anonymous FTP and HTTP protocols (see Figure 2.4, adapted from [2]).

`Ufo` uses whole file transfers to and from the remote file system. Instead of downloading a file each time the user opens it for reading, local copies of previously accessed files are cached to achieve reasonable performance. `Ufo` can reuse the local copy on a subsequent access, as long as it is up-to-date.

Whenever, for example, the *open* system call is intercepted which accesses a remote file, `Ufo` first ensures that an up-to-date copy of the file exists in the cache. Then it redirects the system call to the local copy and allows it to proceed. System calls which only access local files are allowed to proceed unmodified.

2.3 Operating System Shells

Users need a means of communicating with operating systems in a favourable way. To meet this requirement, a particular program is provided to manage the interaction between users and the operating system. This program, which is the outermost layer of the operating system, prompts users for input, interprets that input for the operating system, and then handles any resulting output from the operating system.

2.3.1 What is a Shell?

The interface to the operating system is called a *shell*. At its base, a shell is simply a macro processor that executes commands. A Unix shell is both a command interpreter, which provides the user interface to the rich set of Unix utilities, and a programming language, allowing these utilitites to be combined. The shell reads commands either from a terminal or a file. Files containing programs can be created, and become commands themselves. These new commands have the same status as system commands in directories like '/bin', allowing users or groups to establish custom environments.

A shell allows execution of Unix commands, both synchronously and asynchronously. The redirection constructs permit fine-grained control of the input and output of those commands, and the shell allows control over the contents of their environment. Unix shells also provide a small set of built-in commands (builtins) implementing functionality impossible (e.g., *cd*, *break*, *continue*, and *exec*), or inconvenient (*history*, *getopts*, *kill*, or *pwd*, for example) to obtain via separate utilities. Shells may be used interactively or non-interactively: they accept input typed from the keyboard or from a shell script. A shell script is a sequence of shell and operating system commands that is stored in a file.

The shell has the ability to interpret the commands typed and the keys pressed. It takes a command line and evaluates it according to a fixed set of rules in order to direct the operating system to take an appropriate action.

The shell recognizes three types of commands [6]. First, a command can be an executable file that contains object code produced by compilation of source code. Second, a command can be an executable file that contains a sequence of shell command lines. Finally, a command can be an internal shell command. The internal commands make the shell a programming language in addition to a command interpreter.

While executing commands is essential, most of the power (and complexity) of shells is due to their embedded programming languages. Like any high-level language, the shell provides variables, flow control constructs, quoting, and functions. These utilities are indispensable to system programmers particularly.

Whenever logging in to the system, a user is placed in an environment controlled by the shell program. This is a process which has been started (spawned) at the end of the login process. A new shell is also started for each invocation of a terminal window.

2.3.2 Shell Features

As mentioned in the previous section, shells offer features geared specifically for interactive use rather than to augment the programming language. A shell can provide users with one or more of the following features. Users can

- create an environment that meets their needs,
- write shell scripts,
- define command aliases,
- edit the command line.
- manipulate the command history,
- automatically complete the command line,
- run lengthy tasks in the background,
- store data in user-defined or shell-defined variables,
- link any number of commands together (piping),
- redirect program input and output,

These features are the primary advantages of interfacing to the system through a shell. Unfortunately, shells don't have a standard way of providing them. A single feature (or capability) can be provided in different ways by different shells, which destroys compatibility. A script file written in a shell's programming language, for instance, may not run in another shell as a result of the syntax variation.

2.3.3 How a Shell Works

The shell has the terminal as standard input and standard output. It starts out by typing the `prompt`, a character such as a dollar sign ($). When the user types a command line at the prompt, the shell extracts the first word from the command line and assumes it is the name of a command to be run, and all the remaining words are arguments.

If the command is a built-in, then the shell executes it internally without creating new processes. Otherwise, the shell forks and creates a child process and runs the command as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next command line.

The user can specify that standard output be directed to a file by typing, for example,

```
date > file
```

Similarly, standard input can be redirected, as in

```
sort < file1 > file2
```

which invokes the *sort* program with input taken from `file1` and output sent to `file2`.

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

```
cat file1 file2 | sort
```

invokes the *cat* program to concatenate two files and send the output to *sort* to arrange all the lines in alphabetical order.

If the user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently,

```
cat file1 file2 | sort &
```

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on.

These capabilities are not all that a shell is supposed to provide. The shell can also recognize a number of other metacharacters as special; for example, the most commonly used is the asterisk $*$ which tells the shell to search the directory for filenames in which any string of characters occurs in the position of the $*$. Table 2.1 [51] shows a list of some shell metacharacters.

>	*prog* > *file*	Direct standard output to *file*
>>	*prog* >> *file*	Append standard output to *file*
<	*prog* < *file*	Take standard output from *file*
|	*p1* | *p2*	Connect *p1* to *p2* via pipe
<<	*prog* << *str*	Read standard input, up to next *str*
;	*p1* ; *p2*	Do *p1*, then *p2*
&	*p1* & *p2*	Like ; but doesn't wait for *p1* to finish
&&	*p1* && *p2*	Run *p1*; if successful, run *p2*
||	*p1* || *p2*	Run *p1*; if unsuccessful, run *p2*
(...)	(*progs*)	Run *progs* in a subshell
*	*prog ch**	Match any string of characters beginning with *ch*
?	*prog ch?*	Match any single character beginning with *ch*

Table 2.1: Shell Metacharacters

2.4 User-Related Shells

The advantages that shells provide in interacting with operating systems have arisen out of many new shell implementations with various functionalities. There is currently a number of user shells that have been in widespread use. Most of these shells run on Unix platforms and have many common features.

2.4.1 Standard Shell

The standard shell (`sh`), often known as the Bourne shell, is the original UNIX shell and available on all UNIX systems. Since being the oldest of the shells currently in use, it offers a very limited number of functionalities and so is weak in user conveniences. The initialisation files, `/etc/profile` and `.profile`, are read and executed each time the shell is invoked.

The most important functionality is the facility of writing shell scripts. Bourne shell provides an adequate programming environment, with a powerful syntactical language built in to it. The language includes all the features that are commonly considered to produce structured programs; it has particularly strong provisions for controlling input and in its expression matching facilities. But one major drawback is that the shell does not have the interactive facilites provided by modern shells such as the C shell and Korn shell. For the interactive user, there is no way to repeat or re-edit previous commands, and to control background jobs. Some interactive facilities are implemented in more recent versions of `sh`, though.

As an interface to the standard shell, the job control shell (`jsh`(1)) is developed which has all of the functionality of `sh` and enables job control. Every command the user enters at the terminal is called a job. Each job can be run in the foreground or background and stopped for a while. In fact, this is a common feature of all the user shells, excluding the standard shell.

2.4.2 C Shell

The C shell (`csh`) is an alternative to the Bourne shell. It provides a number of convenient features for interactive use that are not available with `sh`. These features incorporate a history mechanism, job control facilities, interactive file name and user name completion, and command aliasing. It also offers a C like language with which to write shell scripts.

The C shell begins by searching the user's home directory for the `.cshrc` and `.login` files. Both files contain any customized user information pertinent to running the C shell. Note that the `.login` file is an equivalent of the Bourne shell file `.profile`. Commands from the `.login` file are only executed at login time. The `.cshrc` file is executed both at login time and every time the csh command or a C shell script is invoked. The `.cshrc` file is generally used to define C shell characteristics like aliases and C shell variables (for example, `history`, `noclobber`, or `ignoreeof`). Before logging the user off the system the C shell processes the commands stored in the `.logout` file, if the file exists.

Among common shells, only the C shell imposes limitations such as

- Words can be no longer than 1024 bytes.
- Argument lists are limited to ARG_MAX bytes, which are found in the `/usr/include/limits.h` file.
- Command substitutions can substitute no more bytes than are allowed in an argument list.

- To detect looping, the shell restricts the number of alias substitutions on a single line to 20.
- The csh command does not support file name expansion based on equivalence classification of characters.

The C shell normally ignores quit signals. Jobs running detached are immune to signals generated from the keyboard (INTERRUPT, QUIT, and HANGUP). The user can control the shell's handling of INTERRUPT and TERMINATE signals in shell procedures with *onintr*, although the C shell doesn't support full signal trap handling.

2.4.3 Korn Shell

The Korn shell [54] (`ksh`) is a superset of the Bourne shell, which is suitable for interactive use. It is basically intended to remove non-standard features of both the Bourne shell and the C shell. Conforming to the POSIX standard, it is provided with standard features. So it is also known as the POSIX shell (`psh`).

However, it contains many of the same features as the Bourne and C shells together with a shell programming language similar to that of the Bourne shell. These features incorporate I/O redirection capabilities, variable substitution, and file name substitution. Each invocation of `ksh` initially runs a (script) file referred to by the ENV variable. There is another file, `.profile`, executed at session startup.

2.4.4 Other Shells

Most recent developments in shells have focused on improving the interactive environments further, causing the appearance of some other shells. These shells are `bash`, `tcsh`, `zsh`, `wish`, `rc`, `es`, and `fsh`.

The Bourne-again shell (`bash`) is an `sh`-compatible shell that incorporates useful features from the Korn shell and the C shell. It offers functional improvements over `sh` for both interactive and programming use. Widely used within the academic community, `bash` is quite portable and currently runs on nearly every version of UNIX and a few other operating systems.

The Tenex C shell (`tcsh`), an enhanced version of the C shell, is completely upward compatible with the C shell. Capabilities like spelling correction and programmable command completion are added to the interactive environment. It features a scrollable command history list with emacs style editing.

The Window-based shell [42] (`wish`) is a graphical command interpreter for the X window system. It takes advantage of the observed locality of commands and file name arguments to reduce typing. `Wish` automatically displays file system information that users would otherwise need to request with commands. The command shell provides displays for listing the contents of directories and for browsing through the directory system hierarchy. It also allows invocation of commands and file arguments by pointing and "clicking" with the mouse.

`Rc` [31] is a command interpreter, originally developed for the Plan9 system. It appeared after the problem of porting the Bourne shell to Plan 9 and also is implemented under Unix. Being a powerful scripting language, it provides similar facilities to the Bourne shell, with some small additions and mostly less idiosyncratic syntax.

`Es` [43], which bears a close resemblance to `rc`, presents new programming constructs. In `es`, almost all standard shell constructs (e.g. pipes and redirection) are translated into a uniform representation: function calls. The primitive functions which implement those constructs can be manipulated the same way as all other functions: invoked, replaced, or passed as arguments to other functions.

The functional shell [62] (`fsh`) is designed to support a functional approach to commands. It employs the syntax and semantics of functional programming languages, unlike the other shells in Unix, which typically do that of imperative programming languages. Programs and functions are treated consistently as objects requiring evaluation. In `fsh` there is a set of special types such as `integer`, `string`, and `boolean` provided by following the functional paradigm. This set makes the writing of a script file easier.

The Z shell (`zsh`) is the last shell designed for interactive use. It most resembles the Korn shell among the standard shells, although it is not completely compatible. Many of the useful features of `bash`, `ksh`, and `tcsh` are incorporated into `zsh`. These features are supported with enhancements of many types, notably the command-line editor and options for customising its behaviour.

Another two shells are `Oonix` and `R-shell`. `Oonix` [20] offers the user an iconic, object-oriented environment, representing files and subdirectories as icons in a directory window. It also has the capability of interacting with the window system on the host machine. `R-shell` [94] is an application-oriented extension of the regular Unix shell which provides robotics related data manipulation and world modeling capabilites. Its main purpose is to assist in the preparation of input data for the testing and evaluation of new algorithms in the area of Robotics.

As seen in the above descriptions, the shells are based essentially on a few previous ones, either emulating the existing capabilities or adding new capabilities. However, the features provided can vary considerably. Table 2.2 lists significant features of the shells, so that the user can make a comparison [14].

The abbreviations (letters and numbers) used in the table should be interpreted as follows:

Y	Feature is present in the shell.
N	Feature is not present in the shell.
F	Feature is present only if the shell's function mechanism is used.
L	Feature is enabled if the readline library is linked into the shell.
1	This feature was not in the original version, but has since become almost standard.
2	This feature is fairly new and so is often not found on many versions of the shell.
3	This can be done only by specifing a file via the ENV environment variable.

2.5 Security-Related Shells

Since security is an important concept in interactions with operating systems, a few more shells are implemented that restrict users' actions. Of these shells, remote shell and restricted shell are represented with the same command name (`rsh`) in some Unix machines, which are normally `/usr/bin/rsh` and `/usr/lib/rsh` respectively. To distinguish between them, the former is called `rsh` and the latter `Rsh` below.

	sh	csh	ksh	bash	tcsh	zsh	rc	es
Job control	N	Y	Y	Y	Y	Y	N	N
Aliases	N	Y	Y	Y	Y	Y	N	N
Shell functions	Y(1)	N	Y	Y	N	Y	Y	Y
“Sensible” I/O redirection	Y	N	Y	Y	N	Y	Y	Y
Directory stack	N	Y	Y	Y	Y	Y	F	F
Command history	N	Y	Y	Y	Y	Y	L	L
Command line editing	N	N	Y	Y	Y	Y	L	L
Login/Logout watching	N	N	N	N	Y	Y	F	F
Name completion	N	Y(1)	Y	Y	Y	Y	L	L
History completion	N	N	N	Y	Y	Y	L	L
Fully programmable Completion	N	N	N	N	Y	Y	N	N
Co Processes	N	N	Y	N	N	Y	N	N
Builtin arithmetic evaluation	N	Y	Y	Y	Y	Y	N	N
Periodic command execution	N	N	N	N	Y	Y	N	N
Spelling Correction	N	N	N	N	Y	Y	N	N
Process Substitution	N	N	N	Y(2)	N	Y	Y	Y
Underlying Syntax	sh	csh	sh	sh	csh	sh	rc	rc
Checks Mailbox	N	Y	Y	Y	Y	Y	F	F
Large argument lists	Y	N	Y	Y	Y	Y	Y	Y
Non-interactive startup file	N	Y	Y(3)	Y(3)	Y	Y	N	N
Non-login startup file	N	Y	Y(3)	Y	Y	Y	N	N
Specify startup file	N	N	Y	Y	N	N	N	N
Low level command redefinition	N	N	N	N	N	N	N	Y
List Variables	N	Y	Y	N	Y	Y	Y	Y
Full signal trap handling	Y	N	Y	Y	N	Y	Y	Y
Local variables	N	N	Y	Y	N	Y	Y	Y

Table 2.2: Characteristics of user shells’ features

2.5.1 Trusted Shell

The trusted shell (`tsh`) is a limited version of the Korn shell started with the `tsh` command and provides greater security than the Korn shell. Generally, a user calls the `tsh` shell by pressing Ctrl-X, Ctrl-R, the secure attention key (SAK) sequence, after a login. The `tsh` shell also can be invoked by defining it as the login shell in the `/etc/passwd` file.

To use the SAK sequence to invoke the trusted shell, the terminal the user is using must have SAK enabled, and the user must be allowed to use the trusted path. To exit from the `tsh` shell, three built-in commands (*logout*, *shell*, and *su*) are allowed that either end the login shell or execute another one.

The trusted shell differs from the Korn shell in the following ways:

- The function and alias definitions are not supported. Alias definitions are only supported in the `/etc/tsh_profile` file.
- The IFS and PATH environment variables cannot be redefined.
- Only trusted programs can be run from the tsh shell.
- The history mechanism is not supported.
- The only profile used is the `/etc/tsh_profile` file.

2.5.2 Remote Shell

The remote shell (`rsh` or `remsh`) allows a user to execute a command or a shell script on a remote machine. This doesn't involve having to log onto that machine initially via commands such as *rlogin* and *telnet*. The `rsh` command directly forwards the command to be run to the remote machine. It copies its standard input to the remote command, the standard output of the remote command to its standard output, and the standard error of the remote command to its standard error.

As a result of connecting the local terminal to the remote host, the output of the remote command is displayed on the local terminal as if it were produced locally. Interrupt, quit and terminate signals are propagated to the remote command; rsh normally terminates when the remote command does. Shell metacharacters which are not quoted are interpreted on local machine, while quoted metacharacters are interpreted on the remote machine. If a remote command that `rsh` is to run is omitted, an rlogin session is started.

The `rsh` command can be only used by users who have an account on the remote system. It runs in a user's own home directory on the remote machine. To execute a command that cannot be run in that directory, the user must give the path name relative to the home directory or issue the *cd* command to change the directory to it.

There are serious security issues involved in executing a command remotely. For such a request it is necessary for the remote machine to trust the local machine. In order for `rsh` to work successfully, therefore, there must be an appropriate file called `.rhosts` or `/etc/hosts.equiv` on the remote machine. These files contain machine names and user IDs. If an illegal access is attempted, `rsh` will issue an error message and quit.

2.5.3 Restricted Shell

The restricted shell (`Rsh`) is used to set up login names and execution environments whose capabilities are more controlled than those of an ordinary shell. This scheme assumes that the end-user does not have write and execute permissions in a directory containing commands. Imposing a limited number of commands, `Rsh` controls the user's environment.

Alternatively, *sh* also opens a restricted shell if it is started with the '-r'. The behaviour of the restricted shell is identical to *sh* with the exception that the following actions are disallowed:

- Changing the directory with the *cd* builtin,
- Setting or unsetting the values of `PATH` or `SHELL` variables,
- Specifying path or command names containing a / (slash),
- Redirecting output with the redirection operators such as '`>`', '`>&`', and '`>>`',
- Using the *exec* builtin to replace the shell with another command.

The restrictions above are enforced after the `.profile` file is interpreted. Therefore, the writer of the `.profile` file has complete control over user actions by performing setup actions and leaving the user in an appropriate directory. An administrator can create a directory of commands in the `/usr/rbin` directory that the `Rsh` command can use by changing the `PATH` variable to contain the directory.

The restrictive policy of `Rsh` is not reliable as necessary. With `Rsh`, a system can't protect itself against users' actions properly. Although a command name is not allowed to contain "/", the user can specify arguments containing it. For example, the user can use editors to edit or view any file with read access allowed on the system. Also, some programs, such as editors and telnet, enable a user to escape out to a shell and to run an unauthorized command.

2.5.4 Secure Shell

The secure shell (`ssh`) is designed to provide strong authentication and secure communications over insecure channels [21]. It allows users to log into another computer over a network, to execute commands in a remote machine, and to move files from one machine to another. Thus, it can be used to directly replace the functions of *rsh*, *rcp*, *rlogin*, and *rdist*.

The main security improvements `ssh` provides are as follows:

- Several security holes (e. g., IP, routing, and DNS spoofing) are closed.
- All communications are automatically and transparently encrypted.
- The `DISPLAY` environment variable used in secure X11 sessions is automatically set on the server machine, forwarding any X11 connections over the secure channel.
- Arbitrary TCP/IP ports can be redirected through the encrypted channel in both directions (e.g. for e-cash transactions).
- The insecure programs are automatically replaced with secure ones for the users.
- The client is customizable in system-wide and per-user configuration files.

- The software can be installed and used (with restricted functionality) even without root privileges.

The `ssh` command foils sniffing attempts by encrypting the packets and by only allowing connections with known machines. The client and the server each use RSA public key technology to authenticate the other. The client authenticates the server at the start of each connection, and the server authenticates the client before it allows `.rhosts` or `/etc/hosts.equiv` access. If the remote machine doesn't support `ssh` it will then fall back to using conventional *rsh*, after first informing the user that the communication will not be encrypted.

Users can create private authentication keys. In general, pure RSA authentication never trusts anything but the private key. The secure shell is one step toward a more secure network as a result of helping prevent possible malevolent attacks on networks. It doesn't close all network security holes. Should an attacker gain root access on a local machine, the users cannot be protected.

2.6 Utilities for Maintaining Programs

During the development of sophisticated programs, many programmers employ a crucial utility, often called *make*. As a recompilation program, it provides an elegant method for the compilation of programs which are made up of many components, written in any programming language. To date, many extensions of `make` have been produced to increase its utilization by using very attractive techniques. These extensions equip `make` with more powerful features and remove some arbitrary restrictions. This section describes the popular extensions of `make`.

2.6.1 Original Make

The `make` utility was first developed at the Bell Laboratories [35]. It keeps track of the relationships between parts of a program, and issues the commands needed to make the parts consistent after changes are made. The utility is primarily used in software development environments for creating up-to-date executable programs from their source files. The strength of `make` is its sensitivity to hierarchies of dependencies, such as source-to-object and object-to-executable [70].

Programmers of any programming language normally split up large programs into multiple source files. This is often done so that a change to one source file only requires one file to be recompiled and not all the files. With the *make* program, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work. Besides compilers, `make` is routinely used to invoke generators (e.g., *yacc* and *lex*), linkers, text formatters, and many other tools.

In order to be able to use `make`, the programmer is responsible for creating a *description file*, generally called *makefile* or *Makefile*, which describes the dependency graph. The description file contains a number of targets, the sources that they depend on, and the commands necessary to construct the targets from the sources. In deciding which files to build and how to build them, `make` draws on the description file and the last-modified times of the files specified in this file, sticking to a set of given built-in rules.

```
a.out: codegen.o parser.o library         # dependency line
        gcc codegen.o parser.o library    # link
codegen.o: codegen.c codegen.h
        gcc -c codegen.c                  # compile
parser.o: parser.c
        gcc -c parser.c                   # compile
parser.c: parser.y
        yacc parser.y                     # generate y.tab.c
        mv y.tab.c parser.c               # rename y.tab.c
```

Figure 2.5: A simple description file

The way the *make* program normally works is simple. After changing some source files, the programmer starts *make*, which examines the times at which all the source files and targets were last modified. If a source file has an earlier modification time than the corresponding target, no compilation is needed. Otherwise, *make* knows that the source file has been changed since the target was created, and thus the source file must be recompiled. In this way, *make* goes through all the source files to find out which ones need recompiling, and calls the compiler to recompile them.

Figure 2.5 [35] shows a complete description file. To generate `a.out`, there are four primary files used, `parser.y`, `codegen.c`, `codegen.h`, and `library`. One command uses the output of the other one. For example, the parser generator (`yacc`) transforms `parser.y` into `y.tab.c`, which is an input file for the code generator (`gcc`). A change to `parser.y` requires regenerating `parser.c`, then recompiling and reloading. However, a change to `library` requires reloading but no compiling.

`Make` has a rudimentary macro substitution mechanism, although Figure 2.5 doesn't include any macro definition. A macro invocation is denoted by a warning character ("$"). User macros may be defined either in the description file or on the `make` command line. Command line values override the description file values. `Make` also has a few macros of its own, and these are the only ones whose values change as it operates. Using macro definitions, the default rule for transforming a C source file to an object file is

```
.c.o:
        $(CC) $(CFLAGS) -c $<.c
```

CC is a static macro that has as default value the name of the C compiler. `CFLAGS` is initially null, but can be changed to provide special compilation flags. `$<` is the name of a dynamic macro that is the prefix of the name used to invoke this rule. Thus, if `make` needs to compile `xyz.c`, it issues the command

```
cc -c xyz
```

In `make` an issued command is not always run in a user shell environment. Therefore, a disadvantage is that command *aliasing* and shell *functions* are not preserved in update command blocks. The `make` program uses either the *system* or *exec* call to execute each command line. If a line contains shell metacharacters (`$, |, (), ><`) then it is sent to the shell via *system*, otherwise the command is executed via a *fork* and *exec*. *Aliases* in the latter case are ignored by `make`.

2.6.2 Mk

`Mk` [49] is an enhanced version of the original `make`. Its flexible rule specifications, implied dependency derivation, and parallel execution of maintenance actions are well-suited to the Plan 9 environment.

`Mk` supports program construction in a heterogeneous environment and exploits the power of multiprocessors by executing maintenance actions in parallel. It interacts seamlessly with the Plan 9 command interpreter *rc* to carry out necessary jobs and accepts pattern-based dependency specifications that are not limited to describing rules for program construction. Compared to `make`, `mk` is equipped with several new capabilities. The most important ones are that `mk`

- constructs the entire dependency graph for each target.
- performs transitive closure on metarules.
- supports virtual targets that are independent of the file system.
- allows non-standard out-of-date determination.

To use parallelism and to deal with, for example, several *yacc* commands in a single `mk` run, the programmer has to take care of name clashes explicitly. An implicit rule for creating a linkable object file out of a *yacc* specification file is

```
%.o: %.y
   mkdir /tmp/$nproc; cp $stem.y /tmp/$nproc
   (cd /tmp/$nproc; yacc $stem.y; myy.tab.c $stem.c)
   $CC $CFLAGS -c /tmp/$nproc/$stem.c
   rm -rf /tmp/$nproc
```

Here `$nproc` is the identifier of the currently running process.

The macro `$NPROC` specifies the maximum number of simultaneously executing commands. Normally it is imported from the Plan 9 environment, where the system has set it to the number of available processors. It is programmer-settable, and setting it to 1 implies serial execution of commands. `Mk` has no provision for the mutually exclusive execution of commands, although commands can be executed one after another, using serial execution.

2.6.3 Optimistic Make

Optimistic `make` [19] is identical in functionality to `make`. However, unlike the conventional *pessimistic* implementation of `make`, it begins execution of the commands necessary to bring the targets up-to-date before the user issues the `make` request. Outputs of the optimistically executed computations (such as file or screen update) are concealed until the user types `make`. If the inputs read by the optimistic computations have not been changed by the time of the `make` request, these optimistic results are used immediately. Otherwise, the necessary computations are reexecuted.

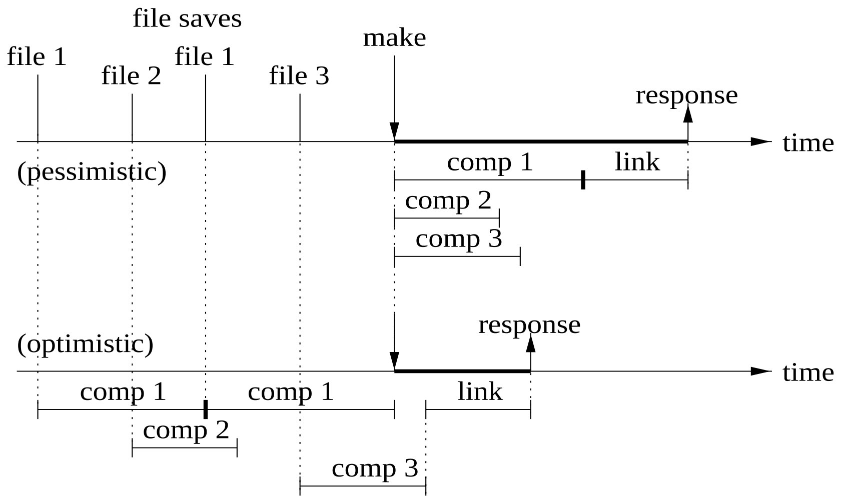

Figure 2.6: Optimistic versus pessimistic distributed `make`

The operational differences between optimistic `make` and pessimistic `make` are shown in Figure 2.6 [19], as well as the potential performance benefits of optimistic `make`. The top portion of the figure depicts the operation of a pessimistic distributed `make`, whereby a number of files is edited and saved, and then a `make` request is issued. The bottom portion of the figure depicts the operation of optimistic distributed `make`. Commands are started as soon as the user saves files, when targets become out-of-date. This improves the response time for the `make` request considerably.

The optimistic `make` program needs to monitor the file system for modifications to the source files on which the `makefile` targets depend. To do file system monitoring efficiently, it makes minor modifications to the kernel and some of the servers. When any file in a specified list of directories is modified, optimistic `make` requests notification from the file server and starts the computations. If there are two or more independent computations to update a target, they can be started concurrently on separate machines.

2.6.4 Fourth Generation Make

The Fourth Generation Make [37], also known as `nmake`, embodies major semantic and syntactic enhancements to the standard `make` program. The enhancements include support for source files distributed among many directories, an efficient shell interface, dynamic dependency generation, dependencies on conditional compilation symbols and a powerful new meta language for constructing default rules. The `nmake` utility also provides improved functionality and performance.

The `nmake` search strategy assumes an extremely modular directory structure. Some directories should be devoted to source files (.c), some to header files (.h), some to libraries (.a), and so on. With this directory structure, the programmer can tell `nmake` where to find files by assigning directories to the targets .SOURCE.c, .SOURCE.h, .SOURCE.a, and so on. There is even a special way to specify a search path for description files.

The update commands are executed by sending the command blocks to the shell *sh*, which runs as a co-process. Command *aliasing* and shell *functions* are preserved in update command blocks. If the user specifies `-jn` as command line option, `nmake` allows concurrent execution of update commands, associating targets with process ids. In this case, each update command block is run by a separate subshell that is started by *sh*, and the dependency graph is used for synchronizing the jobs. The `nmake` utility communicates with the shell via pipes. Commands are sent to the shell on one pipe and status information is received from the shell on a second pipe. While commands are being executed by the shell, `nmake` continues by checking the dependencies of the next target.

File dependencies are dynamically generated from a given set of source files. Scanning the contents of source files for **#include** statements, `nmake` finds dependencies on header files. A file is only scanned if it has been modified since the last time it was scanned. A drawback arises when some **#include** dependencies are from "compiled out" parts of the source files. The `nmake` utility assumes that for a given application the "compiled out" dependencies are rarely modified and use the C preprocessor *cpp* to produce the exact dependencies. In nmake *cpp* is basically used to preprocess each makefile. However, *cpp* scanning may not be appropriate after minor changes to the source files.

2.6.5 Parallel Make

Parallel make [5] (pmake) aims to speed up the operation of make by doing compilations in parallel like mk and optimistic make. The parallelism is hidden from the description file writer. It doesn't burden the programmer with all the details of the parallelism. In pmake the syntactic compatibility with make description files is maintained. Description files of pmake can be interpreted correctly by make.

Parallel make is based on virtual processors. For each command block to be executed, pmake creates a child process called a *virtual processor.* Without waiting for the virtual processor to finish, pmake continues processing the list of dependencies in which the target appeared. The virtual processor controls the execution of the command lines in the command block. The command lines are run one after another, each in a separate shell environment. As soon as one of the commands fails, the virtual processor exits with status *false.* If all commands succeed, then the virtual processor stops with status *true.* This helps pmake in deciding whether to make the parent target.

While the commands in a command block are executed sequentially, the command block itself is executed as a whole in the background. This is achieved by surrounding the command block by { and }&, and using a single shell to run it as a whole. When the parent target needs the command block's result, pmake waits for all virtual processors dealing with the dependencies to finish, before making it.

A description file can contain commands which cannot run in parallel with each other, such as *lex* and *yacc.* Command blocks in which these commands are included need to run mutually exclusively (i.e. one command block at a time). To prevent certain command blocks from executing in parallel, the programmer defines a *mutex* on the group of command blocks, by declaring a rule having target name .MUTEX. This suppresses parallelism and causes pmake to behave like make.

As in mk and nmake, the user can explicitly specify how many virtual processors may be used by defining the `-Pn`*num* command-line option. Otherwise, a default number is chosen, and it is up to the system administrator to select an efficient number, depending on the number of available physical processors. The virtual processor mechanism introduces a minor overhead because an extra process is required for each virtual processor to control the execution of the command block.

Chapter 3

Interactive Undo Support

Undo support is generally a facility which allows a user to restore a system to what it was at an earlier stage. It plays an important role in enhancing the usability and learnability of human-computer interaction systems. Undo support can cover the correction of state changes imposed by both system errors and user errors. Many techniques have been developed which deal with recovery from errors the system itself causes. This kind of recovery is usually under the system administrator's control. Conversely, recovery from user errors is achieved on line as a result of using a few recovery commands provided within most interactive systems.

This chapter performs a concise examination of interactive undo support, which is concerned with user errors in particular. For our purposes, some sections of the chapter are adapted from those in [98]. It starts off by giving a general characteristic of user recovery support and by describing the nature of system-oriented recovery in operating systems and database systems. Then the necessity and practical utilities of user-oriented recovery are introduced. It also presents a characterization of user-oriented recovery, where existing undo support models are discussed in detail, and their advantages and drawbacks are analysed.

3.1 Introduction

Undo support is a capability which is directly concerned with maintaining the integrity of a user's work [63] and should allow easy reversal of actions as long ago in the history as possible [79]. The psychological behaviour of a user may make interactions with the system frustrating. For example a command may give an unexpected response and, therefore, a system should provide recovery from unwanted actions conveniently and easily [36]. However, frustration can be caused by irreversible actions such as the accidental deletion of important data. As it is difficult to cope with irreversible actions, interface actions should be made as reversible as possible [68]. Also, the provision of undo support may help users in minimising the time spent in correcting errors. To avoid such a waste of time, error-recovery methods should be designed in terms of learnability and efficiency [22].

Many programming environments, and application programs such as editors and word processors, support a *single undo* facility. The most recent command can be undone and redone, but commands previous to that cannot be reached. Examples are the Smalltalk system [39] and the Vi editor [16].

More sophisticated environments and applications support a *multiple undo* facility in which the most recent commands can be undone consecutively, back to some previous commit point such as the beginning of the current session. A general model for this has been presented by Archer et. al. [4] and examples include Microsoft products, Cope [3], Interlisp [83], Pecan [75] and Sam [71].

Most systems with multiple undo also support *redo* facilities. The most important property of a redo facility is recoverability, as described by Gordon et. al. [41]. It should be possible to revert to any previous state that the system has been in, including all states which were abandoned by undoing some commands and then issuing alternative commands. A number of approaches to the provision of completely recoverable undo facilities have been investigated by providing various undo, redo, skip and rotate commands and defining how these act on each other, for example by Vitter [93] and Yang [97]. The Emacs editor [82] achieves a completely recoverable undo facility using only a single special undo command. Consecutive undo commands provide a multiple undo facility, recovering from any number of previous commands. However, any command other than an undo command breaks the sequence of undo commands and, at this point, any previous undo commands are treated as ordinary changes that can themselves be undone.

Beyond this, it is possible to provide *selective undo and redo* facilities in which an arbitrary previous command can be chosen and its effect undone or redone. General models are described by Berlage [8, 9] and by Prakash & Knister [72], who include issues to do with multi-user interaction, and an example system is GINA [81]. Desirable features of selective systems are that the current state should correspond to the sequence of non-undone commands which have preceded it, and that a command which is redone should be executed in a state which agrees with the state in which it was originally executed. Unfortunately, conflicts can arise in which some commands cannot be undone or redone without violating these desirable features. We will discuss these conflicts in the later chapters.

3.2 User Recovery Support

Interactive recovery support has been investigated in two separate modes; system mode and user mode. This section presents the characteristics of recovery carried out in these modes, discussing examples of the situations which recovery is needed for.

3.2.1 Recovery characteristics

Recovery refers to the reinstatement of some past state of the computer environment, usually after some kind of error or malfunction. Four types of recovery situations have been identified [98]:

(1) abnormal exit from a program or code emulator
(2) system crash
(3) user correction of data input
(4) user cancellation or annulment of prior commands

The first two situations originate with an action of the system. They require recovery by the system. The latter two recovery situations require the user to detect the undesirable situation and initiate corrective action. They require recovery by the user. Although these labels are not mutually exclusive with respect to any particular error, they can be used as a convenient way of highlighting some of the differences among their application environments, motivations and requirements.

In general, system recovery has been studied within the context of programming languages, development environments, fault-tolerant systems and databases. The motivation of system recovery is to provide reliability in using system resources. The system can itself or with little operator's help recover to a valid state or restore to the current state after a system failure. Since system failures are infrequent events, the corresponding recovery facilities can be expensive both in time and in space and need not be especially easy to use.

User recovery has been examined within interactive environments such as text and graphics editors. The motivation of user recovery is to provide usability and efficient resource utilization. The recovery emphasis of interactive computer environments is on reversal from user's mistakes or changes of mind. The user recovery allows the user to reverse selectively the effect of previous actions and to put the system into a more desirable previous state. This is considered to be a frequent event, so user recovery should be convenient and relatively inexpensive to perform.

3.2.2 Recovery situations

Recovery situations of type (1), like abnormal termination from program execution, from compilations, or from any program controlling a user action, typically occur because of special cases or exceptional situations. Often, in such cases, the user is given no more than an abrupt "JOB ABORTED" or similar message. This situation can be improved by using exception handlers [27, 56, 57]. One commonly used approach is to supply subroutine calls with extra parameters for dealing with exceptions.

Recovery situations of type (2) occur when the system crashes or experiences an unscheduled and abrupt termination of operation. Handling this problem smoothly is crucial to fault-tolerant systems and databases. The basic concepts used within fault-tolerant systems are atomic action, recovery line, commitment and compensation, and the techniques encompass recovery blocks, recovery caches, process conversations, program redundancy, forward error recovery and backward error recovery [53]. Within database systems, the recovery methods are mainly recovery to a correct state, recovery to a consistent state and crash resistance. The techniques used for recovery encompass restarting, program salvation, incremental dumping, and retaining previous versions [15, 44].

Recovery situations of types (3) and (4) can be supported by three groups of support features that help a user recover from errors and unwanted situations. Features in the first group, called `escape`s, are used when the user is in the middle of specifying a command. They allow the user gracefully to back out of the command. Features in the second group, called `stop`s, are used to terminate or suspend a command while it is executing. Features in the third group, called `undo`s, are used to reverse the effect of a command that has been executed [13].

Most interfaces provide for local editing before transmission. This enables escape from a command to be achieved simply by not transmitting what has been typed so far. However, it is clearly desirable that the user should also have available some reverse key or command to provide for ending a command following transmission. Most interactive systems have the second groups of support features for error handling and recovery such as the `break` and `quit` signals in the Unix operating system, because they can also be provided easily. The third group of support features include `undo` and `redo` commands (or one command for undoing and redoing), which are the hardest to provide. Undo commands allow the effects of task-oriented commands to be reversed. Redo commands allow undone commands to be re-executed.

3.2.3 Recovery in operating systems

Recovery techniques that are adopted in operating systems are more or less variations of ones used in database systems. Just as there are many kinds of failures, there are several recovery techniques that have been defined over the past years [92]. However, the kind of failure always imposes a limit on these methods and a recovery mechanism thus provides recovery only from certain kinds of failures.

On a multiuser system, the system administrator makes persistent backups of the file systems onto dump tapes so that, in the event of a system failure such as a disk crash or accidental deletion of files, the most recently copied version of the data or program can be reinstated from the dump. But, for a large system, it is not practicable to dump all the stored information on every occasion because it renders backing up awkward and time-consuming. Such an inconvenience can be avoided by using methods that reduce the total amount of information to be copied during the dump.

In Unix, methods such as incremental dumping and full dumping present the user with a way of protecting any original file against a possibility of loss. Incremental dumping is used to make a regular dump of only those files that have been modified since they were last dumped. This method is implemented by keeping a list of the dump times for each file on disk. The dump program checks each of the files on the disk, comparing their dump times with their last update times. Files that have changed since the last dump are dumped again and their dump times are replaced with the current time. This allows dumps to be made more frequently.

Even though the backup strategy can be considered as a recovery facility, the storage where the backups are kept is not normally available to users. That is, existing methods don't allow users to take any action through their own command line to repossess the desired information by using the system resources. So the user should direct an appropriate request for the provision of the relevant information to the system administrator on the related machine. In some cases incremental dumping may not make bringing all the files to their previous consistent states possible. For example, not all updates performed by jobs running at the time of the crash may be reached by the processing of the incremental dump tape because some active files may not have been dumped in time.

3.2.4 Recovery in database systems

Recovery in a database system means, primarily, recovering the database itself; that is, restoring the database to a state that is known (or assumed) to be correct after some failure has rendered the current state incorrect, or at least suspect [29]. The underlying principles on which such recovery is based is related to `redundancy`. The way to make the database recoverable is to make sure that any piece of information it contains can be reconstructed from some other information stored, redundantly, somewhere else in the system.

Database recovery is basically transaction-oriented. A transaction is a short sequence of interactions with the database, using operators such as `FIND` a record or `MODIFY` an item, which represents one meaningful activity in the user's environment. A typical example for a transaction is the transfer of money from one account to another (see Figure 3.1). Users interact with a database system by invoking programs that create desired transactions.

```
begin_transaction;
EXTRACT (account1, amount);          /* do debit */
if any error occurred {
    rollback_transaction;            /* undo all work */
    return;
}
INSERT (account2, amount);           /* do credit */
if any error occurred {
    rollback_transaction;            /* undo all work */
    return;
}
commit_transaction;
```

Figure 3.1: A sample transaction (pseudocode)

The concept of a transaction, which includes all database interactions between `begin_transaction` and `commit_transaction` in the above example, requires that all of its actions be executed indivisibly: Either all actions are properly reflected in the database or nothing has happened. No changes are reflected in the database if the transaction fails at any point in time before reaching the `commit_transaction`.

A transaction may be defined as having the following properties:

- Atomicity: Either all or none of the transaction's operations are performed.

- Consistency: A transaction transforms the system from one consistent state to another.

- Isolation: An incomplete transaction cannot reveal its result to other transactions before it is committed.
- Durability: Once a transaction is committed the system must guarantee that the result of its operations will persist, even if there are subsequent system failures.

These four properties, atomicity, consistency, isolation, and durability (ACID), describe the major highlights of the transaction paradigm, which has influenced many aspects of development in database systems.

In summary, a transaction can terminate in three ways. It is hoped that the transaction will reach its commit point, yielding the all case (as in all-or-nothing dichotomy). Sometimes the transaction detects bad input or other violations of consistency, preventing a normal termination, in which case it will reset all that it has done (abort). Finally, a transaction may run into a problem that can only be detected by the system, in which case its effects are aborted by the database management system [44].

3.3 Utilization of user-oriented undo

Undo support plays a key role in helping users control their interactions with interface systems. Many significant situations need dealing with by way of a user-oriented undo command. This section discusses the importance of undo support and gives a number of situations where undo support is provided.

3.3.1 Why is undo support important?

There are many reasons why undo support is an important research topic. Firstly, human beings are fallible. Making errors is normal for human beings. While it is usual to focus on the result rather than on the errors made along the way, making errors often makes valuable contributions to finding the path to the user's goals. Much research has been conducted to understand the causes of errors [77, 91]. Generally speaking, there are two broad causes of errors committed at computer interfaces: human factors' inadequacies of computer systems and human psychological limitations. Human factors' inadequacies of systems include mismatches between the functionality of the system and the task domain, ambiguous feedback information and so on. Human psychological limitations include users' lack of knowledge about the functioning of the system, their lack of knowledge of the result of previous actions within a particular task, users' learning limits and so on.

An undo command reduces the consequences of some errors caused by human fallibility and inadequate system design. Functionality and operation of software of moderate or greater complexity are not completely transparent to a user. When using such a system, the user is bound to make some syntactic and semantic mistakes using it, no matter whether the user is experienced at using it or not. A command which has been executed may not be what the user intended. In this case, undo support allows a user to use an undo command to reverse the effects of the command.

3.3.2 Repairing mode errors

An obvious example where undo support is desirable is a mode error. A user commits a mode error when the user classifies a situation wrongly, acts correctly in the light of the user's classification but falsely for the actual situation [98]. In other words the user assumes the machine is in one state when it is in fact in another. For example, some editors have a text mode for entering text and a command mode for issuing comands. Failure to identify which mode the system is in leads to errors of attempting to insert new text while the system is in command mode or specifying commands while it is in text mode. In the `Vi` editor, typing the word "muddy" while the system is actually in command mode leads to deletion of one line of the manuscript. Sometimes mode errors can have more serious effects, such as destruction of the entire manuscript.

Modes are generally undesirable in human interfaces because they require the user to remember what state the system is in. They also complicate the user's conceptual model of the keyboard because keys have different functions in different situations. Some suggestions have been made to remove or minimize mode errors [64, 68, 86]. One way is to increase the `width` of the user interface by introducing new keys or additional input devices as alternatives to the keyboard. Another way is to rationalize the functionality of the system so that the commands required by different modes are different. However, these suggestions cannot always be followed, and if they are, they can sometimes lead to other classes of difficulties [68]. Where interfaces use modes, undo support allows a user to correct mistakes due to mistaken mode assumptions.

3.3.3 Handling deadlocks

Another situation where command reversal may be needed is to prevent interactive deadlocks. Interactive dialogue systems produce transitions from old states to new states [28]. Enabling conditions for such a transition `t` from old state `s` to new state `s'` in a dialogue system therefore include

1) the existence of a command `c` corresponding to `t` such that: `c` is applicable in `s`, that is, the system will accept `c` in `s`;
2) `c` is knowable in `s`, that is, the user knows or is able to learn `c` in `s`;
3) `c` is processible in `s`, that is, the system has the means available for processing `c` in `s` or can obtain such means reasonably soon.

The term interactive deadlock is introduced to denote deadlocks that arise through the failure of commands to fulfil either or both of the first two conditions, that is, where

(a) no command is applicable in the current state `s` that enables a transition `t`

(b) a command `c` exists but the user does not know `c` and cannot learn `c` in `s`.

Interactive deadlocks of type (a) are most often found in completely developed or experimental systems whose designers have neglected to provide commands for all of the elementary tasks that are essential to the performance of the main tasks for which the system is designed. Interactive deadlocks of type (b) occur in poorly documented systems and in systems with poor or no online help facilities. Faced with such a problem, users often notice that they can get around an apparent deadlock by going back to some old state and issuing a different command sequence. In such cases, undo support can provide a versatile way of coping with interactive deadlocks.

3.3.4 Changing one's mind

Undo support can also be used to deal with a situation where users change their minds as to what commands they want to have issued. A user may have issued several commands which have been executed, but may revise these in the light of the outcome and want to change the effect of some of those commands. For example, in a design process in solid modeling, a user may have performed the following actions:

1) first use several local commands to design a solid A;

2) next use several local commands to design a solid B;

3) then use a set command to combine the two solids A and B;

At this point, after evaluating the current state, the user may want to re-shape the solid A. With the help of undo support, the user can achieve this easily. The user first undoes step (3), step (2), and step (1) and redesigns the solid A. Then the user redoes step (2) and step (3) by either reuse commands or redo commands depending on the functions provided by the system [98].

All of human behaviour can be characterized in terms of goals and plans. In particular, most of what people do can be analysed as performing plans of action that have been devised in order to accomplish a goal. With some goal in mind a user is required to translate the user's needs into a form which matches the methods and operations by which a computing system can achieve it. When interacting with a computer, a user develops a plan in the expectation that its execution will achieve the given goal. The user has to partition the goal into subgoals and then subgoals into lesser subgoals and so on.

The process of executing a plan on an interactive computer system can include many stages of user activity [69]. Errors can arise at any stage. By revising and updating a plan of action, the user may be able to go back several steps and execute another plan in order to achieve the goal. At this point, undo support can provide a short cut for the user to go back several steps in order to achieve a revised plan.

3.3.5 Exploring systems

Another capability provided by undo support is to enable a user freely to explore new or unfamiliar functions of the system without the fear of committing catastrophic mistakes. Exploration is a learning mode whereby people educate themselves by interacting with a situation of interest as opposed to reading about how to handle it, being told, or looking at examples [98]. Exploration is common in everyday life. It allows a person to deal with various situations that the person has come into contact with for the first time or that the person has forgotten how to deal with partly or totally.

When using an interactive system, a user often wants to experiment with the system facilities so that the user can learn how the system works by trial and error. One proposed approach is to replace teaching by exploration [60]. If learners' first guesses are not correct, learners may still be able to figure out what to do. To do so, they have to be able to tell what happened when they tried out their guesses and they have to be able to recover from the effects so that they can try something else. This strategy is commonly known as learning by doing.

One feature which is helpful for exploratory behaviour is making it easier to correct errors. Novices spend as much as 50% of their time in trying to leave a situation that they erroneously stumble into [23], e.g. getting into a mode that does not allow them to continue writing text. It is an important aid in mastering a system that a user is able to undo mistakes. Furthermore, as more and more knowledge is built into an interactive system and more and more automatic actions are performed by the system because of available techniques, even an expert user may feel confused and would like to explore alternatives.

3.3.6 Using undo as a generic command

The most important characteristic of undo support is that undo commands are generic commands. A generic command is a command which is recognized in all contexts of a computer system. It is generic to all systems. Generic commands can be viewed as extremely general actions which make minimal assumptions about their objects. The particular interpretation of such commands depends on the nature of the objects to which they are applied. No matter which object an undo command is applied to, it always reverses the effects of some previous command. Many designers have discussed the advantages of using generic commands [58, 80, 85, 96]. Generic commands

- reduce the number of commands and command names a user has to know.
- increase the consistency and predictability of a system by using a similar set of actions across different contexts.
- allow users to map their everyday actions more easily into the computer domain.

3.4 Undo Support Literature and Facilities

There has been much work done on the analysis and provision of undo support. This section reviews current literature concerning undo support. Firstly the importance of undo support in the literature is examined. Then current undo support facilities are described within their functional context.

3.4.1 Recommendations of undo support

The issues concerning undo support have drawn the attention of many authors for some time. They have examined the relation between undo support and the interface of a computer system and also focused on the reasons why an undo support facility is important. A concise survey of the undo support literature is introduced below.

Foley and Wallace [36] reasoned that the psychological behaviour of a user may bring an interaction into frustration. A typical example of frustration is a situation where users are devoid of an ability to convey their intentions to the computer easily. This results in either an unexpected response or none at all. If a user can't undo the unexpected result or determine what response actually occured, the situation becomes further complicated. Consequently, they concluded that undo support is a more general system requirement and a system should provide recovery from incorrect or unwanted actions conveniently and easily.

Meyrowitz and Van Dam [63], in the examination of interactive editing systems, pointed out that undo support is a capability which is not directly involved with manipulating text, but rather with maintaining integrity of a user's work. They argued that, although being critically important and time-saving, undo support is not yet a universal feature of interfaces. It frees the user from the burden of making sure that each command does exactly what is expected of it by guaranteeing that any result can be undone. Their point was that a general undo command facilitates risk-free experimentation and makes the user interface more powerful and helpful.

Rutkowski [78] advocated that a system should provide a way to allow a user to change any decision even after the system has acted on it. Literally, an undo key is involved in reversing the user's decision. He investigated possible uses of an undo command and emphasized that it is as important as reversing an action that an undo action itself is able to be reversed. He also urged that an undo capability must have both its scope and granularity clearly defined.

Card et al [22] discussed that the provision of undo support is a necessity to minimize the time spent in correcting errors because even expert users may spend up to about 30% of their time in erroneous situations. In order to avoid such a waste of time, error-recovery methods should be designed in terms of learnability and efficiency. Considering that there are so many kinds of errors whose effects need reversing in different contexts, a designer must specify carefully how it will be used for each error.

Norman [68] focused his examination on human errors. After observing that it would be difficult to cope with irreversible actions, he argued that interface actions should be made as reversible as possible. Users wish to be able to recover gracefully from their failures and their fallibility compels all interface systems to essentially consider the undo command. Maintaining that human error is fundamental to human operations, he suggested that designers should use the analysis of errors to help design systems.

Rissland [76] discussed certain general features of an intelligent user interface and characteristics desirable in an interface. He argued that among the desirable ones for intelligent interfaces is being helpful and forgiving (e.g. "infinite" undo/redo capabilities) and minimizing errors.

Shneiderman [79] pointed out that one of the fundamental principles of interface design is to allow easy reversal of actions at as far back a point in the history as possible. This helps a user feel free from the anxiety of making errors and exploring unfamiliar options of interfaces. He also believed that it would be useful that recovery consists of simple operations such as a single action and a data entry.

From this literature, one can conclude that undo support makes an interactive system more usable and learnable. It presents a convenient way to support the fallible nature of human beings and avoids creating frustration where a user discovers that an unexpected result of a command cannot be reversed. It encourages the user's exploration of unfamiliar options of the interactive system by permitting ready recovery from many simple errors. However, such broad principles involve making more specific interpretations so that they can be readily exploited in the engineering of usable interfaces. This means formulating characteristics of undo support and economical techniques to implement it effectively.

3.4.2 Program development environments

Undo support has been designed as a crucial auxiliary component in various program development environments. Their undo support provision varies considerably. The most important examples of these environments are Cope, Interlisp, Pecan and Smalltalk.

Programming environments interpret the granularity and range of undo support differently [98]. The granularity of an undo command expresses how many commands this undo command can reverse at once. It can cover one command or multiple commands, i.e. an undo command can reverse the effects of either one command or many commands at a time. However, the range of undo support indicates how far a system state can be reversed. The range of undo support can apply across the whole current interaction session, across the most recent part of the current interactive session, or can apply only to the most recent command. Undo support with one-step range only has the ability to reverse the system state one step back.

Cope [3], an integrated program development environment, provides an undo command with non-restrictive one-command granularity. The undo command cancels the effects of the last effective command every time it is invoked. By submitting the undo command consecutively, the system can be reversed back to any state in the range of a fixed whole system. In Cope, the effects of an undo command can be destroyed only by a redo command. The undo and redo commands don't operate on themselves.

Interlisp [84] is a programming development environment for the Lisp language which provides the most powerful undo support facility. Undo support is implemented in its two utilities, the Editor and History Package. The Editor maintains an internal list to which each change made by the structure modification commands is recorded. The undo command it supports has only a one-step range and so undoes the most recent one of the structure modification commands. As an undo command can operate on itself its effects can be reversed using another undo command. The History Package, a set of past commands and functions, provides an undo command with multiple command granularity. Undoing in the reverse order of execution will restore the system to its correct previous state through the current state. Otherwise, undoing in any other order may cause the state to be unpredictable.

Pecan [75] is a family of program development systems with undo and redo facilities. It allows changes to data structures to be undone and redone. When attempting to undo a particular command, a user points with the locator device at that command so that the system can undo or redo the chosen point. The undo support provided by Pecan has a global undo/redo facility for the user interface.

Smalltalk [39] provides an integrated collection of tools for interacting with components associated with an object-oriented programming language. It has the least powerful undo support. An undo command that will reverse all the users' typing and backspacing operations is presented in a pop-up menu. The submission of two consecutive undo commands leaves the state of the system unchanged.

GINA [81] is an object-oriented application framework which was designed to facilitate the construction of interactive graphical user interfaces. As part of the application, a history mechanism is developed which is based on storing a command object for every user interaction [12]. It allows selectively undoing/redoing of commands, handling potential conflicts between them. By using meta recovery commands the framework provides, each state in the tree of command objects can be reached.

Amulet [66] is a user interface development environment which allows programmers to create highly-interactive, graphical user interface software for operating systems such as Unix and Windows. All built-in operations in Amulet are automatically undoable. It also supports the writing of a custom undo method if the programmers create custom commands that perform application-specific actions, like deleting the wires when the gates are deleted. There are three different undo mechanisms that the developer can choose from: single undo like the Macintosh, multiple undo like Microsoft Word Version 6 and Emacs, and a novel form of undo and repeat, where any previous command, including scrolling and selections, can be selectively undone, repeated on the same object, or repeated on a new selection.

3.4.3 Editor environments

Many text/graphics editors also present undo support in a few variations. Representative examples of such editors are Emacs, Sam, Lisa, and Vi.

Emacs [82] , a real-time display editor, has a powerful undo command. All changes made in the text of a buffer can be undone, up to a certain number of changes. The first time the user issues an undo command, it undoes the last change. Consecutive repetitions of the undo command, in a linear way, change the text back to the limit of what has been recorded. Any command other than an undo command breaks the sequence of undo commands. Starting at this moment, the previous undo commands are considered ordinary changes that can themselves be undone. This is equivalent to the redoing of the original commands.

Sam [71] is an interactive multi-file text editor for bitmap displays. It has an undo command which allows a user to undo the last n changes. Without taking notice of how many files are affected, it merely demands undoing all files whose latest change has the same sequence number as the current file. A sequence of undo commands moves the editing state further back in time.

Lisa [65] is a word processor developed by Apple Computer Inc. The "Undo last change" command is available in every program which allows a user to undo the effects of the last command issued.

Vi [16] is a screen-oriented editor of the Unix system. It includes two separate undo commands to undo the last change and all the changes made to the current line. The successive repetitions of these commands either consist of destroying each other's effects or have no effect. In addition, Vi presents a user with a simple redo facility that only authorizes the repetition of the last editor command.

Many generations of Microsoft Word, as in other Microsoft products such as Excel and Powerpoint, present a multi-step undo mechanism. With this mechanism, users can reach all previous versions of a document in one file, without having to save multiple copies of the document to keep track of different versions. There is also a facility to undo a set of most recent commands at a time.

3.4.4 Other uses of undo support

Undo support is also of great use in some other interactive environments. Representative examples of these interactive environments are Designbase, CSS, and US&R.

Designbase [90] is a solid modelling system all of whose high-level operations are implemented using primitive operations. All primitive operations used in a design are stored in a tree representing the history of the solid design process. A user simply specifies a path moving up the tree to undo a sequence of operations.

CSS [17] is a computer Cricket Scoring system designed to control an electronic cricket scoreboard. The main feature of the system is a recovery facility that allows the operator to undo events that have been performed erroneously. The system stores events minimally, so that they can be redone later. The user may backtrack over events in order to return to a point in the past at which a deletion or insertion of an event can take place.

US&R [93] is a component of an interactive graphics layout system implemented on a workstation. It provides `undo`, `redo` and `skip` commands. The `undo` command reverses the last action in the current state and it can be used consecutively. The `skip` command causes previously undone commands to be skipped. The `redo` command re-executes an undone command.

3.5 Characteristics of Undo Support

This section characterises relevant concepts for undo support and describes basic properties that an undo command may possess. These descriptions enable undo support models to be classified in terms of their main characterising properties. Formal models are used, because they are amenable to precise analysis and actual systems can be verified against them. The models, therefore, are applicable to a large class of real systems.

3.5.1 Basic concepts

In existing interactive systems with undo support facility, user-issued commands are divided into two separate classes in terms of handling, namely task-oriented commands and recovery commands. Consequently, the domain of undo support is only a subset of all these commands. A task-oriented command is a user-selected command which changes the state of a system to a new state. A recovery command is another user-selected command which changes the state of a system to a previously existing state. The situation of recovery can be defined as an interactive cycle during which the user selects a recovery command and the system executes it.

Task-oriented commands are maintained in a *history* list in which they are stored in the same order in which they are executed on the system. Some systems only keep the commands that modify their states. The *state* of a system is defined as the ordered sequence of task-oriented commands still in effect. For example, if the task-oriented commands C_1, C_2, C_3, and C_4 are executed, the state of the system is

$$C_1 \;\; C_2 \;\; C_3 \;\; C_4$$

This means that the system performs the commands C_1, C_2, C_3, and C_4 in sequence, bringing its state from the one before C_1 to the one after C_4. For a command C that is reversible, C^{-1} represents an inverse command that will reverse the effect of C. For instance, in an editor, the effect of an `INSERT` operation can be inverted by a `DELETE` operation. If C_4 is then undone, the state reverts to

$$C_1 \;\; C_2 \;\; C_3$$

In general, the inverse of a task-oriented command C can depend on the state of the system prior to C. One way to invert C is to execute the inverse C^{-1}; either C must be placed into the history list with sufficient information or this information must be stored somewhere else in the system so that C^{-1} can be uniquely specified. Another method for inverting C is to back up the system to some previous checkpoint and reexecute commands up to but not including C. A thorough treatment of these issues appears in Leeman [57].

3.5.2 General characteristics

Gordon et al [41] discuss the desired characteristics of different undo commands and analyse their inter-relationships, defining basic properties of undo support. Some of these properties are contradictory to each other. They show that a single undo command cannot have all of these properties. Some important concepts used in their analysis are further examined by Yang [97].

The following two properties, *reversibility* and *inversibility*, are proposed by Yang as sufficient properties for a facility to be an undo support facility. `C` is used to represent the set of task-oriented commands which belong to the undo domain. `U` is used to represent the set of undo commands. `S` is used to represent the set of states of the working object, and s_0 is used to represent an original state or a state which is equivalent to an original state. The form $f_n(f_{n-1}(...(f_1(s))...))$ is abbreviated as $f_n...f_1(s)$. The above properties are defined as follows:

1) **reversibility:** c_1 ... c_n on $\mathtt{s} \in \mathtt{S}$ is reversible if $\exists u_1$ ... u_m such that

$$u_m \ldots u_1 c_n \ldots c_1(\mathtt{s}) = \mathtt{s}$$

The property of reversibility states that the state of a working object can be reversed to a previously existing state after a sequence of task-oriented commands have operated on the working object. If an undo support facility has the property of reversibility, the effect of any task-oriented command within the domain of undo support can be reversed. If $\mathtt{m}$ can only be 1, the reversibility is single step; otherwise it is multiple step. Single step reversibility is the basic property of undo support which guarantees that undoing is possible. Special cases where $\mathtt{m}$ and $\mathtt{n}$ can have various values are discussed by Yang [98].

2) **inversibility:** A sequence of undo commands v_1 ... v_n on $\mathtt{s} \in \mathtt{S}$
is inversible if $\exists u_1$ ... u_m such that
u_m ... $u_1 v_n$ ... $v_1(\mathtt{s}) = \mathtt{s}$

The property of inversibility states that for any undo command, its effect can be reversed by one or more undo commands. If an undo support facility has the property of inversibility the effect of any undo command can be reversed.

Although reversibility and inversibility both concern the reversal of the system to some previously existing state, their orientations are different. During task processing, reversibility operates in a backward manner and inversibility operates in a forward manner. In other words, the function of reversibility concerns the reversal of task-oriented commands and makes the system go backwards. The function of inversibility concerns the inversion of undo commands and makes the system go forwards.

Since undo support may have more than one undo command and different undo commands have different commands, another two properties, *self-applicability* and *unstacking*, are proposed by Yang as propertie to classify a particular undo command.

3) **self-applicability:** $\mathtt{u}$ is self-applicable if $\forall \mathtt{s} \neq s_0$, $\mathtt{uu}(\mathtt{s}) = \mathtt{s}$

If the undo command $\mathtt{u}$ has the property of self-applicability, $\mathtt{u}$ can reverse its own effect.

4) **unstacking:** $\mathtt{u}$ is unstacking if $\forall\ \mathtt{s} \in \mathtt{S}$, $\forall c_1$ ... c_n,
$\mathtt{u}$ ... $\mathtt{u}c_n$ ... $c_1(\mathtt{s}) = \mathtt{s}$ (n copies of $\mathtt{u}$)

If the undo command u has the property of unstacking, u is not self-applicable.

These two properties are for a particular undo command. They are disjoint, that is, an undo command which has the property of unstacking cannot be self-applicable and vice versa. Because of these disjoint properties, all the undo support models can be classified into two classic categories [97], *primitive undo model* and *meta undo model*. The undo command in a primitive undo model is self-applicable. The undo command in the meta undo model has the property of unstacking.

All the above properties describe the functional characteristics of undo support. The following property, *thoroughness*, is for basic performance. This property is proposed by Yang as a basic performance requirement. In order to describe this property realistically, a concept *committed* state is introduced as a state where the user is prohibited subsequently from undoing any command which has been issued before this state.

5) **thoroughness:** $\forall$ u $\in$ U, $\forall$ c $\in$ C if state uc(s_k) is a committed state,
$\forall s_k \in \mathtt{S}, \forall c_1 \ldots c_n, c_n \ldots c_1\mathtt{uc}(s_k) = c_n \ldots c_1(s_k)$

The property of thoroughness says that the situation where the effects of a command are reversed is equivalent to the situation where that command was not executed. The property guarantees that undo support provides an enhancement of facilities for the usability and learnability of the system. It does not change any overt functionality of the system. It only concerns the effects of commands on the system, not the way to get the effects. This leaves the designer free to choose implementation strategies.

3.5.3 Primitive undo model

In the primitive undo model, undo support is presented with minimal facilities. There is no difference between task-oriented commands and recovery commands that are defined as any user-issued commands. As far as recovery is concerned, they are all considered as ordinary commands. Thus all commands which have been issued by the user reside in the history list. The undo command, which is the only one recovery command in the primitive undo model, can operate on any command in the history list, whether it is a task-oriented or recovery command.

The redoing of a command is based upon the use of the undo command on that command twice.

The primitive undo model has the properties of self-applicability and reversibility. However, capabilities of a primitive undo command can be divided into two categories in terms of reversibility, *primitive undo* and *multiple undo*.

(1) **Single undo:** This undo model supports a simple recovery facility (single reversibility). A single undo command reverses the effects of the last user-issued command. If the last user-issued command is a task-oriented command, the undo command reverses its effects. If the last user-issued command is an undo command, the undo command restores the effects of the command which has been undone by that undo command.

To show how single undo works, suppose a user executes four task-oriented commands A, B, C, D, putting the system into the state

$$A\ \ B\ \ C\ \ D$$

These commands appear in the history list. The user can only undo D, but not the other commands. After executing an undo command (U), the system turns back to the previous state

$$A\ \ B\ \ C$$

and the history list appears as follows:

$$A\ \ B\ \ C\ \ D\ \ U$$

Then if the user issues a second undo command, it will cause the command D to be placed to the history list again.

The `Vi` editor represents a typical implementation of single undo. It is the simplest undo. The user has to realize the potential requirement of recovery from a command as soon as it is executed. Execution of another command drives the command out of the domain of undo.

(2) **Multiple undo:** The multiple undo model provides a more powerful recovery facility than the single undo model. It allows the undo command to take a count n that specifies the number of previous commands to undo. With this count, the user can return the state of the system to any former value

(multiple reversibility). When the number (m) of undo commands executed consecutively is greater than the count (n), then (m-n) undo commands have the effects of redoing on the state.

For example, suppose a user issues four task-oriented commands A, B, C, D, and the count n is defined as 3. Using three successive undo commands, if the user restores the state before B was executed, the history list looks like

$$A \ B \ C \ D \ U3$$

where the number that is tagged to U shows how many times the undo command is used consecutively. At this stage, the next two undo commands bring B and C back to the state, and the history list appears like

$$A \ B \ C \ D \ U1$$

Now consider that the user issues another command (E). Unfortunately, at this point, the only way the user can reach the desired final state

$$A \ B \ C \ E \ D$$

is to reenter manually the one command (D) undone earlier. The undo command is of no use.

Some principles of recovery in the Interlisp system and Emacs editor are based on the multiple undo model. The Interlisp `undo` is a command that is itself appended to the history list. Not only can the Interlisp `undo` reverse the execution of the previous command, but a user can specify that any subsequence of the history list be undone. Since `undo`s themselves can be present in the history list, a sequence to be undone can include one or more `undo`s. This can be extremely confusing and make it difficult to anticipate or even understand the system state that will be produced. The Interlisp `redo` is completely independent of `undo`. It is effectively just a "repeat" or "copy" command which appends to the history list a copy of commands that are already present at an earlier point.

As a result, the history list in the primitive undo model doesn't give the user a clear view of the transformation of the system states because of the uniform treatment of commands. Recovery commands don't guarantee the proper manipulation of commands in the history list. This causes inconvenience and confusion.

3.5.4 Meta undo model

The meta undo model has more convenient recovery facilities than the primitive undo model. Recovery commands are differentiated from task-oriented commands. The model regards a recovery command as a meta command which cannot operate on itself. It provides two separate command lists, *history list* and *undone list*. The history list is the list of task-oriented commands which are still in effect. The undone list is the list of task-oriented commands which have been undone. Unlike the primitive undo model, there are two recovery commands: `undo` and `redo`. The `undo` command reverses the effects of the previous task-oriented command. The `redo` command restores the effects of the task-oriented command which has been undone.

The meta undo model has the properties of unstacking and multiple reversibility. Given the user's selection of commands to undo, two different types of undo can be supported by the model, *linear undo* and *selective undo*.

(3) **Linear undo:** The linear undo model allows the undoing and redoing of a sequence of task-oriented commands. Any previous state of the system can be recovered by following a linear path from the current state backwards. Using successive recovery commands, the user can reach any command in the history list. One undo command is only applicable to the most recently executed or redone command and one redo command is only applicable to the most recent undone command.

For instance, if the initial state of the system consists of the commands A, B, C, D, two undo commands make both the history list and the state appear as follows:

$$A \;\; B$$

The commands C and D are included in the undone list. In this case, the effects of the undo commands can be reversed by two successive redo commands to restore the initial state. But, if a new command, E, is executed, which puts the system into the state

$$A \;\; B \;\; E$$

then there is no way to restore the inital state using recovery commands. This is because using one undo command makes the redo command applicable for E, but not C.

A typical example of the linear undo model is implemented in Pecan. A significant limitation of the model is that the user may want to redo a command other than the most recently done one. The only way to do this is to reissue the command manually. The Cope system presents a developed version of the model. However, the above example could easily be complicated by introducing a nesting of recoveries, in which case recovery with Cope or Pecan would be very difficult.

(4) **Selective undo:** The selective undo model provides recovery facilities in the most favourable way. All commands in the history list are directly in the domain of recovery commands. Recovery from an arbitrary command can be achieved by only one recovery command without having to handle any intermediate command. In fact, selective undo is intended to concentrate on reversing the effects of past commands rather than restoring past states of the system. Its basic function is just to remove a command from the current state.

As an example, in the state consisting of the commands A, B, C, D, suppose B is an isolated command[1]. An undo command used for B turns the system to the state

$$A \;\; C \;\; D$$

causing the command B to be added into the undone list. A redo command can bring the command back to the state. It doesn't matter wherever B is placed after A in the history list (its prior position or any other position), because the subsequent commands do not depend on it.

When the undo command is issued for a command which is not isolated from the current state, the recovery situation gets complicated. That's why only a few interactive systems support selective undo. Systems that employ the selective undo model themselves have to deal with conflicts imposed by unisolated commands. Some variations of this model will be examined in the next section.

[1]We use the word "isolated" to refer to commands whose removal from or insertion to the state doesn't affect any subsequent commands which are currently in effect.

The history list of the meta undo model explicitly gives a clear view of the transformation of system states. It is quite easy to analyse the current state. Although the philosophy behind the model is more advantageous, it can force systems to cope with many different recovery situations and so create a greater burden of recovery.

3.6 Current Interactive Undo Support Models

The primitive and meta undo models discussed above can be treated as a superset of many other models which are used to satisfy diverse undo support requirements of interactive applications. Undo/redo facilities provided by these models range between linear and nonlinear recovery. This section describes currently existing undo support models and analyses their recovery capabilities.

3.6.1 Script model

Archer et al [4] present an application-independent script model (ACS) for interactive systems that allows undo support to be defined precisely. In the script model, the user is viewed as constructing a script, and the system is viewed as providing a continuous interpretation of the script. The interactive cycle has two sorts of activity:

(1) `Edit:` The user is editing the script.

(2) `Execute:` The system interprets the script.

The cycle is repeated until the user is satisfied that the script will, upon execution, produce the desired state. Consider two successive iterations through edit and execute activities. For the first iteration, the user constructs a script S, composed of a sequence of n commands $C_1,\ C_2,\ C_3,\ ...\ C_m$. Then the user submits the script for execution, and the system will perform some activity. But there is no reason to presume that the system will interpret the entire script. For example, in a line-based dialogue the system will not interpret the last line of user commands until an `enter` or similar key is typed. Another example is a shell script containing a mistyped command, where the script is executed until the command is encountered. Thus S is partitioned into two sequences: E and

P. E is the prefix of S containing commands that have been executed and P is the remainder of S containing those commands whose execution is still pending. The ACS model assumes that the script is necessarily evaluated sequentially, that E and P are in order and never interleaved: $S = EP$. The execution of S will result in

$$\underbrace{C_1\ C_2\ \cdots\ C_{i-1}}_{E}\quad \underbrace{C_i\ C_{i+1}\ \cdots\ C_n}_{P}$$

After the execute activity terminates, control returns to the user who now edits the script. Let the edit activity result in a script S' of m commands C'_1, C'_2, ... C'_n. S' has two parts U' and M', where U' is the longest prefix of S' that is also a prefix of S, and M' is the balance of S'. Unless the user has made radical changes, C'_1 will be equal to the original C_1, C'_2 will be equal to the original C_2, and so on up to the first command C'_j which is different. C_1 to C'_1, then, make up a common initial equence of unchanged commands in the scripts. Subsequent commands in the script will have been modified by the user.

$$\underbrace{C_1\ C_2\ \cdots\ C_{j-1}}_{U'}\quad \underbrace{C'_j\ C'_{j+1}\ \cdots\ C'_m}_{M'}$$

U' should include at least the executed commands from the previous cycle. Any cycle in which E is not an initial subsequence of U' leaves the system in an inconsistent state. The process in which consistency is re-established is called recovery. When the user attempts to modify some parts of the script that the system has already executed, the system has to recover from the previous commands.

The script model is a simple and powerful model. However, the `undo/redo` concept is not well modelled. It describes an interaction as a script composition procedure and represents the action of undo and redo commands as editing operations performed on the script. This creates the side effect of precluding any possibility of directly representing the undoing of undo and redo commands themselves. As a consequence, the script model doesn't seem to help both the user in understanding undo support and the designer in supporting undo and redo functionality internally.

3.6.2 US&R model

Vitter [93] proposes a new interactive approach to undo support named US&R. The US&R model allows a user to specify the desired recovery using only a few simple undo support commands, `UNDO`, `SKIP`, and `REDO`, which can operate on other actions but not on each other. `UNDO` reverses the last action of the current state of the system, and that action is deleted from the state. `SKIP` causes the last undone action to be skipped in order to avoid redoing it; this skip action is appended to the current state. `REDO` causes some action that was previously undone to be redone and appended to the current state.

In order to represent the command script, the model uses a directed acyclic graph. In this graph commands are represented as nodes, and actual sequences of commands submitted by the user as paths through the graph. The current state of the system is then characterised as the sequence of actions still in effect, where an action is either the execution or the skip of a primitive command. Since the script records history by using a graph it can only represent the successor command choices for each command, and not the complete history. Thus, the user has to make decisions to choose which child to take when redoing a command.

With US&R, many kinds of recovery can be achieved. For example, suppose the user issues the primitive commmands C_1, C_2, C_3, and C_4. If the user executes an `UNDO` and then two more primitive commands C_5 and C_6, a new path on which C_5 and C_6 reside is added into the US&R data structure.

$$C_1 \to C_2 \to C_3 \to \begin{cases} C_4 \\ C_5 \to C_6 \diamond \end{cases}$$

where $\diamond$ points to the current node (in this example, C_6's node), which is defined to be the node for the last action in the current state of the system. Should the user wish to reverse C_5 and C_6 and re-issues C_4, it is adequate to execute two `UNDO`s, two `SKIP`s and one `REDO` consequently. Note that the last undone command is a logical choice for each `REDO`.

Although US&R is powerful undo/redo model, the relation between `SKIP` and `REDO` is ambiguous. There is no rule to specify when to redo an undone command or skipped undone commands except to let the system present the alternative to the user. The user does not know what the choices are until after issuing a `REDO` command. Besides, the history structure can become very complicated because of skipping and linking branches. This makes it difficult for a user to understand the model if the user is not familiar with the underlying data structure.

3.6.3 History undo model

The history undo method also allows a sequence of commands to be undone, but unlike the US&R method, appending the inverse of commands to the end of the history list as well as the executed commands. During the repeated undoing of commands, any command other than an undo command breaks the sequence of undo commands and makes the undo pointer move to the end of the history list [82]. A new undo command starts the process of undoing again from the end of the list. After such a situation arises the pevious inverse operations are considered as ordinary commands that can themselves be undone. For instance, in the sequence of commands C_1, C_2, C_3, C_4, assume that C_4 is undone. Then, the history list will be like this:

$$\begin{array}{ccccc} C_1 & C_2 & C_3 & C_4 & {C_4}^{-1} \\ & & \uparrow & & \end{array}$$

At this point, adding a new command, say C_5, breaks out of the undo mode and the pointer indicates the position of the C_5 command which has been appended to the end of the history list. Now, if two more undo commands are done, the history list will become:

$$\begin{array}{cccccccc} C_1 & C_2 & C_3 & C_4 & {C_4}^{-1} & C_5 & {C_5}^{-1} & {C_4}^{{-1}^{-1}} \\ & & & \uparrow & & & & \end{array}$$

In this way, it is possible to go back to any previous state in the history and the possibility of conflicts doesn't arise since commands are tracked in order of their execution. But, as far as our model is concerned, there is another possibility of re-executing a command that has been undone in a previous stage, even re-undoing the same command. To illustrate, two undo commands above are also performed on the command C_4, in which case it is called ${C_4}^{{-1}^{-1}}$. This stands for the C_4 command.

3.6.4 Triadic undo model

Another undo support model, called triadic undo model, is proposed by Yang [97]. In terms of functionality and commands provided, this model is quite similar to the US&R model. One similarity is that the triadic model has three undo support commands, but this time calling the third command `rotate`. The `rotate` command determines the next command to redo or the place where the undone command should be inserted. In the model, two command lists are maintained, a state list and an undone list. The state list is a list of task-oriented commands which have been executed and are still in effect. The undone list is a list of task-oriented commands which have been undone.

The triadic undo model states that:

(1) a one-step `undo` command will reverse the effect of the last command in the state list and record that command on the undone list, and a n step `undo` command will reverse the effects of the last n commands in the state list and record them on the undone list in the order of undoing.

(2) a one-step `redo` command will re-execute the last command in the undone list and a n step `redo` command will re-execute the last n commands of the undone list.

(3) the undone list can be rotated so that a desired command will be in the last position and will be re-executed when a following `redo` command is issued.

(4) when a user issues a task-oriented command the command will be executed and recorded on the state list.

With a proper sequence of these three commands, the triadic undo model is able to supply the required range of undo support. For instance, in order to undo an isolated command, the user has to undo a number of steps, rotate the undone list, and then redo a one-less number of steps. To redo an isolated command, the user has to rotate the undone list until the desired command appears and then call `redo`.

The data structure of the model is very simple and understandable. However, a wide submission of the undo support commands can complicate the structure;

e.g. random insertion makes the undone list become an unordered collection and so there is no indication which subsets were once sequential paths. This situation and the action of rotating the undone list cause another problem; a command selected from the undone list may not be redone because of the disappearance of the original state, and even if the command might be redone it could leave the system in an inconsistent state. Like the previous models, the triadic undo model assumes that commands can be redone in a state other than the one they were originally issued in.

3.6.5 Script-defined selective undo model

All the above undo models allow the reversal of commands from the last one backwards, although their `redo` capabilities can be used in a nonlinear way. To undo one command several steps back in the history, all subsequent commands must first be undone and then redone (skipping the command during the `redo` phase). In order to avoid this disruptive situation, Prakash and Knister [72] introduce a new model, called the script-defined selective undo model, which is based on transposing pairs of commands.

The model is basically used in a general framework [73] developed for undoing commands in goupware applications. It allows users to be able to undo and redo not only their own commands, but also other users' actions selectively, dealing explicitly with location shifting and dependencies among users' commands. In a groupware application, the last command done by a user may not necessarily be last in the history list and so undoing of a particular user's last command may involve an irregular movement over the history list, skipping other users' commands.

When an `undo` command is issued, the undo algorithm of the selective undo model works by making a copy of the end of the history list, from the command selected to undo onward. The undo operations are performed on this copy so that the original list is not affected by these operations. The selected command is repeatedly shifted until it reaches the end of the history list and then the inverse is applied. Before each shift, it is essential to check whether a conflict exists with the following command (see Section 5.3.6 for the definition of conflicts). If a conflict is found with a subsequent command, the selected command cannot be generally undone unless the subsequent command is undone too. To track which commands have previously been undone, the inverse of a command is appended to the end of the history list and a pointer named *do-undo* is placed into the history list which links a command to its corresponding undo operation.

The effect of selective undo is determined by a script interpretation. To illustrate how this kind of undo works, let the history list at some point be as follows:

$$C_1\ \ C_2\ \ C_3\ \ C_4$$

Assume that the commands C_2 and C_3 conflict, and there are no other conflicts. If the command C_3 is undone, the history list will have the following appearance:

$$C_1\ \ C_2\ \ \underbrace{C_3\ \ C_4\ \ C_3'^{-1}}$$

where the oval line beneath the sequence indicates a $do - undo$ pointer and C_3' is the command that results from shifting C_3 past C_4. The resulting list is equivalent to $[C_1,\ C_2,\ C_4']$, where C_4' is the command that would have been executed instead of C_4. For group text-editors, the commands C_3' and C_4' are usually identical to the commands C_3 and C_4, except that the position data is different.

Now, suppose the command C_2 is to be undone. The algorithm will first copy the history list from C_2 forwards into a temporary list so that the original list is not affected by shifting commands. Since there is a conflict between C_2 and C_3, and C_3 has a *do-undo* pointer, the C_3 and $C_3'^{-1}$ pair will be removed from the temporary list. Thus the new list will be $[C_2,\ C_4']$. Assuming that there is no conflict between C_2 and C_4', C_2 will be shifted past C_4' giving the command C_2'. Now that the command C_2 has been shifted to the end of the list, it can be succesfully undone using the command $C_2'^{-1}$. This command is then appended to the original history list, with the appropriate $do - undo$ pointers added, giving the desired result:

$$C_1\ \ \underbrace{C_2\ \ \underbrace{C_3\ \ C_4\ \ C_3'^{-1}}\ \ C_2'^{-1}}$$

Note that the resulting list will be equivalent to $[C_1,\ C_4'']$, where C_4'' is the command that derives from shifting C_3 and C_2 past C_4 until they are adjacent to ${C_3'}^{-1}$ and ${C_2'}^{-1}$, respectively, in which case the pairs $[C_3',\ {C_3'}^{-1}]$ and $[C_2',\ {C_2'}^{-1}]$ can be removed from the history list because they behave like an identity operation.

The script-defined selective undo model has a complex data structure. Addition of an undone command to the history list makes it more complex. Even though the $do - undo$ pointer indicates command pairs that are not currently in effect, it may be difficult to determine which commands the real state is composed of. The undoing of a command causes subsequent commands to be re-arranged if that command and the one corresponding to its inverse are removed from the current state. The modification of the original commands can prevent efficient use of the history list.

3.6.6 Direct selective undo model

A second selective undo model is introduced by Berlage [8, 9], called the direct selective undo model. The model is particularly developed for graphical user interfaces and also applied to text editor applications. A simplified version of this model is employed in GINA [81].

A selective `undo` command creates a copy of the selected command, executes the copy and appends it to the history list. The rest of the history list is not modified by this command. The direct selective undo model also provides a `redo` command which enables a user to redo commands selectively from the old history branches that are currently not in effect. These branches have been created during the previous undo operation and all the commands on them have been undone. To determine applicability of selective undo/redo to a command, the model looks at the command to undo/redo and the current state rather than the history in between. This is achieved by two separate methods that checks undoability or redoability of an arbitrary command from the history in the current state.

The distinguishing property of this model is that the effect of selective undo is not restricted to a script-based interpretation. The following example will illustrate this difference of interpretation. Assume that a user is working with a graphic editor and that the last five commands in the history list have carried out the following actions:

(1) creating a circle object with the current fill colour installed in a palette (C_1),

(2) resizing the object (C_2),

(3) recolouring the object by selecting a new colour (C_3),

(4) moving the object from the current position to another one (C_4),

(5) duplicating the object (C_5).

Later, if the user would like to go back to the old colour (i.e. by undoing C_3, script-defined selective undo will also affect the duplicated object; the undo of C_3 is interpreted by evaluating the history as if C_3 had never been submitted. The result is that the original as well as the duplicated object have the old colour again. But, when direct selective undo is used the history list will appear as follows:

$$C_1 \;\; C_2 \;\; C_3 \;\; C_4 \;\; C_5 \;\; C_3'^{-1}$$

where $C_3'^{-1}$ recolours only the original object by selecting the old colour without affecting the colour of the duplicated object. In fact, this inverse copy ($C_3'^{-1}$) created by direct selective undo is an instance of C_3, but with exchanged parameters.

In general, since a special change the `undo` command causes on an object is moved to all figures in which that object appears at the current state, it also influences the object which resides at a different position during undoing, regardless of any other intervening changes. This includes the modifications on the object such as colour and size. In the above example, selective undo of C_1 will not make sense in the current state because of the dependency of C_2 on C_1, i.e. the object would no longer be available at the time of execution of C_2. In such cases, the method prevents the user from executing the selective `undo` command on C_1. If C_2 is undone before C_1, in which case the size of the original object changes to that of the object created by C_1, then C_1 can be undone.

3.6.7 Object-based undo model

With the above two selective undo models, the user can select any command in the history list to undo/redo. However, these models do not allow selecting any object and changing its state by `undo/redo`, without having to reverse the effects of the entire command. To support such object-based recovery operations, an object-based undo model is proposed by Zhou and Imamiya [100]. The model allows the user to select either any command in the history list or any object-based operation of that command to undo/redo.

The object-based undo model reorganises the commands stored in a single history list into subhistories corresponding to the objects so as to support the new functionality. This is done because more than one command might be executed in a single interactive cycle even if only one of them is submitted by the user (see Section 3.6.1). Commands executed in the same interactive cycle are defined as synchronous commands. Commands executed in different interactive cycles but influencing each other are defined as asynchronous commands. Objects operated on by synchronous commands are recorded in the order they were operated on. In this way, the synchronous commands can be undone or redone together so as to protect the dependence among corresponding objects. Asynchronous commands exist in subhistories and often concern undo-possible and redo-possible verification.

The undo mechanism of the model can be applied to both main history and subhistories, which both form the recovery history. Since the main history represents the sequence of interactive cycles, selectively undoing or redoing of a command as a whole is based on the main history (global undo/redo). A subhistory solely belongs to an object and thus the subhistories enable the user to select any object to undo or redo (local undo/redo). There are two concepts of the commands recorded in the recovery history, active commands and inactive commands. The rules that determine whether a command is active or inactive are as follows:

1. A new command is an active command in the done state. A previously active command (if it exists) affecting the same attributes of the same object is set as inactive.

2. An `undo` command makes an active command inactive. Among the inactive commands (if they exist) in a done or redone state affecting the same attributes of the same object, the one executed latest is executed again and set as active.
3. A `redo` command makes an inactive command active. A previously active command (if it exists) affecting the same attributes of the same object is set as inactive.

These rules state that it is not always the case that a command in a done or redone state is an active command, because the result of such a command may be removed or replaced by other commands executed later. Undoing an active command which operates on an object in a subhistory involves checking commands in other subhistories for the possibility of any dependency, which makes the undo operation impossible. Redoing an inactive command needs a similar checking procedure.

3.6.8 Multi-user undo model

In the previous sections, we have concentrated on the single undo/redo models. Much work has already been done concerning multi-user undo/redo. Below this literature is reviewed briefly. An examination in more detail can be found in Chapter 8.

Abowd and Dix [1] investigated the problems of providing an undo support facility in a synchronous shared or group editor. The investigation led to two separate kinds of shared (multi-user) undo, a local undo operating on the user's own actions and a global undo operating on the combined actions of all users. A formal model is used to resolve the conflict between the system's preference for global undo and the users' preference for local undo. Their analysis interpretes undo as an intention of the user, not an aspect of system functionality. This intention can be supported either by backward error recovery or by forward error recovery. Nonetheless, they suggested that designers must focus on supplying tools to facilitate recovery from a recognised erroneous state by use of any system function.

Choudhary and Dewan [24] developed a multi-user undo/redo model by extending a single-user undo/redo model that had previously been implemented as part of a system. As a reason for the absence of undo/redo from previous multi-user user-interfaces, they put forward the lack of semantic and implementation undo/redo models. Thus, the model incorporates a semantic model and an implementation model that makes it applicable to general multi-user programs. The semantic model deals with issues such as the construction of command histories and undoing/redoing of commands, while the implementation model offers a framework for implementing the semantic model for a particular multi-user program. Besides, they identified a number of requirements that a multi-user undo/redo should satisfy [25, 30].

A similar framework is presented by Prakash and Knister [73]. This framework splits the undo support facility into two parts, local undo and global undo. Local undo deals with the correction of a user's own changes to the state. Global undo copes with reversing other users' actions.

Berlage and Genau made a general consideration of undo in multi-user applications, in particular in an application framework called GINA, and analysed a concept of resolving conflicts which selective undo/redo can cause [10, 11].

Chapter 4

Undo and Command Analysis

Interactive systems can allow users to recover from their commands in various ways. The most desirable way is that selective undo is provided. A selective undo mechanism needs more space and recovery-related computation than all the other undo mechanisms. On the other hand, a system in which an undo command is not included can still support recovery requirements, e.g. via the use of other commands. From the user's point of view, however, this does not seem to be a convenient way because the user might not be able to determine how to reverse the effects of a particular command.

This chapter carries out an analysis of undo commands and task-oriented commands. Its scope is confined to the following three topics:

- comparing undo mechanisms,
- conceptualising the undo command,
- examining undo's role in state transformations,
- classifying the Unix commands in common use.

As a part of these topics, three kinds and overheads of undo are described, and the Unix commands are examined in terms of their functions and effects.

4.1 Introduction

It is an essential feature of interactive user interfaces to provide the capability to recover from unwanted or incorrect actions. Many users need to have user-oriented control over reversible actions, manipulating them at will. An "UNDO" function, which reverses the effect of any previous action taken, is often used to provide this capability. The ability to immediately recover from an inadvertent or erroneous action (or an action with different results than expected) has several advantages for users [18]:

(1) It can eliminate the necessity to reinitiate a whole series of actions that led to the previous step in the dialogue.

(2) One can recover from mistakes that might otherwise have led to the destruction of data, requiring reentry or perhaps permanent loss of those data.

(3) It can minimize users' fear of getting into a situation from which they do not know how to return.

(4) By trial-and-error exploration, users can learn the system or expand their knowledge to functions they have not previously used.

In one sentence, UNDO makes user interactions much easier and faster. As an example, consider the following situation: A user is not sure which of the titles shown on a menu or selection list performs the desired function. The user chooses the one that seems most likely, but the result is not what was wanted, and the original data have been modified. In this case, the UNDO function permits the user to reverse the last transaction and thus avoid having to reenter or reconstruct the original data.

The provision of undo for an ordinary command involves storing enough information to reverse that command. After executing a command `k`, in order to determine the prior state s, undo needs to know more than just the current state `ks` [41]. For example, let state s_a consist of the string 'abcd' and state s_b consist of the string 'bacd'. If f is the command to change all 'a' to 'b', then $f(s_a) = f(s_b) =$ 'bbcd'. That is, there exist states s_a and s_b and commands f such that:

$$s_a \neq s_b, \text{ and } s_c = f(s_a) = f(s_b);$$

but then

$$u(f, s_c) = s_a \neq s_b = u(f, s_c)$$

Therefore, knowledge of the current state and the edit command that produced it is not sufficient to determine the prior state uniquely, because in general commands do not have unique inverses.

Similar situations can be observed in operating system environments as well. For example, consider two different states; one contains *fileA* and the other contains *fileB*. When a Unix command *mv file* fileC* is applied to these states the resulting state will contain *fileC*. In this case, it is not possible to determine which of the two states the command has been executed in, unless information about the previous states or command history is stored.

4.2 Undo Analysis

The achievement of recovery in interactive systems does not always come with undo. The provision of undo is not really the only way to allow recovery. Some system commands can behave like undo on a particular command. This section gives a functional analysis of undo and presents undo's role in recovering from erroneous commands.

4.2.1 What is undo?

Undo is basically a command that can have the ability to act on all other commands. It provides some way of manipulating the issued commands. The user can make undo operate on any issued command that does not match the desired current state properly. By inserting commands into or removing them from the current state, an interactive system with undo facilities can be moved in both directions, i.e. backwards and forwards. In this way, it is possible to be able to navigate through previous system states.

Undo is conceptualized in various forms in some analyses of undo support. Gordon et al [41] consider undo as a normal function rather than a special function. This conceptualization contrasts with the analysis of Yang [97], who emphasizes

> *The semantics of an undo command is different from that of a task-oriented command. The semantics of an undo command depends upon its context. For example, in $uf(e)$, u is equivalent to f^{-1}. In $ug(e)$, u is equivalent to g^{-1}. If $f^{-1} \neq g^{-1}$, u in $uf(e)$ does not mean the same command as u in $ug(e)$ although both occurrences of u have the same syntactic form.*

Of course a task-oriented command may depend on the context, e.g. taking the current directory as an implicit argument. However, the point is that the `undo` command carries out a different operation for each task-oriented command. It is capable of operating on all the commands, showing a command-dependent behaviour.

A similar analysis of undo is done by Abowd and Dix [1]. In their analysis, the `undo` command does not function as a system command. Instead it is interpreted as the user's intention, i.e. the purpose of undo is to allow the user to recover from a recognised erroneous state. The user may achieve this intention by use of any system command, not necessarily the `undo` command. However, undo is extremely useful to reduce the risk in interaction. Besides the presence of user's errors while interacting, another kind of risk is the mismatch between the user's and system's point of view on the same command [61].

In our design, undoing is a process during which, rather than a new task being performed, an old task is rolled back. It is normally done by removing the effects of the task from the current state. In other words, with undo, the components of the system which have been modified by that task are returned to one of their previous states. A previously-existing state becomes effective again. Consequently, we treat undo as a task-inactivating command.

4.2.2 Error recovery with undo

There are very few interactive systems which consider undo facilities for errors from the outset. A sensible reason for this is that it is exceedingly difficult to implement a mechanism for undo support thoroughly due to the existence of non-undoable interactions with the world [87]. A prototype design may not make it possible to predict the consequences of user errors. The experience with the system at the end of the design process has a significant role in the consideration of undo. In consequence, such a way helps not only minimise user errors, but also redeem all design faults. Instead of including undo from the start of the design, it is preferable to take feasibility of this capability on a completed design into account.

Strategies for dealing with the damage done to the system state when an error is encountered can generally be classified into backward and forward error recovery techniques. Such techniques aim to place the system in a state from which interaction can proceed and the error can be repaired. They can also be composed of some further subtechniques that have emerged because of the possibility of different considerations of undo support for every command interactive systems provide. These subtechniques, in fact, are relevant to efficiency and will be introduced in Section 6.4.2.

4.2.3 Backward error recovery

Backward error recovery involves the provision of recovery points (*checkpoints*) which represent the initial or intermediate state of a system. It is a method by which a state that is error-free can be recorded within the system and later reinstated at will. The system is usually reinforced with an undo command which is explicitly used, for instance, by typing a script or clicking on a button, and which keeps the system in a consistent state. The user invokes the undo command to undo the previous command in response to a recovery request [1].

With this sort of recovery, the system moves back to a preceding state by undoing everything done since passing its last recovery point, not just the things it did wrongly [74]. In this way, when an error occurs the user restores the system to one of the checkpoints prior to the error by employing the undo command without invoking any other system command. Then the interaction begins with the right commands again. The system behaves as if the undone command was not executed. The effect of the undo command itself can be destroyed by a redo command. In a system with no redo command, resubmitting the undone command can achieve the same task.

4.2.4 Forward error recovery

Forward error recovery is a technique by which the user who wishes to recover from an erroneous state may achieve this intention by use of any command supported by the system. It is to a very large extent dependent on having identified the error, or at least all its consequences. The techniques relating to forward error recovery must be designed as integral parts of the system they serve. Recovery is considered on parts of the system affected by only the faulty command. Therefore other parts of the system and thus the effects of other executed commands are free from the recovery operations [74].

This type of the system's behaviour can be seen as an implicit undo, where the user performs ordinary system commands to undo the effects of incorrect ones. For example, a mistyped character in a text editor can be deleted without explicitly using the undo command. As a second example, the effect of the Unix *mv* command can be reversed by using another *mv* command unless some other commands intervene. This emphasizes the fact that the non-existence of an undo command may not necessarily mean that the system doesn't provide a recovery facility.

It doesn't seem practical to provide undo support by constructing for each system command a new command which reverses it. A reason for this is that it is very difficult for the user to remember the pair of commands that recover from each other's effects (although it might be possible to generate the reversing command automatically). This also causes the system to be overloaded with many unnecessary commands, because none of them is a task-oriented command and so offers an important system service. The most convenient way to bind the operations corresponding to the reversal of all system commands is apparently to use only one command (undo).

4.3 Overheads of Recovery Commands

The current recovery facilities introduced for interactive software are basically concerned with two kinds of undo, *single undo* and *multiple undo*. Provision of an undo capability imposes some overheads on systems. This section discusses the undo facilities and compares their necessities.

4.3.1 Single undo

Single (or single-step) undo is a facility that is most commonly used in Macintosh applications as well as editors such as *vi* and undoes the last command the user performs, but not the ones prior to that command. The use of two consecutive undo operations neutralises each other (i.e. the effect of the last command remains unchanged). The same procedure is usually used to implement the redoing of the last command.

This facility is very effective because it is so simple to operate. The user interface is simple and clear, easy to describe and remember. The most recent command issued is always in the domain of the undo and redo operations. This is by far the most frequently implemented undo, and it is certainly adequate, if not optimal, for many programs. For some users, the absence of this simple undo is sufficient grounds to abandon a product entirely [26].

Users generally notice most of their command mistakes right away: something about what they did doesn't feel or look right, so they pause to evaluate the situation. If the representation is clear, they see their mistake and select the undo function to set things back to the previously correct state, then proceed from that point.

The biggest limitation of single-step undo is when the user accidentally short-circuits the ability of the undo facility to reverse the desired action. This problem crops up when the user doesn't notice a mistake immediately. For example, assume someone deletes six paragraphs of text, then deletes one word, then decides that the six paragraphs were erroneously deleted and should be replaced. Unfortunately, performing undo now merely brings back the one word, and the six paragraphs are lost forever. The undo function has failed to achieve the expected recovery by behaving literally rather than practically, because the six paragraphs are more important than the single word.

4.3.2 Multiple undo

Multiple undo, which can revert more than one previous command, can be generally classified into two rudimentary categories, *sequential undo* and *selective undo*. Each classification provides a significantly divergent level of recovery facilities. In either case, users can modify the interactive history by using recovery commands (undo and redo). More information about the models that provide multiple undo is given in Chapter 3.

The basic idea of sequential undo is that users have to follow a linear path in undoing and redoing commands. The order in which commands are submitted specifies the procedure of carrying out recovery operations. The most recently submitted command is the first command that is affected by recovery operations. To undo a command prior to the last command in the history, all intervening commands have to be undone as well (and then reissued, if necessary).

Typical examples of sequential undo are history undo [82], US&R [93], and triadic undo [97] (see Figure 4.1). Although these undos all have sequential undo facilities, their redo facilities vary considerably.

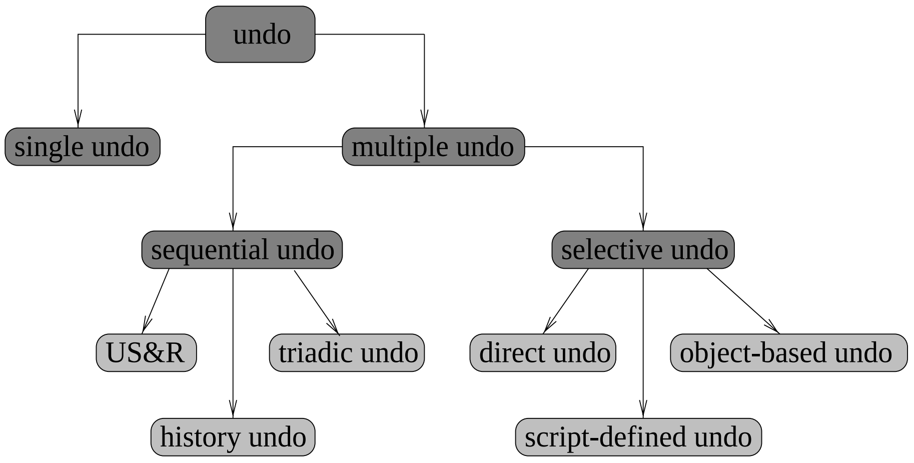

Figure 4.1: The current undo capabilities

The notion of selective undo is to allow users to undo or redo commands in an arbitrary way regardless of which order they have been executed in. It is possible to remove any executed command from the interactive history without having to first deal with the intermediate commands. Similarly, redoing a command doesn't require operations on intermediate commands. However, this kind of facility has some limitations. The most important limitation is that recovery operations on a command may have an effect on other commands of the history, making the system appear in an inconsistent state. This is called a *conflict* in the existing models. Therefore, every attempt at recovery involves determining whether there is a conflict between commands or not.

Some typical examples of selective undo are script-defined undo [72], direct undo [9], and object-based undo [100] (see Figure 4.1). These undos support efficient use of recovery facilities for the isolated commands of interactive history in particular. Other commands may not be undone easily because of the restrictive nature of selective undo.

4.3.3 Comparison of undo capabilities

In order to provide a particular one of these undo capabilities, a system must meet some requirements. The easiest undo facility to provide is one that restricts undo's domain to the reversal of only the last command (single undo). Selective undo is the most difficult capability since undoing a command selectively is likely to affect some other commands.

An interactive program with single undo stores information for only the last command so that it can support undo. This information is concerned with the changes that are made by the command to the state. Execution of a new command causes the program to release the information of the previous command and to store that of the new command. Similarly, the program also keeps the effects of the undo function on the state to allow a second undo to reverse the first one (redoing the last command).

As far as sequential undo and selective undo are concerned, all recovery commands have to carry out their operations in a state-dependent way since recoverability is essentially determined by the state. Commands that they handle become meaningful only if executed in an appropriate state. The overheads recovery commands impose on the system can be characterized in six particular points:

(1) The movement over the history of commands

(2) The applicability of recovery commands

(3) The kind of state to be reached

(4) The monitoring of commands

(5) The number of file versions to be saved

(6) The making of backups

Sequential undo doesn't allow an arbitrary movement over the history list to undo commands. A command is always undone in the state that was reached after the original execution. Commands always make sense to undo. The same is true for sequential redo. A command is always redone in the same state that it was originally executed in. But some kinds of sequential undo support arbitrary redoing by skipping the next command(s). This forces undone commands to be redone in a different state from the original one and so many commands may not be naturally redone [8].

Selective undo enables users to freely move back and forth over the history list. Recovering from a command involves analysing past part of the interaction to see whether or not it creates a conflict, i.e. it makes sense to undo/redo or destroys the effects of any other command. The models that have selective undo themselves have to deal with such problems by employing conflict-detecting mechanisms with various techniques. The applicability of recovery commands depends generally upon the current state. In many interactive environments, each submitted command can tend to partly or completely destroy the previous commands. This makes recovery commands inapplicable and thus the destroyed ones have to be handled first.

Saving is what recovery mechanisms have to do just before commands are carried out. The amount of information to save for each command, as well as the kind of that command, varies according to the kind of facility the mechanism provides. Given commands that involve changing where data resides or renaming it, sequential undo requires relatively less information to save than selective undo. For instance, assume a recovery mechanism in Unix uses the partial checkpoint recovery technique [4]. When a move command ($mv\ f_1\ f_2$) is to be executed, storing only f_2 if it exists before the execution would be adequate for purposes of sequential undo. However, selective undo involves saving both of the files because f_2 which is to carry the contents of f_1 can be accessed and modified by later commands.

4.4 Command Classification

It is very difficult to satisfy the operational requirements of the user with a small set of commands. The number of commands available in a system can affect the ease of inclusion of undo support into that system. This section carries out a functional examination of commands provided by various Unix systems. Classifying these commands according their effects on the system state, it also defines the domain of undo support.

4.4.1 Functional classification

The Unix operating system presents users with a large number of commands. But, most of them are of uncommon use because of reasons such as difficulty of use and the infrequent need for them. For example, although there are over 400 commands available to users, they use only a small proportion of these commands frequently. A list of the most frequently used commands from the command line can be found in [45, 88]. We will make a new classification of commands for the undo/redo model. Thus, the commands which are of frequent use can be classified into six groups in terms of their major functions.

(1) *File management commands* (`management`) that deal with shaping text and other system objects have the most important effect on the file system. Execution of some commands in this group may render the system irrecoverable from them unless the required data is saved. The management commands can include both text editors that provide many editing primitives in a particular context and stand-alone Unix programs that perform various actions on files and directories. For example, the text editors *ed* and *vi* enable text files to be manipulated by using primitives such as insert, copy, delete and move through their own environments; *ln* establishes a symbolic link to a files; and *cd* moves a user through the hierarchical file system.

(2) *Orienting commands* (`status`) inform users about the environment in which they are working, the status of objects and processes in that environment. Since the consequences of their actions are associated with the status information they have no effect on the state of the system. This group can incorporate commands that show users where they are in the file system (e.g.,

pwd), what processes they are currently running (e.g., *ps*), and how much usage they have on the system disk (e.g., *du*, *quota*).

(3) *Process management commands* (`process`) don't make sense when they are run on their own and behave like the operators in a programming language. They allow users both to enter more than one command at a time and manipulate them. Sometimes, it may be difficult to predict what effect they have on the state. These commands have functions such as redirecting the output of a command to a file (>), piping the output of a command to the input of another (|), arranging the sequential execution of commands (;), and executing a command in the background (&). While writing the standard output to a file, for example, it must be checked whether that file already exists. If so, the state may become irreparable.

(4) *Communication commands* (`communication`) allow users to communicate with each other or devices such as a printer on the network and exchange information through the various system programs. Most of them can be seen as real-time commands because of communication with the physical world. For instance, *lpr* sends a request for printing out a specified file to the printer; *mail* helps users handle mail messages both in receiving from and sending to a user; and *write* permits direct on-line communication with other users.

(5) *Text processing commands* (`text`) perform some searching operations on the system files and their contents. The files are processed in terms of the object to be searched and the results are reflected on the terminal. Like orienting commands, the state is not affected by execution of these commands. We can include *diff* which compares two files and searches for line-by-line differences between them, *grep* which seeks a specified string in one or several files, and *wc* which finds the number of lines, words, and characters in one or more files.

(6) *Task specific commands* (`task`) are used to fulfil some specific job assignments. Just as the commands of this group have many uncommon features, there are many kinds of assignments. As an example, *kill* is used to terminate a background process, while *at* is required for scheduling a script file to be run at a specified time and date. Besides, *tee* transfers data from the standard input to the standard output, making a copy in one or more files. In particular, it is useful to sample data at an intermediate point in a pipeline by storing the

Command	Function	Category	Group
cat	Show contents of file	ineffectual	status
cd	Change to a new directory	reversible	management
chmod	Change file permissions	reversible	management
|	Pipe output of commands		process
cp	Copy files	reversible	management
diff	Display different lines in two files	ineffectual	text
;	Execute commands sequentially		process
find	Find files	ineffectual	text
grep	Find pattern in file	ineffectual	text
kill	Terminate processes	irreversible	task
ln	Make links to files	reversible	management
>	Redirect output of command to file		process
lpr	Send requests to printer	irreversible	communication
ls	List contents of directory	ineffectual	status
mail	Send or read mail	irreversible	communication
mkdir	Make directories	reversible	management
&	Execute a background process		process
mv	Move or rename files	reversible	management
ps	Report process status	ineffectual	status
pwd	Identify current directory	ineffectual	status
rm	Delete files	reversible	management
<	Read input from a file		process
rmdir	Delete directories	reversible	management
sort	Sort or merge files	ineffectual	text
talk	Talk to another user	irreversible	communication
tee	Replicate the standard output	reversible	task
>>	Add output of command to file		process
vi	Screen editor	reversible	management
wc	Make word count in file	ineffectual	text
write	Write to another user	irreversible	communication

Table 4.1: The 30 Most Frequent Unix Commands From Command-Line Data

intermediate results in a file or diverting them to the standard output.

In addition to the commands given in the above classification, Table I shows the most frequently used commands along with a description of each command. Using process management commands, a user can form quite a complicated sequence of commands in the command-line data. Although this case seems advantageous to users, the same thing cannot be said for a recovery mechanism. Reversing the effect of such commands is not as easy as it sounds. They can be treated as if embedded in a script file. The recovery mechanism may have to determine whether all commands in the script work correctly or not. In the case of execution of more than one command from the same command-line data, the Unix shell runs the script until encountering an invalid command, regardless of checking whether each command in the script is valid. Therefore, if there is an invalid command in the script, the execution of the script will be aborted and the preceding commands will remain executed. That is the point when the user needs to use the recovery mechanism more frequently, so that the script can be run again after making it entirely executable. Thus, as the user can recover from these kinds of commands, the mechanism must save required data for each valid command in the script according to the order of their executions. The issue of how to identify a valid command will be examined in Chapter 7.

4.4.2 The domain of undo support

In the preceding section, we classified the commands according to the six crucial functions they serve. This sort of classification doesn't meet the need of detecting the commands our model has to handle for recovery. Therefore, in this section, we analyse commands in terms of the effects they cause on the state. The analyses are based primarily on our information of the Unix operating system and also supported by the functional classification above.

An ideal recovery mechanism would allow a user to reverse all the effects of a command that has been executed [97]. But, on many large systems such as Unix there are a lot of commands which can't be reversed. However, the effects of some commands don't need to be reversed. The domain of a general undo support facility to be implemented for Unix constitutes a subset of all the commands which a user can issue on this system. Within the context of carrying out this undo facility, the Unix commands can be classified into three main categories (see Table I):

- ineffectual commands,
- reversible commands,
- irreversible commands.

An *ineffectual* command is a command which has no effect on the system state but has significant effects on the appearance of a screen. It provides information about the state of an existing object such as the system clock and a processed file. The orienting and text processing commands given above can be thought of as being in this category.

A *reversible* command is one whose effect on the state of a system can be reversed. Generally, since commands have differing degrees of 'reversibility', there are differing difficulties of being able to reverse them. This diversity affects the amount of information to be saved. Some commands need the whole contents of a file to be kept (e.g., *rm*), while other commands need nothing substantial to be saved (e.g., *rmdir*). The file management commands and some of the task specific commands are reversible commands.

An *irreversible* command is a command whose effect may not be reversed. Irreversibility of a command is caused by the physical effects of that command and the scope of undo support. Physical irreversibility can be thought as a result of occurrence of a real-time communication within a system (e.g., *write*) or sending of information beyond the boundary of a system (e.g., *mail*). The above communication commands construct a remarkable group of irreversible commands. On the other hand, for undo support that can only reverse a limited number of commands issued in the current interactive session, other commands cannot be reversed.

Given the fact that operating systems contain very complicated commands, there should be no need to parse the command line. For example, the fact that each command in Unix is provided with a set of options makes a recovery-associated analysis of commands more complicated. An option can expand the functionality of a command, increasing its effects considerably. Consider the command `rm -r junk`. The command normally operates on files, but when used with `-r` it can affect directories.

The Unix operating system does not provide version numbers in file names and so updating a file in Unix automatically destroys all record of what the file used to contain. Thus, some editors themselves deal with this task. Saving a file from Emacs copies the old contents to another file which is called the "backup" file and named the same as the corresponding original file but with a tilde (~) appended. The backup file contains the contents from before the current editing session. But, the Vi editor doesn't save a backup of the original file. This problem can obviously be solved by creating a command which, when invoked, first makes the necessary file backups using the Unix cp command, then invokes Vi in the usual way. When file versions are held in this way, there is no need for a recovery mechanism, since those versions are still in the current state. But the previous version of a file that is called through an editor and modified cannot be stored, which requires control of dynamic accesses.

4.5 Effects of Commands

The ability of being able to reverse commands depends entirely upon their effects on the file system. The mechanisms presented in the previous section introduce some ways of achieving the reversal of commands. The reversing operation for a particular command only deals with the permanent effects of that command and so these effects detected need to be analysed. Most of the permanent effects are usually associated with the modification of file contents. To undo those effects it seems to be adequate that a copy of the old content is taken. However, it may not be possible to reverse some effects.

First, there can be other local state changes such as changing permissions or other attributes on files, or changing the directory structure. These state changes can generally be dealt with in the same way as with files. In the case of directories, we will see in Section 7 that it is not necessary to keep old versions of directories.

Second, there are changes to the state of the shell itself, e.g. changing the current directory, altering the values of environment variables, keeping track of command aliases and so on. Similarly, some shell commands are intended to be used in scripts as programming tools, e.g. for controlling loops. These are all necessarily commands which are built in to the shell. As they are under the direct control of the shell, they cause few problems; indeed, the shell state can be stored in files and the mechanisms for keeping track of versions of files can be used for these too.

Third, programs may have external effects, such as interacting with the user, printing a document, sending an email, communicating interactively with another user or changing remotely stored files. Tracking these effects can be difficult [87]. Although the relevant system calls can be trapped, it may be difficult to determine what they do or how to undo them. In general, these external effects cannot be undone, and we regard these issues as being beyond the scope of our shell.

Finally, some shell commands deal with concurrent process control and interactions between multiple users. This includes compound commands which run several programs together, possibly communicating with each other via pipes. It also includes direct process control commands which create long-running processes which execute concurrently with the shell and each other. Undoing the effects of a complete process group may be relatively easy, but undoing the effects of individual processes within the group is more difficult. Concurrency issues are discussed further in Section 8.

In addition to tracking the effects of commands in order to undo them, it is also desirable to control the effects of potentially untrustworthy foreign programs. It is conventional for World Wide Web applets to be run in extremely restricted environments which effectively prevent them from making any local changes or accessing any local information. In addition, Goldberg et. al. [40] discuss ways to restrict helper applications which get run as a result of downloading files from the Web. However, it is also common to obtain complete application programs directly from the Web or other untrustworthy sources, and at present, such programs are usually run with no restrictions at all.

To some extent, providing an undo facility in itself provides some protection from such programs; if a program deletes files it shouldn't, they can be recovered easily. However, stricter knowledge and control of what a program is doing is desirable. It is possible to use a restricted shell to prevent a user from doing damage by forbidding access to unsafe programs. However, our aim is rather different. We want to allow a user to run any desired program, but to prevent the program from doing damage by running it in a restricted way. We want the program to run in its normal environment, allowing it to do useful things such as creating files. However, since all file accesses are monitored, a security policy can be applied, and any unauthorised access is forbidden or referred to the user for confirmation.

One further issue which needs to be addressed is the question of aliases. If there are two names for the same file and a command writes to the file using one name, it also changes the contents visible via the other name. The shell needs to know about this to keep track of the effects of commands. In a concurrent setting, if two commands attempt to access the same file via different names, the resulting contention problem needs to be recognised and addressed by the shell.

If all commands which create or manipulate aliases are issued by the proposed shell, it has complete control and can keep track of them. Otherwise, alias problems have to be detected as and when they occur. Detection methods differ from one operating system to another, and detection of all cases of aliasing can be difficult. However, reasonably good solutions are usually possible. For example, given a pathname on Unix, a unique device identity number and file identity number (inode number) can be obtained and used to record where the file is physically held.

Chapter 5

Specification

The most important requirement of recovery mechanisms is that all user interactions must be observed very carefully. In Unix, this is related to an elaborate observation of each process created. Existing shells have very restricted control over processes. A shell starts the execution of a process that runs a particular program. It is also capable of suspending or terminating an executing process. But, once the process is started, it takes over control of the execution and then cooperates with the kernel to complete the execution. The shell doesn't deal with the process any more, except that it might send signals to the process or wait for it to terminate. The process itself manages its interactions with other components of the system, such as the kernel. It is thus impossible for an ordinary shell to gain information about what the process is doing.

In this chapter we specify the provision of a recovery mechanism transparently. First, we describe a simple and consistent mental model for the user. Second, the requirements of an operating system to be able to support `brush` are given. Third, some `brush` concepts (transactions, conflicts, etc.) are presented, describing the way in which a user command is modelled as a series of transactions and the need for insertions into the history list. Finally, the the chapter deals with the description of `bmake`, our automated version of the Unix *make* utility.

5.1 A Mental Model for Undoing

In this section we describe how undo and redo facilities look to the user. The description concentrates on changes to individual files, but applies to other permanent resources such as directories. Concurrency issues are deferred to Chapter 8.

To allow the user to undo previous commands, the shell provides a history list, i.e. a list of the commands which have been issued recently, either up to some fixed number of commands, or back to some previous commit point such as a logout. Many text-based shells already provide such a history list, where each entry records the text that was typed. In the case of graphical interfaces, a textual description or other visual representation of the recently issued commands needs to be provided.

With text-based shells which provide a history list, it is common to provide a *resubmit* facility. This allows the text of a previously issued command to be copied, edited if desired, and then submitted as a new command. This is a cut-and-paste facility which saves time by reducing the amount of typing the user needs to do. The undo and redo facilities described here can be added without affecting the resubmit feature.

A selective undo facility is provided where the user can select one of the previously submitted commands and ask for its effects to be reversed. We assume that there is a `write` command which creates a file from nothing (this might just be a particular way of using an editor), and `edit`, `move` and `copy` commands which have obvious effects:

```
write fileA
edit fileA
move fileB fileC
copy fileC fileD
```

The next command might be a request to undo the `edit` command. The new version of the file `fileA` is removed from view and saved, and the old version of the file `fileA` (which was saved when the `edit` command was issued) is reinstated. The appearance of the `edit` command in the history list is changed, perhaps by being greyed out or having a different colour, to indicate that it has been undone. Here, we use brackets for this purpose:

```
write fileA
(edit fileA)
move fileB fileC
copy fileC fileD
```

This change of appearance acts as a record of the undo command, and so the undo command itself need not appear explicitly in the history list. From now on, we refer to the commands in the history list as *active* or *inactive* according to their current status and hence appearance (see Section 5.3.5). The `undo` command can only be applied to active commands.

A redo command is provided to reverse the effects of undoing. An inactive command such as the `edit` command above is selected, and the original effects of the command are reinstated. The appearance of the command in the history list is also reinstated. As undo can only be applied to an active command, and redo can only be applied to an inactive command, a single command name or mouse button or keystroke can be used for both.

The redo feature is very different from the resubmit facility. What happens is that the changes to file versions carried out by the undo command are reversed. The new version of `fileA`, which was saved when the undo was issued, is reinstated. This contrasts with a resubmit where the editor is re-executed.

The intention of these facilities is that when a command is undone, the state of the filing system is exactly the same as if the command had never been issued. When it is redone, the state is exactly the same as if the undo had never been requested. This is an important principle which makes it easy to understand the meaning of undo and redo. It can be stated as an invariance condition:

The current state should be completely determined by the initial state and the active commands in the history list.

In addition, there is a stronger requirement:

If the active commands were to be executed on the initial state, each command would act on the same file versions as it did when it was originally executed.

This provides a very simple mental model for the user of what undo and redo mean. However, it follows that undo and redo are not always possible because of dependencies between commands, as described by Prakash & Knister [72]. For example, the `move` command above cannot be undone because `fileC` would not be available for the following `copy` command. This brings into question another important principle:

> *It should be possible to return to any previous state.*

In fact this is possible with the facilities described so far, though it is not necessarily very convenient. It is always possible to undo the last active command in the history list, and so to undo all the active commands sequentially, from the last one backwards. After that, the active commands in the desired previous state can be redone in a forward direction. Of course, it may be possible to achieve the desired result more efficiently in practice.

A feature of the system which would make undoing more convenient would be that when an undo is requested for which there are later dependent commands, the system could offer to undo those later commands at the same time as the requested command. Similarly, a request to redo may result in an offer to undo or redo other commands as well, to make the requested one possible. The issue of what constitutes a dependency will be explored further later.

A further principle which needs to be addressed is whether desired versions of files can always be recovered. Specifically:

> *It should be possible to recover any desired collection of versions of files, or other resources.*

This is difficult if we want to recover two different versions of the same file, or versions of two files which do not correspond with each other. A simple example of this is the following:

```
write fileA
edit fileA
```

The first command creates one version of `fileA` and the second replaces it with a new version. Suppose it is now discovered that important information was deleted by accident during the editing session. Important new information was also added during the editing session, so both versions are needed in order to resolve the situation.

A further feature of the design which takes care of this sort of problem is the ability to insert a command into the history at an arbitrary point. Inserted commands, as with undo and redo, have to be checked for dependencies before being allowed. In the simple example above, the recovery procedure would be first to undo the `edit` command, then insert a copy command before it, and then redo the `edit` command, to give:

```
write fileA
copy fileA fileB
edit fileA
```

Now both versions of the file are available as `fileA` and `fileB`.

It is important to make sure that the proposed shell behaves correctly when commands fail. For example, suppose the user types `move fileA fileB` at a time when `fileA` does not exist. The state of the system should be made the same as if the command had never been issued. The command can be added to the history list as an inactive command, to allow editing of its text and re-submission.

5.2 System and User Requirements

A general recovery facility must be easy both to implement and to use in an operating system. This section presents basic requirements in order for operating systems to allow providing user-level recovery facilities easily and users to employ these facilities effectively.

5.2.1 Operating System Requirements

The ideas involved in `brush` can be implemented, in principle, on a wide variety of operating systems. However, there are two requirements which an operating system must satisfy in order to be able to support a `brush`-like interface.

The first is that the kernel of the operating system must use hardware memory management and user mode facilities to protect the kernel from user processes, and to protect processes from each other. A user process must not be able to access memory or other resources which do not belong to it, and must not be able to execute privileged instructions. These restrictions should ensure that a user process can only access files and directories via system calls (sometimes called kernel calls). This is often achieved by providing system calls as "illegal" instructions which are trapped, security checked, and then interpreted by the kernel.

The second is that the operating system must make the trapping of system calls available to user processes. One user process must be able to run another in a tracing mode in which all system calls (or all relevant system calls) can be trapped just before and/or just after execution. `Brush` uses this to take backup copies of files and record information before allowing a system call to continue as normal, and in some cases to gather result information when a system call returns.

5.2.2 User requirements for recovery

There are many distinct classes of users working in interactive systems. Each user can perceive the concept of recovery differently and so expect it to be made clear. Accordingly, a recovery command must meet the requirements of the more casual class of user, as the more experienced user will be aware of the alternative ways that are available for a variety of situations[87].

What recovery commands are supposed to be is related to the system in which they are included. The following ideas highlight very general requirements for recovery, which may apply to all existing interactive systems.

- *Recovery commands must be simple:* An undesired recovery operation has already been carried out, and reversing the effects of the undo and redo commands could get unnecessarily awkward. Recovery commands should obviously be easier to use than the implicit recovery from a command: to undo the change of a file name should certainly be no harder than reentering the command that made the change.

- *The domain of recovery commands must be clear:* Some commands cannot be recovered. It is of great importance to specify which commands are undoable and which commands are redoable. Commands for which a recovery command is applicable should be made known to the user. For example, the user should be informed that a process which is killed while running an application cannot be restarted with the values when it is terminated.

- *Recovery commands must be general:* The user might make an incorrect observation of the current state and execute some commands that do not match the state. It would be exceedingly frustrating to find that the particular sequence of commands which changed it could not be undone. Recovery commands should cope with as many complicated situations as possible.

- *The scope of recovery must be unrestricted:* Some erroneous commands may not be noticed immediately and further commands may be issued after them. The existence of the subsequent commands should not affect the ability to recover. Besides, if these commands are relevant, then the reversal of the erroneous commands should be accomplished in a way that is not time-consuming. This can defeat the first requirement of basic simplicity, however.

- *Recovery must lead the system to consistent states:* The use of recovery commands is to remove commands from the state or to re-add commands to the state. It is essential that the state which is reached as a result of recovery is consistent. Both `undo` and `redo` should deal with the consistency of the target state.

- *Invocation should be easy;* It is desirable to achieve recovery with little effort. The user should be allowed to invoke recovery commands easily. Like many broadly applicable Unix commands (*help, man, type*), `undo` and `redo` deserve consistent names across all parts of interaction with the shell, which can be dedicated special keys on the keyboard.

From the above requirements, the most important one is to keep the state of the system consistent between recovery operations. The possibility of inconsistent states essentially arises from the dependency of commands on the state when they have been issued. This is also known as conflicts created by commands.

5.3 `Brush` Concepts

Before going into the detailed description of `brush`, it is important to introduce some relevant concepts that are used in the examination of user-oriented recovery. In this section, we define the concepts such as state, transactions, and conflicts, and examine their relationships with recovery. Furthermore, general expectations of a recovery mechanism are addressed, how command recovery is able to threaten the state consistency is discussed, and an extra facility that makes file recovery easier is given.

5.3.1 State

Interactive computer systems are used to create and modify persistent data structures called *objects*. The *state* of an object at a particular time is represented by the values of its components at that time, possibly including the content of each component. In order to view or change the state of an object, a computer system is instructed by a user issuing *commands*. The execution of a command causes the display of some portion of an object or a transformation of the object state. The effects of execution are assumed to depend only on the state existing when the command is executed. This means that when the current object state and the currently issued command are given, the resulting new state of the object is determined.

A *system state*, which refers to the file system state in this thesis, is defined by the values of all its components. A command performs operations on either the system itself as a whole, or one or more components of the system. Also, one and the same command may correspond to many different transformations, depending upon the context in which the command is executed. For simplicity, the state of a system can be viewed as a set of commands executed to transform the states of its components.

The components of the Unix file system (files, directories, etc.) are somewhat complex ones because, as well as their contents, they have some other values such as access permissions and modification times. From `brush`'s point of view, only the contents and access permissions of a file or directory make up the state of that file or directory. Other properties of files such as modification times are not considered at present. The current state of the file system consists of files and directories with the current contents and access permissions. The names of files and directories, including aliases, are also an important part of the state, since they correspond to the contents of a directory. Strictly speaking, when these values are modified, we assume that the state of the file system is transformed. The modification of other values does not affect the state.

In addition, shells have a state established by some environment variables that are specific to themselves, and provide built-in commands to modify their values. The user will also require `brush` to deal with such local states, allowing recovery from the commands.

Brush has its own `save` directory, e.g. stored as a subdirectory of a user's home directory, which holds a *history list*, a *transaction list*, and a collection of backup copies of files. These files contain information about the effects of commands on the file system, such as *transactions* (atomic file operations, see next section) to be performed and copies of files to be modified. This information is used by `brush` to determine how to recover from the effects of a command. The `save` directory is regarded as belonging to `brush`, and is not intended to be visible to the user, whether via brush or via any other shell. Even if it is held as a hidden directory (`.save`), the user cannot be allowed to perform any command on it. Thus it can be assumed that the `save` directory is not a part of the state.

`Brush` keeps track of previous states as well as the current state. The `save` directory contains enough information to reconstruct any previous state, or the state resulting from any sequence of undo/redo commands.

We will see in Section 5.3.4 that `brush` records the transactions which each command executes, as well as the command lines themselves. Roughly speaking, a *transaction* represents a single indivisible effect on the file store. A new sequence number is generated for each transaction, and these sequence numbers are used by brush as times. Each file version in the `save` directory is uniquely identified by the *time* at which it was created, i.e. the sequence number of the transaction which created it. Desired previous or alternative states are reconstructed just by moving files between the `save` directory and the visible filestore.

To illustrate what is stored in the `save` directory, and how different states are reconstructed, consider the following sequence of Unix commands, executed in an empty directory called `dir`:

```
1 edit text
2 cp text data
4 edit text
```

These commands are stored by `brush` in the history list. Each command is shown here preceded by the start time at which the command was executed. The sequence of transactions stored in the transaction list, corresponding to these commands, is:

```
1 Create text
2 Read text
3 Create data
4 Read text
5 Delete text
6 Create text
```

Each transaction is preceded by its time of execution, i.e. its sequence number. The first `edit` command creates a file called `text`. The `cp` command reads it and uses it to create another file. The second `edit` command reads in the first version of the `text` file, then replaces it by a second version. This replacement is represented as two transactions, one which deletes the old version and one which creates the new version.

As well as carrying out the commands, `brush` saves file versions in the save directory as necessary. After the transactions shown, the save directory contains just one file called 1, representing the first version of the `text` file (created at time 1). By now, the working directory `dir` contains the file `data` and the second version of the file `text`.

Suppose that the last command is undone by the user, so that `brush` needs to re-create the state just after the `cp` command. `Brush` can do this by executing the following commands "behind the scenes":

```
mv dir/text save/6
mv save/1 dir/text
```

In general, `brush` stores all the versions of files which have existed at any time, excluding those which are currently in the visible part of the filestore. These, together with the information in the history list and transaction list, allow any desired state to be reconstructed.

5.3.2 Commands

Commands that come with a computer system can have several different types. In the classification of commands with respect to undo support, we will use Yang's approach [98], which divides commands into the following types:

.primitive:	a basic instruction given to perform some operation
.compound:	several commands intended to run together
.macro:	a named command that invokes other commands
.meta:	a command that acts on commands for recovery

In our context primitive, compound and macro commands will be referred to as task-oriented commands or shortly commands, while meta commands refer to undo-like commands.

Operating systems supply many commands with which users can manipulate the file system or get information about the status of it (see Section 4.4.1). Recovery involves focusing on commands relating to the file system and keeping them under full control. For our new shell to manage the file system more effectively, it needs complete information about which files are affected by each command that is executed. It needs this information to undo the effects of commands by saving and restoring files appropriately.

Also, in order to execute commands safely, foreign programs in particular, it is desirable to be able to monitor all file accesses which a program attempts to make, and apply suitable security policies, beyond what can be done with conventional file permissions. Getting information about file access and controlling it is easy for system commands, which are implemented entirely within the shell. However, commands which involve running arbitrary application programs are more difficult. With conventional operating systems, programs may access the file system directly through the system libraries. The shell has no immediate knowledge of what the program is doing. We need a mechanism whereby programs can only access the file system via the shell, or at least with the knowledge of the shell, without having to re-implement the system libraries.

To be able to reverse the effects of a command, it is necessary to be able to detect and analyse those effects. Often, the permanent effects of a command consist of the files which it creates or alters. These can be monitored by our mechanism, copies can be taken of old versions of files before they are altered, and the effects of a command on the file system can be reversed by reverting to those old versions. However, commands can have a wide variety of different effects, other than their effects on files, some of which are difficult or impossible to reverse. A more detailed discussion of the effects of user commands has been given in Section 4.5.

Operating systems can support a variable set of commands or differently-named commands which do the same thing. We wish to be able to provide a recovery mechanism which works in a command-independent way. This is necessary to avoid having to recompile already-existing user commands. We also wish to be able to add new commands into the system, even after the mechanism is installed. Addition of new commands must not involve making any modification to the mechanism. That is, the pre-designed mechanism should reverse these new commands, too.

5.3.3 System Calls

In Unix and many similar operating systems, user programs make use of a system call interface to communicate their requests to the operating system kernel. When a user program is run, it interacts with the kernel by invoking a well-defined set of system calls, such as *open*, *read* and *close*. Since the main concern of our recovery mechanism is with the file system, we choose to deal with system calls associated with users' files and directories, which are briefly called file system calls throughout the thesis.

File system calls consist of issuing such operations as accessing existing files, creating new files and manipulating file attributes such as dates and permissions. These file system calls play an important role in incorporating the recovery mechanism into the operating system. The operating system can be prepared for recovery during the execution of system calls that a program performs. To accomplish this, it is adequate to stop system calls on entry to the kernel, or exit from the kernel, for a very short instant, so that the shell can gather the required information about what effects each user program has on the state. The fact that many contemporary operating systems support such an approach via tracing facilities makes it possible to achieve a user-level implementation.

Adopting such a design method, in fact, means that the mechanism does not need to be aware of which programs users are actually running. This enables the mechanism to function independently of which programs are currently installed and which new programs are incorporated into the system in the future. The mechanism is not independent of the operating system or its version, however, since these can differ in the set of system calls provided.

In discussing the importation of user-oriented recovery (or undo support) facilities into the Unix environment, it seems preferable that the relationship of commands to system calls is investigated. By system calls in this investigation, we mean user-callable library procedures. Note that there is a subtle difference between the user-callable library procedure, for example *read*, and the actual system call, READ, which is invoked by *read*. In many contexts they are effectively interchangeable, however.

5.3.4 Transactions

`Brush` need to be able to find out what effect each command has on the file system. Since all file operations are ultimately carried out via the kernel, all the effects of a command on the file system can be captured purely by monitoring its system calls. The effects of each system call on the file system can be further analysed in terms of a small number of simple (atomic) transactions. In this way, each command is regarded as consisting of a series of transactions and the transactions are determined from the system calls which the command makes.

Using system calls (*sysCall1*, *sysCall2*, etc.), Figure 5.1 explicitly shows the link between user commands and atomic transactions.

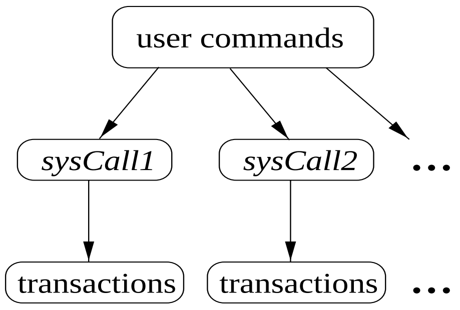

Figure 5.1: Splitting user commands into atomic transactions

Some system calls do not give rise to any transactions, because they do not have any relevant effects on the file system, or because their behaviour is outside the understanding of `brush`. Some system calls give rise to several transactions, because they are regarded as compound actions having several components. A transaction is not the same as a system call; it either succeeds completely or fails completely. Moreover, a system call acts on a file name, whereas a transaction explicitly acts on a file version. Transactions are indivisible atomic actions that are derived from system calls on the file system, e.g. a system call which overwrites a file corresponds to several atomic actions, as will be seen below. Also, `brush` adds extra actions to every relevant system call so that *every transaction can be reversed*.

Describing user commands in terms of a few simple transactions simplifies the user-oriented design of undo support in an operating system. Most file system calls can be viewed as performing combinations of three basic transactions: `Read`, `Create` and `Delete`, which correspond to reading the contents of a file, creating a new file and deleting an existing file, respectively. Rather than recording system calls themselves, our mechanism keeps track of the equivalent simplified transactions.

The `Read`, `Create` and `Delete` transactions do not directly represent all the operations that commands can perform on the file system. There are many other internal operations such as writing, appending, renaming and so on. However, these can usually be represented as a combination of the basic transactions. For example, the Unix command *mv file1 file2* typically uses the *rename* system call to pass a request to the kernel to rename *file1* as *file2*. This is recorded as two or three separate transactions. First, there is a `Delete` for *file1*. Second, if there is already a file called *file2*, then there is a `Delete` transaction for this as well. Finally, there is a `Create` transaction for the new version of *file2*.

Of these transactions, `Read` clearly doesn't cause any changes to the file system, and so no particular actions are needed to undo or redo a `Read` transaction. The reason for recording `Read` transactions is to document the fact that a command depends on the existence of a particular file. If the user attempts to undo the previous command which created that file, the undo is disallowed on the grounds that it would leave an inconsistent command sequence. `Read` transactions also affect the applicability of redo requests.

The transactions given above are related to system calls which operate on the contents of files. More transactions need to be defined for components such as directories. This issue will be examined in further detail in Chapter 6.

In `brush` *read* and *write* system calls are not monitored, and so do not give rise to transactions. Every transaction which alters or deletes a file has a corresponding behind-the-scenes `brush` action which saves a backup copy of the file. The reading or creation of a file is a single transaction. This ignores concurrency for the moment. When concurrency is considered (see Chapter 8), this should become two transactions; open for reading or writing, and close (for safe locking), which are required to prevent interference between commands and to unauthorise recovery commands on a currently open file.

5.3.5 Recovery Commands

Recovery represents the process of both undoing and redoing. In either case each object of the system that the recovery operation must be carried out on is brought into one of its prior states. Recovery commands are undo and redo commands that are used to start the process of recovery. Both `undo` and `redo` are meta commands, and so do not operate on themselves.

Recovery commands are only applicable to commands stored in the history list, which is visible to the user. The user can move to or specify any command in the list, and ask for that command to be undone or redone. A command is marked as *active* or *inactive*. Only active commands can be undone, and only inactive commands can be redone. Undo/redo commands are not stored in the history list, other than as changes to the active/inactive status of other commands.

The user model is that the current state corresponds to the sequence of active commands. If the system were returned to its initial state, and these commands were executed, the current state would be reconstructed. Active commands are associated with the components of prior states.

Undo/redo are selective. Any previous command can be chosen to undo or redo directly. In the context of sequential undo, this is equivalent to undoing all active commands back to the chosen command, then redoing all but the chosen command. If there are no conflicts between the chosen command and the intervening commands, the system results in the same state (see Section 5.3.6). This is a strong consistency property; any sequence of undo/redo operations which leads to the same sequence of active commands appearing in the state should lead to the same current file system.

The implementation will be such that `undo` and `redo` never lead to a command being re-executed. This is necessary to minimise the response time of `brush`. The effects of many commands are on a very limited number of files, and so it is more practical for recovery comands to deal with those effects directly. However, each command would act on the same file versions as it did in the actual history. To recover from such a command, a recovery command needs to know which versions of files it has been involved in.

The user's model of undo is deliberately linear, to keep it simple. A tree-structured history, which is adopted in the undo models such as US&R and direct selective undo (see Chapter 3), allows undo and redo to be executed under all possible circumstances, but is difficult for a user to grasp, and the extra functionality would not be worth the extra complexity. Instead `brush` provides a linear history structure on which recovery commands are allowed to operate selectively.

A history list usually contains many commands which depend on each other. As far as selective undo is concerned, it is quite common to encounter situations that conflict. This means that such commands cannot be undone or redone at least unless the others are handled first, as will be seen below. No matter whether command history is linear or tree-structured, conflicts can always occur.

5.3.6 Conflicts

It has been observed during the development of undo methods that undoing commands in an interactive session requires some kind of restrictions to prevent uncontrolled and undesired modifications on the state. When interacting with a system supporting recovery, a user often encounters such restrictions. They take place due to the user's arbitrary preference between execution and recovering of commands. A request for an undo or a redo leads to a *conflict* if it would result in an inconsistent state, i.e. one which violates the second invariance condition given in Section 5.1, which stated that:

> *If the active commands were to be executed on the initial state, each command would act on the same file versions as it did when it was originally executed.*

The consistency requirements involve `brush` detecting situations that create conflicts. As commands can make various modifications to a single file, it is very important for `brush` to keep track of relationships of commands with file versions. For conflicts to be discovered effectively, therefore, it is assumed that each modification made by commands to the same file causes a new version of the file to be created, and the modifying command is tied to the related file version. In order to ensure that recovery commands operate properly, user commands have to be undone or redone with correct versions of files. In this case, we say that the recovery process results in a *consistent* state.

The essential reason that conflicts can arise is the existence of *dependencies* between commands. A dependency arises between two commands when they both act on the same version of the same file. The second command relies on the first command having been executed, so the first command cannot be undone unless the second has already been undone. Even if the second command only reads from the file acted on by the first command, and makes no changes to the file system, allowing the first comand to be undone without the second would violate the above principle.

We illustrate the dependency problems which arise with examples, and then use these examples to develop the notion of conflict into a list of concrete dependency rules which must be checked by `brush` at each request for an undo or redo command, before allowing the command to be executed.

Suppose the following sequence of commands is executed:

```
edit fileA
mv fileA fileB
cp fileB fileC
edit fileA
```

The first `edit` command creates a file `fileA`. The `mv` command reads it and deletes it, while creating a new file `fileB` from the contents. The `cp` command reads `fileB` and uses it to create `fileC`, and the second `edit` command creates a new version of `fileA` from scratch.

Then suppose a request is given to undo the `mv` command. There is a conflict because if the undo were allowed, the file `fileB` would not exist at the time of the `cp` command. There is also another conflict, because the second `edit` command would act on an old version of `fileA` instead of creating a new version from scratch.

This leads to the two rules which `brush` must check before allowing an undo command. Suppose there is a command `C` some steps back in the history list. If a user invokes `undo` for `C`, `brush` must check whether or not:

(1) A subsequent active command acts on (i.e. reads, creates, or deletes) a file version which was created by `C`,

(2) A subsequent active command creates a new version of a file which was deleted by `C`.

Similar considerations lead to rules for `redo`, although now there are possible conflicts concerning previous commands, as well as subsequent commands. If the user attempts to redo a command `C`, `brush` must check whether or not:

(1) A subsequent active command acts on (i.e. reads, creates or deletes) a file version which was created by `C`,

(2) A subsequent active command creates a new version of a file which was deleted by `C`,

(3) The state at the time of executon of `C` contains an older version of a file which was created by `C` from scratch,

(4) The versions of files which were originally read or deleted by `C` do not exist in the current state at the time of executon of `C`.

These checking procedures meet the need to detect conflicts. When a conflict is encountered (i.e. one of the above procedures returns *true*), `C` cannot be undone or redone independently. More information and examples for conflicting commands will be given in Chapter 7.

However, it may be possible to rectify or prevent some conflicts. The difficulty of dealing with them depends on interactive environments where users work. In an operating system environment, a user can rectify a large amount of conflicts as a result of executing new commands. For example, an undo operation which is disabled because it causes the destruction of a file can be performed after renaming the file or by first undoing the command that creates this conflict.

What should `brush` do when a recovery command fails due to a conflict? There are several approaches to address the problem. The first and simplest approach is to tell the user about the conflict and to refuse the undo/redo. The second approach is to detect the conflicting commands and to ask the user to recover from them one by one before the chosen command is handled. The third approach is for `brush` to determine all the conflicting commands and to give the user the option to recover from the chosen command, along with all the conflicting commands. The provision of such an option is a design issue. On the other hand, if the only conflicting commands are ones which don't make changes (i.e. their transactions do not include file writes or deletes, just reads), then it might be desirable to undo/redo them automatically, along with the chosen command. This hasn't been implemented in the current version of `brush`.

5.3.7 Insertions

The requirements presented so far are that any previous state can be reached, and also any state described by a consistent subset of the commands in the history list. A further requirement (an `insert` facility) is that any desired collection of previously existing files can be recovered. This is especially important for a user to be able to recover past versions of a single file simultaneously. The facility is also useful in eliminating some kinds of conflicts.

The `insert` facility is generally used together with recovery commands; a file that is recovered by a recovery command is copied via a command to be inserted, to ensure that the copy stands even after subsequent recovery operations. Suppose a file `myfile`, version 1, takes a long time to construct. Then this is overwritten by version 2, which also takes a long time to construct:

```
edit myfile   (create version 1)
edit myfile   (overwrite with version 2)
```

When the error is realised, the user wants to see both versions at the same time in order to recover the effort. With the facilities described in the previous sections, this is not possible; either version can be accessed, but not at the same time. The `insert` facility sorts out the problem, allowing the user to do the following:

(1) Undo the second `edit` command,

(2) Insert a command that takes a copy of `myfile` (version 1),

(3) Redo the `edit` command.

where (1) brings the old version of `myfile` back, (2) copies this version to a new file (e.g. using a *cp* command), and (3) brings the new version of `myfile` back. Thus the user sees both versions currently. The point to insert the command into the history list is necessarily immediately before the last undone command (i.e. between two `edit` commands). `Brush` uses this policy for insertions.

With the `insert` facility, all past versions of a particular file can be recovered. However, the kind of the command to insert is very important. The inserted command must not affect the undoing or redoing of any later command. In the above example, for instance, if the user inserted a *mv* command, the new version of the file would not be able to be recovered because the `edit` command could not be redone. In Unix systems, the most convenient command for the insertion is the *cp* command.

5.4 An Automatic *Make* Facility

The motivation in tracing system calls is that it makes the shell environment more user-friendly. In particular, programs that are based upon the state of the file system can be easily rewritten with new functionalities. This section specifies the importance and ease of rewriting one of such program, the Unix *make* utility, in order to show the utilization of the method.

5.4.1 *Make* discussion

The *make* utility is a useful tool that is employed by many programmers during the development of sophisticated programs. It provides an elegant method for the compilation of programs which are made up of many components, written in any programming language. The method is fundamentally based upon the idea that source files must be kept consistent with corresponding object files. When a programmer modifies one of the source files of a program, *make* limits itself to recompiling the affected source file only. This offers a substantial decrease in the time of regenerating the program (see Chapter 2).

Before being able to use the *make* utility, a programmer must create a particular file, generally called *Makefile*. The file contains the information that is necessary to compile and link the files needed for an executable program. Using the *Makefile*, the *make* program also determines which source files need recompiling. The programmer may have to modify the *Makefile* during the development of a program as a result of incorporating new source files, or making the production of an object file dependent on new files. For large projects, the *Makefile* can become very complicated, dealing with sub-projects, libraries etc., and maintaining it and keeping it correct can become a burden.

5.4.2 Describing a new *make* utility

Our method of building a shell with undo support facilities allows the Unix *make* program to be enhanced with new functionalities, which is called `bmake` [48] throughout the thesis. The `bmake` utility is intended to function in an environment controlled by the shell. Unlike an ordinary shell, our shell keeps track of transactions caused by the interaction of user commands with the file system. In the shell these transactions are basically used for recovery purposes. However, they can also be used to meet the requirements of `bmake`, since they monitor which files are involved in the execution of a command, and what kind of operations (reading, writing, etc.) are made on these files.

In fact, `bmake` has an implementation which is based on command tracing. Each source file is compiled under the monitoring of `bmake` to allow it to discover which files the corresponding object file depends on. By examining the transactions since the last compilation of a source file, `bmake` can determine whether or not it is necessary to recompile the file.

Just as the shell works in a command independent way, `bmake` works in a compiler-independent way. The `bmake` utility has the ability to monitor any compiler (or other translator) supported by the system and keep track of changes made to the files that it includes in the compilation. To make use of `bmake`, programmers do not have to write or maintain a *Makefile*-like component. Instead, each command involved in the compilation must be executed "by hand" or from a shell script at least once, the first time. If the history list maintained by the shell includes multiple commands that create the same object file, `bmake` deals with the most recent one, and the transactions belonging to it.

The `bmake` program strives to provide all the facilities that come with the original `make` program. It also partly supports some facilities that are not in common use, such as associating a target with different sets of commands via a double-colon ("::"). Whichever set of commands is more recent is executed. The implementation saves the programmer from the effort of maintaining definitions, by providing the following features:

- there is no need to maintain a description file,
- the functionalities achieved are command independent,

- local variables, such as macros, are not needed,
- there are no restrictions on the generation of commands,
- dependency checking is done automatically,
- synchronization is handled automatically.

Among these features, the elimination of the description file requirement is the most distinctive feature of `bmake`. It saves programmers from the preparation process of such a file in writing application programs.

5.4.3 Using `bmake`

It is quite simple to use `bmake`. Its only requirement is that, after creating source files (possibly header files, too) required to build an entire program, a programmer specifies the relevant (compiling and linking) commmands once which lead to the production of the program. Afterwards, each time the files are edited, `bmake` can be invoked for updating the program. Since `brush` internally represents each new version of a file using a number (time), it is easy to check whether or not new versions of the files which make up the program are created.

`Bmake` takes the name of any generated file as an argument, which can be either an object file or an executable file. Given the case of using a *yacc* compiler, the argument can also be a source file. Furthermore, there is no restriction in the number of arguments to pass to `bmake`. Each file that is taken as arguments is handled in its order. As an example, suppose the programmer has previously constructed three source files, `myfileA.c`, `myfileB.c`, `myfileC.c`, and issued the following commands.

```
gcc -c myfileA.c
gcc -c myfileB.c
gcc -c myfileC.c
gcc -g -o myprog myfileA.o myfileB.o myfileC.o
```

These commands obviously produce three object files (`myfileA.o`, `myfileB.o`, `myfileC.o`) and an executable file (`myprog`). If the programmer makes modifications to any one of the source files at a later stage, e.g. `myfileB.c`, the `bmake` utility can be used as follows, in order to generate an up to date version of the executable file:

```
bmake myprog
```

where, discovering that `myfileB.c` has been modified, `bmake` executes the corresponding commands to recompile the source file and to produce the executable file, respectively. These executions are actually determined by using the history and transaction lists of `brush`. In summary, what `bmake` does is to

(1) Find the command which led to the current version of `myprog`,

(2) Discover the dependecies of `myprog`,

(3) Determine whether these dependecies are generated files (targets) or not,

(4) Repeat (1), (2), and (3) for each target, such as `myfileA.o`,

(5) Determine which files need reconstructing (`myfileB.o` and `myprog`),

(6) Issue the commands to bring the targets up to date.

Note that this algorithm is specific to the example given above. A more general algorithm can be found in Chapter 6.

Although `bmake` does not use an explicit description file, there are situations in which such a file is desirable. For example, if a software package is distributed, someone receiving it may want to re-make the package, even though they have no history of commands from which `bmake` can work out the dependencies and re-compiling commands to issue. To deal with this case, `bmake` is able to generate a description file which records the dependencies and re-compiling commands currently in effect. This file can then be included in a distribution and used with a conventional *make* command.

5.4.4 Compiler Tracing

A sample execution of a command that compiles one or more source files contains all information the `bmake` utility is meant to require. Instead of the use of the description file, the tracing of the sample execution is a new approach to gaining this information. From the shell's perspective, a compiler does nothing but read existing files (source files) and create new files (object files or executables). In the approach, transactions that represent all file accesses made by the compiler, thus play an important role in implementing `bmake`.

The approach to `bmake` is based on tracking the compilation of source files. As with ordinary commands, compilers work under the monitoring of the shell and file transactions that are performed are stored for the requirements of `bmake`. In order to describe information gathered with compiler tracing, we will use the C programming language and the *gcc* compiler.

Consider the following transaction list for two source files, `fileA.c` and `fileB.c` and for a programmer-defined header file, `fileA.h`.

```
1> Create fileA.c-1
2> Create fileA.h-2
3> Create fileB.c-3
```

Suppose that `fileA.c` includes the header file using the #include preprocessor directive, and that `fileB.c` does not. If the programmer attempts to produce the corresponding object files by typing in

```
gcc -g -c fileA.c fileB.c
```

the compiler performs the following *open* system calls of interest, consecutively.

```
open("fileA.c", O_RDONLY)
open("fileA.h", O_RDONLY)
open("fileA.o", O_RDWR|O_CREAT, 0666)
open("fileB.c", O_RDONLY)
open("fileB.o", O_RDWR|O_CREAT, 0666)
```

Then the transaction list is extended with additonal transactions:

```
4> Read fileA.c-1
5> Read fileA.h-2
6> Create fileA.o-6
7> Read fileB.c-3
8> Create fileB.o-8
```

Note that shared library header files such as `stdio.h` will not be of concern in the shell, as they do not belong to the current user.

Given the transaction list, it is possible to discover all the files (*dependencies*) which are used to produce an object file. However, for the cases where two or more source files are compiled together, there is no graceful way to find out which dependency a source file uses. In the above example, one can't simply say that `fileA.o` doesn't depend on `fileB.c`. The order in which dependencies are read and the corresponding object files are created can't be used to work this out, since the creation of `fileA.o` may not have been completed before `fileB.c` is opened for reading. As a result, each of the files read during the compilation tends to be a potential dependency for every object file created.

Now, consider what happens if the programmer re-edits `fileB.c`. Such a situation creates the following transactions.

```
9> Read fileB.c-3
10> Delete fileB.c-3
11> Create fileB.c-11
```

Of these transactions, `Delete` and `Create` explicitly show that `fileB.c` has been modified after its first compilation. In this case, both `fileA.c` and `fileB.c` must be recompiled and the recompilation must be repeated after any of the files they depend on are modified.

Chapter 6

Design

The development of interactive systems necessarily involves providing a wide range of commands to cope with many complex tasks required by users. The real world is so complicated that there may be no way to consider every command to be within the domain of an undo function. This makes recovery capabilities applicable for a reasonable set of commands, but not for all commands. Furthermore, in systems like Unix which allow addition of new commands that satisfy current requirements, it may be rather difficult to design an add-and-use mechanism of recovery. The undo function seems to be a tool which works on commands rather than tasks. Since commands have a divergent level of functions, undo should be as general a facility as possible.

The last chapter proposed a mental model of undo suppport for the user and examined some concepts associated with recovery. This chapter deals with the design of the mechanism that is implemented as part of a user shell, which will be called `brush`. It aims to present all recovery facilities at the user level, concentrating on the tracing of system calls. Such a presentation raises many design problems, which are described and resolved by giving an analysis of system calls of interest in terms of recovery. The chapter also introduces the design context of the `bmake` utility.

6.1 Introduction

Our system runs as a new user shell called `brush` (the Bristol undo shell) which can stand alone or act as a wrapper around an existing shell, adding whatever is required for the provision of recovery. Our approach allows a user to individually install and run the mechanism without making other users aware of it. The `brush` shell can run all existing commands and programs, and any new ones which are installed. It determines their characteristics purely by monitoring the system calls they make.

We aim to present users with a more friendly interactive environment in which they can selectively undo commands which have had unwanted effects. The fact that damaging commands can be reversed at any time makes interaction easy and free from frustration. From the users' perspective, this functionality also means that if they find that the system is in an inconsistent state, they can reverse their recent interactions to recover consistency.

This chapter not only focuses on building a recovery mechanism on Unix-based operating systems, but also demonstrates a general way to equip any operating system with an undo/redo facility. The primary advantages of our method are that the mechanism

- works entirely at the user level,
- has no need to recompile or relink existing commands,
- provides a program-independent recovery environment,
- imposes no modifications on the kernel,
- requires no changes to standard library interfaces,
- makes it possible to add new commands without fuss.

Like all usual recovery implementations, our mechanism needs to find out what changes each user command makes to the system state. Shells in common use never know which file operations commands pass on to the kernel to execute, since the commands make these requests with system calls. The `brush` shell must itself monitor file requests of commands. Fortunately, many Unix operating systems provide standard tracing facilities such as the `/proc` file system which is used by *truss*-like applications. With these facilities, system calls are intercepted and then resumed after the appropriate information is extracted. This approach is also adopted in [2, 50].

6.2 Extension Techniques

To be able to undo the effects of commands, we need to monitor and control the permanent changes that commands make. For example, if a command overwrites a file, we must detect this and take a copy of the old version first, so that we can restore it later if necessary. In addition, in order to execute foreign programs safely, it is desirable to apply suitable security policies to the changes which a program attempts to make, beyond what can be done with conventional user permissions.

A shell causes programs to run, and those programs access operating system primitives directly, rather than via the shell. What we need to do is to extend and adapt the operating system primitives so that they cooperate with the shell in managing resources, allowing the shell to monitor and control what is going on. Thus, in effect, we need to extend the operating system functionality, changing the way in which its primitives work.

Various methods for extending operating system functionality have been proposed, and are discussed and compared for example by Alexandrov et. al. [2], who use the extended functionality for adapting filing systems. Jones [50] uses extended functionality for implementing interposition agents, and Goldberg et. al. [40] use it for the secure execution of untrusted programs. All these example applications are relevant to our work.

One approach to extending the operating system is to change the kernel itself. The main problems with this are that it requires superuser privileges to make the changes, and that the risk of compromising the integrity or security of the system as a whole is very high.

A second approach is to change device drivers, libraries or managers associated with the operating system. For example, it is often easy to add new communication-based file system managers, as with NFS, and one of these could represent an alternative method of access to an existing file system. The main problems are that programs may need recompiling and/or relinking, and that the approach is unlikely to be sufficiently general.

The most successful approach appears to be to intercept system calls. In most operating systems, commands and programs use system calls as the lowest level interface to operating system facilities. The operating system may provide a way of intercepting the system calls made by programs. For example, many Unix based operating systems provide a device called `/proc`, described by Faulkner & Gomes [34], which allows one process to gain control over another at the system call level for debugging, tracing or other purposes. This facility can be used without needing special privileges, and is used in Unix commands such as `ps`, `truss` and `strace`.

This approach is very suitable for implementing undo, because system calls are the only way in which commands or programs can make permanent changes to the system. Moreover, no high-level knowledge about the intentions of any particular program is needed. This approach is used in the implementation of the recovery mechanism of `brush`, described in Chapter 7.

6.3 Description of the Design

The design of undo support in a developed system is supposed to focus on protecting existing system functionalities. An undo command should be able to handle every situation smoothly. This section discusses some issues connected to selectively undoing commands. It also gives a user-level consideration of undo support with which it is possible to control the user's activities.

6.3.1 Selective Undo and Redo

In systems with recovery facilities, these facilities have usually been designed from the outset, as with database rollback for example. It seems subtly difficult to incorporate an undo facility into an existing system in retrospect. However, it is relatively easy to incorporate the facility into a system which provides a means of manipulating the interface between the user and the filing system. In such systems, undo may not need anything other than user-level modifications. It is sufficient to change the interface or place an extra layer in between that manages recovery.

With our design, users can freely move back and forth over the history of commands, and pick an arbitrary command to undo or redo. This freedom means that commands sometimes make no sense to undo or redo, because they would lead to inconsistent states. The assessment of consistency is not left to the user's own observation of the state. The system itself assesses the current state by examining transactions that commands have conducted, and forbids inconsistent uses of undo or redo.

Undoing the last command in the history list, or undoing several commands in reverse order, always results in a consistent state. However, an attempt to undo a command other than the last one may destroy the effects of subsequent commands, e.g. by causing the unexpected removal of some requisite files. Similar shortcomings apply to redo as well; it may not be possible to carry out the changes needed to redo a given command. For example, the required files may be missing, or have different versions from the ones that existed when the command was initially executed, as a result of previously undone commands.

We call all these situations *conflicts*. Conflicts can damage the integrity of system data and leave some of the user's files version-inconsistent. Since, from the user's perspective, these results are not acceptable situations, we need to discover conflicts and to inhibit undo and redo from proceeding. It is the duty of the recovery manager to handle conflicts. In order to ensure that the effects of commands can be reversed in a conflict-free way, the manager must check to see if the right versions of files are in effect at the time of undoing or redoing.

Users may wish to be made aware of conflicts. Therefore the manager should also inform users about conflicts arising when a recovery command is issued, e.g. a message about which commands cause the conflict. For example, if an undo is requested for a particular command, some other command may need to be undone first. This other command may cause further conflicts, and so on. Thus it is helpful if the recovery manager describes the complete set of commands which would need to be undone together to yield a consistent state.

6.3.2 Monitoring file accesses

The key point in providing an undo facility is to keep all file accesses under complete control of the shell. Since file accesses on Unix systems are made by the use of system calls, this requirement can easily be met by a tracing approach. In this way we catch system calls by interposing a user-level layer, called the *Tracer*, between user programs and the operating system.

With the Tracer, it is desirable to deal at the user-level with all programs that a user can invoke. By attaching to a program, the Tracer intercepts selected system calls made by the program. In effect the Tracer can be thought of as if it is located on top of the system call interface. All that the shell must do to control file accesses is to ensure that every user program issues its file requests through this layer. This allows the Tracer to monitor all operating system tasks in the user's own environment and to inform the Recovery Manager about system calls that are made by tasks of interest.

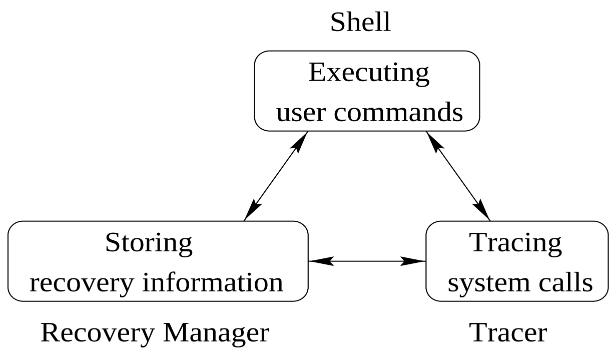

Figure 6.1: A design for user-level recovery

When the shell starts up the execution of a program, the Tracer initially connects to the process and tells the operating system which system calls to intercept. Whenever a system call of interest is issued, the operating system stops the subject process and notifies the Tracer. Then the Tracer passes the system call to the Recovery Manager to acquire appropriate recovery information, as shown in Figure 6.1. The Manager doesn't have to change the semantics of system calls to implement undo. Instead it analyses the related system call in conjunction with its arguments and focuses on system pathnames that it is likely to affect. If the pathname points to a file in the user's own address space a copy of the file is taken. Taking control back, the Tracer then resumes the system call.

For our recovery purposes, we trace system calls such as the *open*, *creat*, and *rename* calls, which operate on files. When the Tracer encounters a system call related to files, the Manager mainly keeps a sequence of the file operations that represent the system call and the name of files that they operate on. Most programs frequently used in Unix perform a few system calls that affect the user's files. For example, of the system calls the *mv* command issues, *rename* is the only system call the Tracer has to catch. For other user programs, there is also a small number of system calls for the Tracer to intercept.

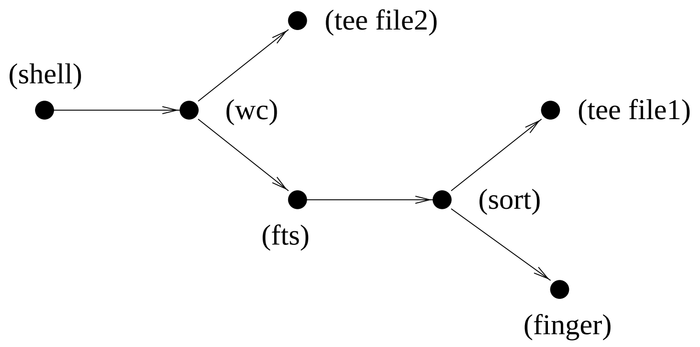

Figure 6.2: A potential hierarchy of processes to monitor

On the other hand, although ordinary user programs are only run by one process that the shell creates, Unix applications have the potential to dynamically spawn many processes. Even commands specified on one command-line can cause more than one process to be created. Consider, for example, a simple shell script below contained in a file named *fts*.

```
finger | tee file1 | sort
```

Then if the user submits the following compound command

```
fts | tee file2 | wc
```

the shell runs six distinct processes described in Figure 6.2. Two of these processes tend to affect the system state by overwriting existing files or creating new ones. Similarly, an application can create numerous processes (the *xfig* program spawns six child processes, for example). This means that the Tracer has to be responsible for monitoring all processes created by a program.

6.4 Recovery From System Calls

In principle, the consideration of recovery in Unix is associated with system calls, but not directly with user commands themselves. It is not important how complicated a user command is. This section analyses recovery-related aspects of system calls and examines various recovery techniques that perform key roles in improving the performance of a recovery mechanism.

6.4.1 Characteristics of system calls

In the design of `brush`, recovery from a particular command is considered as the reversal of the system calls that belong to that command. This makes it necessary to examine the characteristics of system calls.

The basic functional requirement of undo support is *reversibility* (see Section 3.5.2). Reversibility is the property that the state of a system can be reversed to a previously existing state after execution of a sequence of task-oriented commands. The existing application programs involves different implementation techniques to reverse the effects of a command. Some directly remove the changes the command creates, while others perform new commands which result in the removal of the changes. These techniques need adapting for system calls, which will be explained in the next section.

An undo command basically consists of implementing the *inverse* of the system calls performed by a command that has previously been executed (see Chapter 5). Thus, system calls must have some notion of reversibility for undoing them to make sense, e.g. in a shared system, the generation of the inverse may be affected by the state of the working environment which changes as a result of the commands of remote users. To ensure that a system generates the inverse of each system call, adequate information about that call must be stored somewhere in the system [72] and also enough information to counteract data loss of the system it causes must be kept.

How to reverse the effects of a system call largely depends upon the nature of the call. This fact has triggered various considerations on system calls which form the domain of the undo mechanism of `brush`. As a result, there are three kinds of reversible system calls [99].

- self-invertible
- invertible
- destructive

A *self-invertible* system call is a call whose effects can be reversed by executing itself. It is a little complicated to fix which system calls belong to this group. In Unix, whether a system call is self-invertible or not is determined without considering changes on the arguments that the call takes. One example is the *chmod* system call. The reversal of this call is just a matter of executing the call once again by changing its arguments to proper values.

An *invertible* system call is a call whose inverse is equivalent to the effects of one or more system calls other than itself. In Unix, the system calls *mkdir* and *rmdir* are invertible calls. They can reverse each other's effects reciprocally. The system call *symlink* is also invertible because its effects can be reversed by executing the system call *unlink*.

A *destructive* system call is a call that destroys some data on the system. The only way to reverse the effects of such system calls is to retrieve corresponding state components. One example is the Unix system call *unlink* which removes a specified file. As another example, the system call $link(file_1, file_2)$ destroys $file_2$'s contents if it already exists. Either system call can be inverted by bringing back the destroyed file with the original contents.

In fact, since a reverse operation destroys the current state of the system and then reverts it to the prior state, the inverse of a system call may generally depend on the state of the system on which it is executed. The less the destruction of system calls on the state the easier the recovery mechanism for those calls can be implemented. However, as mentioned above, the reversibility of a system call can be decided by the system itself in view of the fact that the system can support a system call that may be able to stand for the inverse of another one. In this manner, without requiring existence of the supplemen-

Operation	Saving
deleting f_1 f_2 ...	contents of f_1 f_2 ...
moving f_1 to f_2	contents of f_1 and f_2 if it exists
updating f_1	old contents of f_1
creating f_1	nothing
appending f_1	contents of f_1
changing attributes of f_1	old attributes of f_1

Table 6.1: Preparations for recovery from system calls

tary structures, the system can undo a system call by carrying out its opposite counterpart alone.

A list of some tasks a user is more likely to perform in an operating system environment is shown in Table 6.1. It also contains the preparations that the mechanism should make for recovery from some system calls. In this way, the saving operations that are intended for selective undoing represent the tasks that provide required information for undoing. For a task of deleting a file, the file name and its contents have to be saved before executing the task. In general, the command executing the task is included into the interactive history along with its arguments, while the system calls issued and file contents affected are stored in another safe place.

6.4.2 Efficiency

In the previous section, we have mentioned that reversible system calls (or commands have widely divergent levels of reversibility (undoability). The same is true for the techniques which are to be employed to reverse the commands. The usability of these techniques is a matter of efficiency. It may be possible for only one technique to cope with undoing of all commands. But this may impose a significant overhead (runtime memory requirement, disk storage, response time, etc.) on the system the undo/redo facility is included in. Consequently, it makes sense to take efficiency into consideration whenever building the facility. Efficiency relates to the optimum use of the system resources at the time of interaction and can be achieved through a collection of recovery techniques that minimizes the amount of space and time occupied during saving and undoing.

Recovery techniques are largely dependent on the nature of the commands and thus their effects on the state. To efficiently implement the inverse operations of the commands in the model, a couple of existing recovery techniques [4, 98] may be used. The primary techniques are:

(1) *Complete rerun:* The initial state of the system and a history of all executed commands are saved. To return the system to the state prior to a specific command, the system is first restored to its initial state and then the commands from the beginning of the history to before the relevant command are redone. Although it seems quite simple, the space necessary to save the initial state can be very large. Also, since execution times are proportional to the length of the history its response time can become very long. For instance, for the file system commands of Unix, all files and directories from the outset are required to be saved along with entities such as their contents and permission information. Thus, its efficiency is quite low.

(2) *Full checkpoint:* The whole state of the system is saved after executing each command. The copy of the state is called a full checkpoint. Recovery is accomplished by restoring the desired state without rerunning any command. This technique uses more space and time than the complete rerun technique does and is therefore inefficient. Instead, an interval checkpoint technique can be used which saves the state infrequently and keeps a history of all issued commands. In this case, an intermediate state can be reached by reversing the system to the most recent state saved before that state and then re-executing the commands that have been previously submitted between two states. This could also obviously be expensive in both time and space.

(3) *Inverse function:* The notion is that the system may support the commands that correspond to the inverses of issued commands. A command with an inverse function does not require state information to be saved. As long as the inverse of the command is known, what needs to be maintained

is only the history of interaction which stores that command. Reversing a command is done by executing the inverse of the command. The Unix commands such as *mkdir* and *rmdir* are invertible and can be undone in such an approach. However, not many commands have inverses, because they render the system irrecoverable unless state information is stored. To reverse the effects of these commands, past states must be preserved by means of other recovery techniques.

(4) *Partial checkpoint:* This technique, which is often used, is based on the fact that execution of an individual command affects on only a few components of the system. So, only the states of the components to be changed by the execution need saving. For instance, it can be employed to reverse destructive commands efficiently (see Section 6.4.1). When a command in these groups is executed, the system saves the state variables which are changed or the data which is destroyed by its execution. Recovery from the command is achieved by restoring the destroyed data. An advantage of the technique is that it is capable of undoing a command without requiring the corresponding inverse commands.

(5) *Deferred execution* The most frequently undone commands are supposed to be in a sequence of the most recent commands. Deferred execution involves delaying the execution of commands for as long as possible while making the system seem as if they had been executed. That is, a system need display the effects of commands to users without doing the actual executions and preserve this pretence for a restricted sequence of them. Recovery is accomplished by removing the relevant command from among the postponed commands. The deferred execution technique is not applicable for long sequences of commands and also especially commands that are issued to complete real-time processing of data. Nonetheless, it can be useful for the *kill*-like Unix commands that have substantial effects on running processes.

These techniques can be used to recover from a sequence of commands of any system. They also apply to recovery from Unix system calls. In other words, the effects of reversible system calls that a command issues can be undone by using one of the above recovery techniques.

In an undo mechanism, two (or more) separate techniques can concern themselves with the undoing of a single command (or system call) regardless of efficiency. In Unix, for example, recovery from a *mv* command (or *rename* system call) can be implemented with the same result by either the partial checkpoint technique (restoring the relevant copy back) or the inverse function (renaming or moving the relevant file again). Both of the techniques are effective for sequential undo of the command. However, the partial checkpoint is the only efficient technique for *selective* undo of the command because a *rm* command (or *unlink* system call) following that command (or call) makes implementing the inverse function impossible.

6.5 Brush Design

`Brush` is developed by using a simple design procedure. The simplicity of the design is actually to accomplish both a smooth understandability and a straightforward implementation of the shell. This section concentrates on only the design procedure. First the components that construct `brush` are described and the issues that pertain to its design are examined. Then the facilities `brush` presents as a shell to the user are enumerated.

6.5.1 The `Brush` architecture

`Brush` is a user-level process that connects to other user-level processes and enables them to make all file system requests through itself. Once connected to a subject process, it intercepts system calls (that operate on the file system) and stores changes of state. The program is unaware of the existence of the `brush`, since `brush`'s interception never affects its normal execution. It can reach all system resources without being confronted with any restriction.

`Brush` is designed in three modules: the `Shell`, the `Tracer` and the `Recovery Manager` (Figure 6.3). The `Shell`, as one guesses, is a command interpreter. The `Tracer` is responsible for intercepting system calls and forwarding them to the `Recovery Manager` module. The `Recovery Manager` module functions as a file version controller and consists of four layers: the `File Checker` layer which distinguishes an individual user's own files from others, the `Filestore` layer which has control over the history and transaction lists and stores a copy of each file version, the `Conflict Detector` layer which analyses undoability or redoability of a command, and the `Undo/Redo` layer which deals with undoing or redoing of the command. Note that these layers are not indicated in the figure.

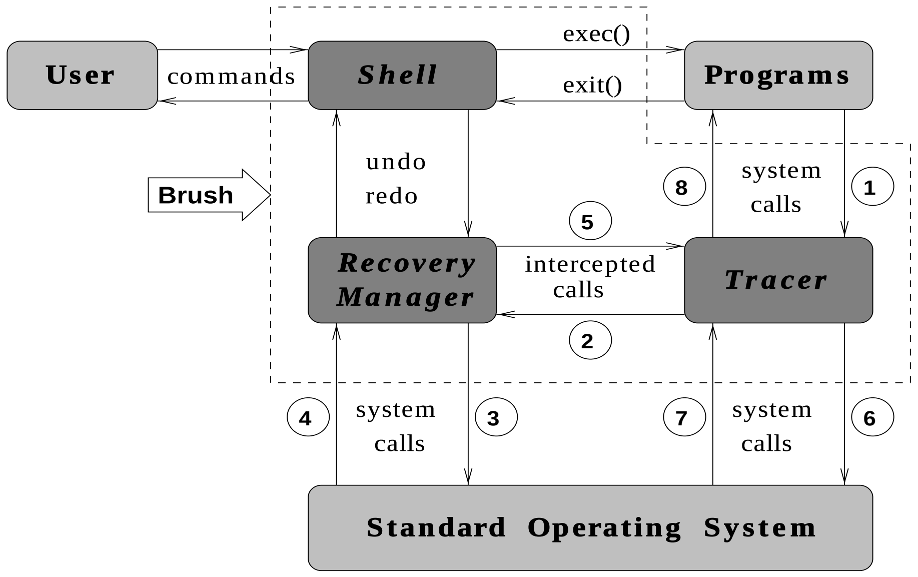

Figure 6.3: General architecture of `brush`.

Figure 6.3 also shows the steps involved in servicing a file system request made by a program. When the program issues a system call (1), it can go directly to the kernel through the `Tracer` (6) or, if it is file-related, get intercepted by the `Tracer`. For intercepted calls, the `Manager` takes control (2) and determines whether the system call operates on a file owned by the user, possibly using kernel services (3 and 4). If the user is not the owner of the file, the `Manager` does nothing. Otherwise, if the user is the owner and the system call is to affect the file, the `Manager` creates a new copy of the file in the filestore and lets the request proceed to the kernel (5 and 6). After the request is serviced in the kernel (7), the result is returned to the program (8). The return from the system call may also be intercepted. In this case, `brush` needs only the `Tracer` to handle the system call, as the `Manager` has nothing to do with the returned value.

Besides its simultaneous operation with the program, the `Recovery Manager` interprets and copes with `undo` and `redo` commands dispatched by the `Shell`. During recovery, it has full control of the execution and uses system utilities, possibly performing new system calls. Generally, two layers of the `Manager` focus on the preparation of the system for recovery, while the other two layers carry out the task of selectively undoing and redoing user commands in a secure way.

6.5.2 Design issues

Our goal is to design a shell named `brush` that provides recovery facilities. Since we concentrate on a user-level design of a recovery mechanism, `brush` can't cope with system errors such as system failure and media failure and so can't protect the user's files against these destructive errors. With the design, we allow the user to reverse the effects of commands which the shell mediates.

One of our design goals was to store all recovery information in regular files. Keeping the information in data structures in the shell has some drawbacks; the most important one is that, if the shell process unexpectedly terminates, whatever is locally stored gets lost. We want to be able to re-establish the interaction, even after the shell crashes. A newly created shell must acquire the information the previous shell kept just before it terminated.

To achieve this, the shell maintains two files containing the history list and the transaction sequence. There is a remarkable difference between the user's point of view and the shell's point of view in visualizing operating system tasks. Users are not interested in what the system uses internally to achieve a task, but only in the state into which the task will bring the system. Therefore, it is required that the design keeps user-oriented tasks (commands) themselves as a history list for users and a set of the kernel-centred operations (system calls) executed by them as a transaction sequence for recovery purposes.

The design doesn't impose any restriction on the use of existing programs because it is based completely on monitoring the execution of each program. Users can run any program they wish. Our shell focuses on the individual behaviour of programs, attaching itself to them to intercept a particular set of system calls. It not only deals with ones which affect files, but also ones associated with process management.

We want to intercept as few system calls as possible for reasons of performance. Tracing a user program potentially adds a non-trivial amount of overhead if too many system calls are intercepted. In fact, not all the system calls that are likely to affect files have to be traced. A program, for example, can perform many `read` or `write` system calls using the file descriptor returned by an `open` system call. Which file is affected, if any, can be discovered by analysing the arguments of the `open` call. The `read` or `write` calls do not need to be intercepted. This saves a lot of potential overhead, and contributes significantly to the performance of the system.

Ordinary Unix shells allow users to construct a sequence of commands in a script file or on the command-line in some way. In particular, using process management commands such as `&`, ;, and `|`, they can integrate several commands into a single one. Running such compound commands spawns many child processes. Similarly, at run time every child process can create new subprocesses and so on. As a result, we have to deal with all child processes created both statically and dynamically.

As usual with shells, we must ensure that `brush` protects itself against interruptions (e.g. triggered by Control/C), otherwise `brush` could leave the system in an irrecoverable state.

A program may issue many `open` system calls. Many of these are not of concern to the Recovery Manager, particularly when they involve reading central installation files or shared library files. The manager must be able to distinguish these files from ones which belong to the user. It is not possible to tell whether a file resides in the user's file space just by looking at the pathname passed to the `open` call. This is because there are features of the file system such as relative pathnames, symbolic links and hard links which can be used to create aliases. Instead the Manager uses device and file identity numbers to work out where a file is physically held.

Previous versions of files are saved, and named according to their time of creation. In addition, whenever a command involves a file, `brush` needs to record which version of the file was involved. This is because the relevant version is not necessarily the most recent one. For example, suppose a command (c_1)

which creates a file `f` is undone, and a new command (c_2) is executed. If (c_2) involves `f`, the relevant version is not the one created by command (c_1), but some previous version.

Another of our design goals is to retain the behaviour the system had before the undo facility was added, so that users of the original system can continue to use the system without difficulty. In `brush` we trace system calls at the user-level just to gain information about what kind of operations to perform on the file system. Unlike the method adopted in Ufo [2], there is no need to change the semantics of system calls. As they are stopped on entry to or exit from the kernel and then resumed, they run in the usual way. Accordingly, commands lose nothing of their original behaviour.

Before executing undo and redo, we employ a conflict-detecting algorithm to determine whether to authorise them or not. As a conflict arises from a dependency between commands, we have to analyse past commands of the interaction. In discovering conflicts, it is not necessary to analyse the current state and the current versions of files. Instead `brush` looks only at the transaction list. In particular, the analysis has to cover the transactions of the commands that have been run on the file system. Undo only requires that the algorithm considers commands which have been issued after the command for which undo is requested. Redo involves taking all effective commands in the history list into consideration. For undo and redo to be authorised, the checking is concerned with the consistency of versions of files which have been saved. To redo a command, there is no requirement for the command to be run again. Instead, the relevant versions of the files affected are made current again, e.g. by copying them into the current directory.

6.5.3 `Brush` facilities

As seen above, building an environment with recovery facilities has raised many issues that must be dealt with, and so some additional requirements must be considered. When needed, `brush` characterizes an issued command by way of aspects of its system calls. With this characterization, it may be impossible to determine whether a command is reversible or irreversible. Instead we use the words *effectual* and *ineffectual* to refer to command discrimination. A command that performs a system call of interest on the file system counts as an effectual one.

As well as well-known features of modern shells, such as resubmission and command-line editing (see Chapter 2), `brush` presents the following facilities:

(1) *Undoing/Redoing of commands*: The undo and redo commands are the most crucial provision of `brush`. Using them, the user can keep interactions under complete control as a result of eliminating unexpected situations. Since they reverse each other's effects, i.e. only one of them can be applicable to a command at a time, both `undo` and `redo` can be controlled by a single key.

(2) *Discovery of conflicts between commands*: It is very natural that recovery operations cause clashes with the current state of the file system. On the user's behalf, `brush` analyses the state and a related recovery operation through the command transactions, and determine if it is safe to allow recovery or not. This covers the situations where recovery from commands needs the relevant versions of their input or output files.

(3) *Manipulation of command history*: The user can tell `brush` how to arrange command history. The arrangement is made according to the evaluation of the behaviour of commands, which distinguishes between effects and executability of them. There are three kinds of commands that are likely to make up the history list; erroneously typed commands that are not executable, (effectual) commands that affect the file system, and (ineffectual) commands that do not. With the user's request, `brush` can leave any one or two of these classifications out. This allows creating an elegant form of the history list. Note that undo/redo/resubmit facilities apply only to the stored commands.

(4) *Insertion of commands into the history*: In `brush`, the user can correct some conflicts either by undoing/redoing the command that creates the conflict or by executing extra commands. A command that is issued to correct a conflict is placed into an appropriate position in the history list. Extra commands appear in the interactive history as if they were executed at the previous stage of interaction.

(5) *Automatic reversal of erroneous executions*: Command executions can abort or terminate unsuccessfully because of situations such as hardware failure and faulty interactions. It is quite easy to avoid the effects of command abortions by customizing `brush`. If a command which fails during its execution has produced file system transactions, `brush` can reverse those transactions automatically. In this case, the command is stored as having been undone (inactive) in the history list.

(6) *Observation of the kernel activities*: The user may wish to see how a shell cooperates with the kernel in terms of system call-based requests of processes. A separate facility is provided to observe the kernel's role in executing commands. For the last command executed, `brush` stores a record restricted to interactions of the kernel with the user's own file system. This record consists of the original names of system calls and their arguments in which the command is involved.

6.6 System Calls and Transactions

System calls associated with undo support mainly make simple changes to the user file system. In this section, these changes are classified in terms of three components of the system, namely files, directories, and the working directory. A single system call can deal with two separate components, which need to be recorded as different changes.

6.6.1 Basic transactions

Our principal aim was to simplify the design of `brush` as much as possible. Therefore, we split interactions of system calls with the operating system kernel into a few basic transactions, which are actually capable of reflecting their original effects on the file system. In this way, the interaction of each system call is represented by one or more transactions. Rather than the system calls themselves, the `brush` shell only records their corresponding transactions. Recovery operations cope with the reversal of these transactions. Handling transactions makes the determination of potential conflicts between commands much easier.

File transactions	Directory transactions	Other transactions
Read	SLink	Path
Delete	Remove	
Create	Make	
FPermit	DPermit	

Table 6.2: Grouping the file system transactions

System calls that operate on files perform four kinds of transactions: `Read`, `Delete`, `Create`, and `FPermit`, which individually carry out only one atomic operation. One `Read` represents the reading of an existing file. One `Delete` represents the removal of an existing file from the file system. One `Create` represents the creation of a file from scratch. One `FPermit` represents the modification of access permissions of a file. Any other file operation such as writing to or renaming a file are represented by a combination of these transactions.

System calls that operate on directories are fairly similar to those which operate on files. In representing directory operations, four separate transactions are used effectively: `SLink`, `Remove`, `Make`, and `DPermit`. In the specified order, the transactions denote the symbolic linking, deletion, creation, and access permissions change of a particular directory.

Another transaction, `Path`, is required to keep track of the working directory during an interactive session. One `Path` denotes a change on the working directory.

The transactions given in Table 6.2 allow `brush` to record all kinds of changes caused by system calls on the file system. But they are not all that may be required for recovery and so some additional information is stored, such as version number of files that system calls use. Note, below, that it is extremely important to record the transactions of the system calls in the right order, because an `undo` command needs to handle them in reverse order. To find out if a system call runs successfully, we check for the existence of access permissions of files and directories that it is going to affect. In the following figures, the checking of their existence only is indicated. More detailed information can be found in Chapter 7.

6.6.2 *chmod* and *fchmod* system calls

The *chmod* and *fchmod* system calls change the access permission mode of a file, the former using the file's name and the latter using a file descriptor that points to the file. The permission mode is composed of various combinations of `read`, `write`, and `execute` permissions which refer to permissions of the user, the user-group and all others. To represent these calls, a special transaction, `FPermit`, is assigned. Thus one `FPermit` must be recorded for each *chmod* or *fchmod*, recording the new permission, together with the previous permission to allow the undoing and redoing of the calls.

Both calls can also change access permissions of a directory. The change on directory permissions causes larger effects than on files. This allows the user to be able to lock a directory, making all the files and subdirectories in that directory inaccessible by other users. When operating on directories, the effects of *chmod* and *fchmod* are represented by one `DPermit` each (see Figure 6.4).

```
int chmod (char *pathname, mode_t newmode)
int fchmod (int descriptor, mode_t newmode)
    if pathname or descriptor belongs to a file,
        FPermit pathname oldmode newmode
    if pathname or descriptor belongs to a directory,
        DPermit pathname oldmode newmode
```

Figure 6.4: *chmod* and *fchmod* transactions

The access permissions have great control over the executability of some commands. The absence of a particular permission can put restrictions on operations on the contents of files or directories. For example, a directory that has no `execute` permission does not allow files to be accessed or removed. Similarly, a file that has no `read` permission cannot be read. These situations can substantially affect the recording of transactions for recovery operations and therefore need detecting. From now on, in the analysis of system calls, we assume that they have all required permissions on files and directories.

There are some complications created by the *chmod* and *fchmod* system calls. One potential complication occurs when `undo` operates on a file which was previously made readable by the call. Manipulating access control permissions of such a file by using `undo` may be inconsistent with later file operations. In this case, the `undo` must be disallowed. The situation can be illustrated by the following example:

```
chmod +r fileA
edit fileA
```

where, as one can guess, `chmod` is a command that invokes the *chmod* system call. It ensures that the second command runs successfully. Because of this dependency between the commands, the `chmod` command cannot be undone independently of the `edit` command. The new permission mode recorded for the call helps the shell sort out situations like this.

6.6.3 *open* system call

The *open* system call opens a file for operations such as reading, writing, and appending, and returns a file descriptor for that file. Using the file descriptor, other system calls perform desired operations on the file. The type of operations is determined by a particular flag in the *open* system call, which specifies file access modes.

There are many file access modes (flags) which are likely to affect an existing file: O_WRONLY, O_RDWR, O_APPEND, O_CREAT, and O_TRUNC. Of these flags, O_TRUNC immediately causes truncation of the file to zero, while O_WRONLY and O_RDWR allow subsequent system calls to modify the contents of the file. Therefore, when a flag is trapped which consists of or is composed of any one of these access modes, one `Delete` and one `Create` must be recorded if the file exists. The creation of a file from scratch can be make by O_CREAT only, in which case one `Create` must be recorded. The other access mode, O_APPEND, has no effect on the file on its own. For instance, O_APPEND does not allow the file to be modified unless it is used together with O_WRONLY or O_RDWR, i.e. O_WRONLY | O_APPEND and O_RDWR | O_APPEND.

```
int open (char *pathname, int flag, mode_t mode)
    if O_CREAT and O_EXCL are not set or pathname does not exist,
        if any one of O_RDONLY, O_RDWR, and O_EXCL is set,
            if pathname exists,
                Read pathname
        if any one of O_WRONLY, O_RDWR, and O_TRUNC is set,
            if pathname exists,
                Delete pathname
                Create pathname
        if O_CREAT is set,
            if pathname does not exist,
                Create pathname
```

Figure 6.5: *open* transactions

Additionally, one `Read` must be recorded for O_RDONLY, O_RDWR, or O_EXCL. The latter is generally used together with O_CREAT. A process executing *open*() with O_EXCL and O_CREAT set checks whether the file exists or not and *open*() fails if it does. This makes it necessary to record one `Read` in the case of the existence of the file (see Figure 6.5).

The *open* system call can be also used to open directories. But `brush` does not require directory opening operations to be stored, because there is no system call which modifies the structure of a directory by using the file descriptor that points to that directory. The only exception to this is probably the *fchdir* system call that changes the current working directory to a directory pointed to by the file descriptor, which will be examined later.

6.6.4 *creat* system call

The *creat* system call deals with the creation of a new ordinary file from scratch, regardless of the existence of the file. This means that an existing file will be truncated to zero. The truncation of the file involves recording one `Delete` and one `Create`. For a non-existing file, one `Create` can represent the call (see Figure 6.6).

```
int creat (char *pathname, mode_t mode)
    if pathname exists,
        Delete pathname
    Create pathname
```

Figure 6.6: *creat* transactions

In fact, the effect of *creat*() on the file system can be characterized by a special form of *open*() as follows:

open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)

This characterization ensures that the same transactions are produced and recorded, as will be seen in Figure 6.5.

6.6.5 *rename* system call

The *rename* system call gives a file a new name. While doing this, it ignores the existence of the target file. One `Delete` and one `Create` can represent the changes that the call makes to an already-existing target file (see Figure 6.7).

The call does not influence any attributes of the old file including its contents, changing its name or pathname only. However, the file has to be considered as being modified for the purposes of selective undo and so another `Delete` must be also recorded for this file. Unlike sequential undo, selective undo requires a copy of the old file to be saved whenever the call is executed.

When the *rename* system call is used on directories, its effects can be represented by two transactions, `Remove` and `Make`. The call normally fails only if either the source directory doesn't exist or the target directory is not empty[1]. Note that this is different from the behaviour of the *mv* command that places the source directory into the target directory as a subdirectory (invoking *rename* with appropriate parameters). In other cases, the source directory is removed, for which one `Remove` must be stored. Also, according to the status of the target

[1]It is assumed that a directory with only entries for itself (.) and its parent directory (..) is "empty".

```
int rename (char *oldpathname, char *newpathname)
    if oldpathname and newpathname are files,
        if oldpathname exists,
            Delete oldpathname
            if newpathname exists,
                Delete newpathname
            Create newpathname
    if oldpathname and newpathname are directories,
        if oldpathname exists,
            if newpathname doesn't exist,
                Remove oldpathname
                Make newpathname
            if newpathname is empty,
                Remove oldpathname
                Remove newpathname
                Make newpathname
```

Figure 6.7: *rename* transactions

directory, one `Remove` and one `Make` must be recorded, in order to reflect the effects of the call on it.

6.6.6 *unlink* system call

The *unlink* system call removes a file from the file system. It is the easiest call to represent with the simple transactions. Its effect can be depicted by only one `Delete` (see Figure 6.8).

```
int unlink (char *pathname)
    if pathname exists,
        Delete newpathname
```

Figure 6.8: *unlink* transactions

6.6.7 *symlink* system call

The `symlink` system call creates a symbolic link to a file or a directory. The creation of the symbolic link does not depend strictly on the existence of the component to which it is to point. Whether the component exists or not, the symbolic link is made, which can be represented by one `Create`.

```
int symlink (char *pathname, char *newpathname)
    if newpathname does not exist,
        if pathname is a file,
            Read pathname
        if pathname is a directory,
            SLink pathname
        Create newpathname
```

Figure 6.9: *symlink* transactions

On the other hand, most file operations performed on a symbolic link use the linked-to file. In order to record such a situation, there is no need to define any extra transactions. A record of one `Read` is adequate to show the dependency on that file (see Figure 6.9). But, if the symbolic link is made to an existing directory, an extra transaction, `SLink` is needed for a record of that directory.

6.6.8 *link* system call

The *link* system call is quite similar to *symlink*(), except that it creates a new link (directory entry) for an existing file. The dependency on the linked-to file and the creation of the link are recorded with one `Read` and one `Create`, respectively.

Like *rename*(), the call can destroy an existing target file. The track of this situation can be kept by one `Delete` on the target file.

It is possible that the *link* system call operates on directories as well. However, since the kernel allows only a superuser to *link* directories, such an operation of the call will be out of concern.

```
int link (char *pathname, char *newpathname)
    if pathname exists,
        Read pathname
        if newpathname exists,
            Delete newpathname
        Create newpathname
```

Figure 6.10: *link* transactions

6.6.9 *mkdir* system call

The *mkdir* system call creates a new directory. One `Make` must be recorded to represent its effects on the file system (see Figure 6.11). It cannot operate on an existing directory.

```
int mkdir (char *pathname, mode_t mode)
    if pathname doesn't exist,
        Make pathname
```

Figure 6.11: *mkdir* transactions

6.6.10 *rmdir* system call

The *rmdir* system call removes an existing directory from the file system. For the call to succeed, the directory must be empty. One `Remove` can represent its effects (see Figure 6.11).

```
int rmdir (char *pathname)
    if pathname is empty,
        Remove pathname
```

Figure 6.12: *rmdir* transactions

6.6.11 *chdir* and *fchdir* system calls

The *chdir* and *fchdir* system calls change the current working directory. They are only two system calls in Unix that help users and processes change their working directories. The *chdir* system call uses the pathname of the target directory, while the *fchdir* system call uses the file descriptor returned by an *open* system call.

```
int chdir (char *pathname)
int fchdir (int descriptor)
    if pathname or descriptor points to a directory
        Path pathname
```

Figure 6.13: *chdir* and *fchdir* transactions

Both calls are represented by one `Path`. Using the *fstat* system call, it can be easily determined whether the descriptor points to a directory or not. This eliminates the need to keep track of file descriptors returned by *open*s that run on directories.

6.6.12 Other system calls

Some Unix systems supports more file-related system calls, including *open64* and *creat64* whose syntax is identical to *open* and *creat*. These calls are usually performed on large files and are not a part of the POSIX standard. `Brush` intercepts them and enables recovery from them as well.

There are some other system calls which can affect the file system, such as *mknod*, which creates files or directories, and `chown`, which changes owner and group of files. The Unix system does not allow ordinary users to invoke these system calls through their programs. Only a superuser can call them. Since `brush` is designed for an individual user, we regard such system calls as beyond the scope of this thesis.

6.7 A Design for the *Bmake* Utility

This section probes issues concerning the design of the `bmake` utility. Analysing various situations briefly that the original *make* utility encounters, similar tasks that must be handled in `bmake` are discussed. We keep in mind that `bmake` must show as much of the behaviour of *make* as possible. The algorithm by which this aim can be fulfilled is also described.

6.7.1 Make analysis

The Unix *make* utility is instructed by programmers by means of a particular description file, usually named as `Makefile`. The description file tells *make* which source files to check for recompilation. In `Makefile` each source file must be referred to with the programmer-defined files (e.g. header files) it depends on, because *make* also needs to compare their modification times to the time of the corresponding object file.

Sophisticated programs can force *make* to employ quite a complicated checking procedure. In general the production of an executable program includes multiple-level checking of dependencies. Given the executable *program* in Figure 6.14, there are many potential checking points that *make* has to deal with, depending on the programmer's specification. If the programmer includes these points into the *Makefile*, specifying input files for each file that is to be produced, the `make` utility ensures that all the points are checked from *program* downwards and, if necessary, recompiled in reverse order.

In our approach, it would also be possible to use modification times to discover changes made to dependencies. However, sometimes there are some shortcomings in using modification times to decide when a target is out of date. For example, consider two source files, `fileA.c` and `fileB.c`, where the modification times of `fileB.c` and its object file `fileB.o` are more recent than that of `fileA.c`. If `fileA.c` is renamed as `fileB.c`, so replacing it, on many operating systems the new `fileB.c` has the old modification time of `fileA.c`. Then the target (`fileB.o`) would not appear to be out of date with its source file (`fileB.c`), and `make` would wrongly fail to recompile it. To avoid situations like this, there is an alternative approach which is adopted by `bmake`. By ex-

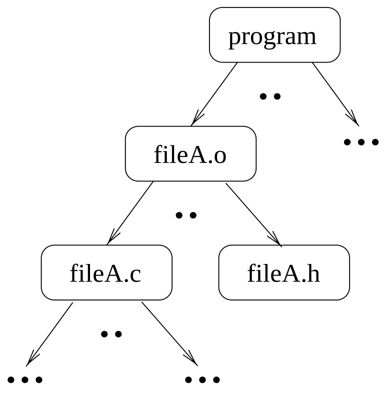

Figure 6.14: A typical tree of checking dependencies

amining the transaction list, the need for recompiling can be determined easily and precisely.

Once an instance of the compilation command is submitted which contains the compiler name, flags, source files, and so on, the compilation can be executed again at any time by typing in `bmake` with the related object file. However, if one of the following situations occurs the programmer cannot naturally use `bmake`, and the relevant compilation commands need to be reissued:

1) Compiler options need to be changed,

2) A new source file is added into the compilation.

3) Files involved in the compilation are renamed.

On the other hand, for example, with the use of the `#`include directive, including new header files, or removing existing ones, does not affect the use of `bmake`. In this kind of situation, the changes are reflected in the transactions recorded for a subsequent compilation, and so are detected by `bmake`, which then becomes aware of the change in the dependencies and can take any necessary remedial action.

6.7.2 Designing *bmake*

The `Tracer` mechanism that `brush` uses to keep command executions under control facilitates the design of the `bmake` program. The design puts trivial overheads on `brush`, although `bmake` needs to cooperate with it, e.g. to have its update commands executed. In addition to the transaction list maintained by `brush`, some other information that is kept locally is of great use in `bmake`.

From the transaction list, `bmake` constructs the entire dependency graph before building any target. As seen in Figure 6.14, the graph normally consists of cascading dependencies: a target can be a dependency of another target which can be a dependency of a third target and so on. Therefore, `bmake` must employ a multi-step procedure automatically, in order to produce the desired target. For example, to bring the executable *program* up to date, it follows a chain of dependencies (through `fileA.o`, `fileA.c`, etc.) until it encounters a file that is not a target. It then executes the commands that update the out of date targets in reverse order, finally updating *program*.

Like *make*, the main task of the `bmake` program is to keep targets consistent with the files that they depend on. A target is considered out of date when one of the following situations happens:

1. The target does not exist in the current state,
2. The dependent files have been modified more recently.

From the transaction list, it is quite easy to find out if these situations occur. If the transactions after the target has been last produced (i.e. the last `Create` on the target in the list) include a `Delete` on it, this means that the target has been removed from the state. On the other hand, if the subsequent transactions point to a `Delete` or a `Create` on at least one of its dependencies, this refers to the modification of that dependency. When targets are out of date with their respective dependencies, `bmake` must complete the compilations necessary to reproduce them.

The other issue is to discover compilation errors. The *make* program observes the compilation of each source file to find out its status. Encountering a compilation error, *make* stops completing all the scheduled compilations, complaining about the error. Similarly, `bmake` should be able to detect the erroneous compilations. Since the compiling commands are executed by `brush`, this information can be gained from the *exit*() system call that the process performing the failed compilation uses to terminate itself.

Editing the dependencies of a target is not the only way of making the target out of date. Sometimes, programmers bring a target out of date by using the *touch* command (which invokes the *utime* system call). The *touch* command changes file access and modification times of a file and thus *make* considers the file to be updated. To be able to achieve the same in `bmake`, we require the `Tracer` to catch the *utime* system call, so that the desired transactions can be stored (see Figure 6.15).

```
int utime (char *pathname, struct utimbuf *times)
    if pathname exists,
        Utime pathname
```

Figure 6.15: *utime* transactions

In this way, the transaction list shows the effect of the *touch* command and `bmake` can find out the up-to-date version of the file.

There are other issues that `bmake` has to deal with. For instance, the commands issued by `bmake` need storing differently from the conventional way adopted in `brush`, and the commands that produce more than one target at a time must be handled carefully. Issues such as this will be examined in Chapter 7.

6.7.3 The algorithm

Below, we present the algorithm that is employed by `bmake`, usually for recompilations. The `bmake` program can be used for commands that are not compilers as well. The philosophy is certainly the same. It can handle only commands which interact with the file system and generate new files, though.

Before describing the algorithm, some concepts need to be made clear. A target is defined as a file that is produced by any user command. Both a source file and an object file (e.g. which might be created by a *yacc* parser-generator and an ordinary compiler command, respectively) can be targets. A dependency is defined as a file that is a prerequisite for some target. In the process of producing an object file, all the files that are needed (e.g. source files and header files) are dependencies.

The generation of a target tends to get bound up with the handling of many other files. Because `bmake` is automatic, it needs to be able to distinguish between targets which need to be remade and original files which do not. Original files, which are also dependencies, are created by commands such as editors; `bmake` needs to avoid re-running the editor commands. Targets and dependencies are determined by the type of transactions of the commands that are associated with them. Targets are created by compiler-like commands and `brush` stores `Create` transactions for them. Thus, the following principle can be defined:

> *A file is a target if the command that produced it has* `Read` *transactions on other files. Otherwise it is an original file.*

Using the principle, all targets that construct an executable program can be discovered smoothly. When a target is to be produced, the command performs `read` operations on dependencies of the target, which presents another principle as follows:

> *A file is a dependency if a command which produces a target has a* `Read` *transaction on it.*

This makes the process of discovering dependencies clear. The number of dependencies of a target are determined by the transactions of the command that creates the target.

The above two principles help `bmake` relate targets to the right dependencies. However, if a particular file is modified by an editor, `brush` stores three transactions: `Read`, `Delete`, and `Create` which run on different versions of the same file (see Chapter 5). In such cases, unless the editing involves other files, the file is considered to be a dependency, but not a target.

Targets (`T`) is kept as a local list of files, and dependencies (`D`) is kept as a local set of files in `bmake`. To illustrate how these are constructed, assume that a file `f` needs rebuilding. Then `bmake` uses the following procedure:

(1) Find the command which created the current version of `f`,

(2) Find the transactions for that command,

(3) Extract the files read by that command,

(4) Extract the files written by that command,

(5) Add the read files to set `D`,

(6) Add the written files to list `T`.

The procedure will not be enough to catch all targets because of the existence of a chain of dependencies. This requirement can be met by the following algorithm which calls the above procedure repeatedly:

```
While D is not empty
    Take a file f from D
    If it does not exist in T and
            is a target rather than an original file
        Call the procedure above with f
```

The algorithm ensures that all targets in `T` are kept in order. After `D` and `T` are fully constructed, `bmake` deals with `T` to rebuild out of date targets in it, checking the appropriate dependencies in `D`. For example, given the target *program* in Figure 6.14, `D` and `T` (having the procedure and then the algorithm invoked) appear like this:

T = {“program”, “fileA.o”, “fileA.c”, ...}

D = {“fileA.o”, “fileA.c”, “fileA.h”, ...}

Beginning with the last target in `T`, `bmake` checks and possibly remakes all the targets. Note that `fileA.c` is both a target and a dependency, as well as `fileA.o`. Sometimes a single command issued by `bmake` produces multiple targets, which can be discovered from the previous transactions of the command. This usually constructs a cyclic group of targets, e.g. mutually recursive modules, and needs such a command to be performed only once. After the command is executed, all the targets that it remakes are removed from `T`.

Chapter 7

Implementation

In order to implement a selective undo facility, there are several essential components that the new shell (`brush`) must be responsible for administering. The shell stores recovery information in them incrementally. In the mechanism used, while the interaction with the system is continuing, the user-level requests (user-issued commands) and the kernel-level requests (command-issued calls) are remembered, and all previous versions of files that user commands have affected are stored. In general, undoing/redoing a user command is a matter of exchanging the relevant current versions of the related files with the old versions. Selective undo also involves checking the applicability of undo and redo commands, using a conflict-detecting algorithm with a suitable policy.

This chapter describes the implementation of the selective undo mechanism which serves as a Unix shell. The algorithms that are used to implement undo and redo commands are given, as well as examples of the situations that bring the system to an undesired state. Then it specifies how the brush modules (`Shell`, `Tracer`, and `Recovery Manager`) work, and cooperate with each other to satisfy the recovery requirement. Analysing and optimising the transactions that a user-issued command performs on the filing system, the method of storing recovery information efficiently is introduced. Lastly, the chapter describes the implementation issues of the `bmake` facility and discloses its relation to `brush`.

7.1 Brush Description

This section presents the behaviour and algorithms of `brush` in detail. User commands are converted into simple representative transactions that can be manipulated easily. Undo and redo algorithms are defined on these transactions and their restrictive roles in carrying out recovery are highlighted.

7.1.1 Operational behaviour

The `brush` shell is to some extent different from contemporary shells. It is said to be in *tracing mode* when executing normal commands and *recovery mode* when executing undo or redo commands. In tracing mode, user commands are traced and recovery information is stored. The tracing and storage of recovery information occur together implicitly during execution of a command. The kind of recovery information stored depends entirely on intercepted system calls, whose effects on the system may vary considerably, and so each system call is handled individually. In recovery mode, undo and redo commands are executed under control of the Recovery Manager.

Recovery from commands is quite simple in our implementation. The effects of the recovery commands consist only of the movement or exchange of file versions between the current state and a backup directory of previous versions. For a command that creates a file from scratch, the `undo` command moves the file to the backup directory and the `redo` command brings it back to the current state. For a command that overwrites a file, `undo` replaces the current version of the file with the previous one kept in the backup directory. Then `redo` reverses this replacement (i.e. undoing the `undo`).

`Brush` locally keeps the history of commands and information about transactions in a data structure. But, this can give rise to some drawbacks. For instance, if the computer or `brush` crashes, the structure is lost and there is no way to retrieve it with its previous values. To make sure any such interruption does not affect information required to undo and redo past commands, `brush` stores recovery information in files as well. Every time the user submits a command the proper changes are made both in the structure and in the files.

A program (e.g. an editor) can overwrite a file many times during execution.

In this case, the program carries out many transactions. Most of these transactions have no effect on the current state and thus on undo and redo operations. So it is unnecessary for `brush` to keep track of them all. The transactions for one command need only contain at most one `Read`, one `FPermit`, and one `Create` or `Delete` for a single file [47]. Using this principle, `brush` optimizes the recorded transactions for a program before storing them. If there is a pair of Create and Delete transactions on the same file version, the optimization removes the pair and any Read transactions in between. Also, if there are multiple Reads on a single file version, only one is kept.

Since all versions of files are saved, after such a program is executed, intermediate copies of files associated with deleted transactions would be stored. Just as transactions are optimized, the corresponding saved copies of file versions are removed. In general, `brush` maintains only two copies of a file which is affected by the execution of a command, the new copy in the current directory and a saved copy of the previous version before the command was executed.

`Brush` stores file copies as a whole in the filestore. Whenever `brush` intercepts the *open* system call, it checks the first argument to see whether it points to a pathname under the user's home directory. If the file is owned by other users or belongs to a standard shared library, for example, the *open* is allowed to go ahead. Otherwise, the second argument is checked to see which operation *open* is to perform on the file. For the operation that is of concern to the Recovery Manager, the relevant information is stored. In the case of the operation representing a `Delete`, for example, a complete copy of the file is taken. (For *open*, a `Delete` means modification of the file as a result of flags such as `O_WRONLY` and `O_TRUNC`.) Then the system call is released to resume as normal.

Like ordinary shells, `brush` starts up the execution of user commands by *fork*ing a new process and making the new process invoke the *exec* system call. Note that shells, `brush` included, locally deal with built-in commands without creating new processes. Once `brush` is started, it runs under control of the Tracer which monitors both it and the later created processes.

Process terminations are intimately observed in `brush`, too. A process in Unix uses the *exit* system call to terminate its execution. The *exit* system call is intercepted for two purposes. One is to fix the time of removing the process from the set held by the Tracer (see Section 7.2.2). The other is to find out the status of the execution of the process. The status of the execution can easily be determined by checking the value of the argument of *exit*. By convention, a status value of zero means that the program terminated normally, and a nonzero value means that an error occurred. However, the latter can also represent the status of a program that partially executed. For example, assuming that *fileA* exists, but *fileB* doesn't, consider the behaviour of the following command:

```
rm fileA fileB
```

Here, as one guesses, the execution leaves one transaction incompleted. The command returns with an *exit* system call whose argument has a value of non-zero to indicate that the execution has failed. But it has partially executed, thereby removing *fileA* from the current directory and returning an error message for *fileB*. In this case, `brush` holds all information for the transactions which completed successfully. If desired, any command which exits abnormally can be undone automatically. In the current version, `brush` provides a user option which controls whether or not this is done.

7.1.2 Implementation algorithms

In this section, we describe the algorithms which are necessary to implement the desired shell features. We assume that system calls are monitored in such a way as to record all the operating system transactions which a command or program carries out. For simplicity, we will initially concentrate on the effects of commands on a collection of files in a single directory.

In order to deal with different versions of the same file, the shell distinguishes them according to their time of creation. The shell numbers all the transactions which occur, and uses these sequence numbers to represent time. The time of creation of a file version is the sequence number of the transaction which produces it. The shell simply needs to keep track of the "current time", i.e. the sequence number to be issued when a transaction is next executed.

The shell keeps all old versions of files in a subdirectory called `save`, say, using their times of creation as file names. For example, suppose that the command sequence:

```
write fileA
write fileB
copy fileB fileA
edit fileB
```

results in the creation of the first version of `fileA` at time 1, the first version of `fileB` at time 2, the second version of `fileA` at time 5 and the second version of `fileB` at time 8. The files existing after these commands are:

```
fileA
fileB
save/1
save/2
```

If `fileA` and `fileB` are changed again at a later date, the current versions are saved by moving them as:

```
fileA  ->  save/5
fileB  ->  save/8
```

Thus the directory is seen in a correct state by programs outside the influence of the shell, except for the extra `save` directory.

The shell ensures that when a command is run, it does not result in any file versions being deleted. File system requests from the command are converted into actions which create new file versions or move them as appropriate.

The shell stores the history list, with the time at which each command was issued, and whether or not the command has been undone. For example:

```
1> write fileA
2> write fileB
3> copy fileB fileA
6> edit fileB
```

The time at which a command is issued does not need to be visible to the user, provided that there is some way for the user to indicate which command to undo or redo as necessary.

In addition, the shell stores the sequence of transactions which occurs as commands are executed. For simple files, we need only consider the creation of a file version, reading from a file version, and deleting a file version. Other file system requests can be handled as combinations; for example, a request to open a file for appending can be treated as a `Delete` followed by a `Create`. For each transaction, the file name and version affected are stored. For example, for the above command sequence, the transactions are:

```
1> Create fileA 1
2> Create fileB 2
3> Read fileB 2
4> Delete fileA 1
5> Create fileA 5
6> Read fileB 2
7> Delete fileB 2
8> Create fileB 8
```

The history list and the transaction sequence allow the shell to work out which versions of files are current, and which versions were current at any particular time in the past. To do this, the shell keeps track of deletions of file versions, which are treated as separate transactions. If a command overwrites a file, this is treated as two transactions; one to delete the old file version, and one to create the new version. If we take time `t` to mean the time just before the transaction tagged with sequence number `t` occurs, the file versions current at time `t` are those with active creation times less than `t` and active deletion times (if any) greater than or equal to `t`.

In addition, the shell keeps track of which transactions are active or not, i.e. which transactions belong to active commands. (In fact, the information about which transactions are active, and the version numbers of the files, are redundant and could be re-created from the remaining information and the history list.)

This information about transactions does not normally need to be visible to the user. For built-in commands, the information can be generated from a knowledge of what the commands do. For other commands, the information can be gathered while the command runs by monitoring its system calls, as discussed in Chapter 6.

It is possible to allow the user to see the transaction sequence for a command, and to selectively undo or redo individual transactions. If not, the fact that all the transactions belonging to a particular command are undone or redone together allows the transaction sequence to be thinned out, if desired. For example, the transactions belonging to one command need only contain at most one `Read`, one `FPermit`, and one `Create` or `Delete` for each file. For a file version that existed before the command was issued, a `Read` is stored if the contents of that file version may have been read by the command, and a `Delete` if the version does not survive after the command. For a new file version that the command produces, a `Create` is stored. Nothing needs to be stored for temporary files, i.e. ones which are created and deleted during the course of the command.

How does the shell implement undo? First, it has to detect whether a request to undo a command is valid. If a file was created or altered by the command, and the new version of the file is mentioned in the transactions of any subsequent active command in the history list, then the undo is invalid because it would lead to a version inconsistency. If a file was deleted by the command, and there is a subsequent command which creates a new version, then again the undo is invalid (because the behaviour of the command may depend on the non-existence of the file, as with the use of the `O_EXCL` flag in Unix). There are no other restrictions; an undo request is otherwise always valid. Next, the shell can use the command's transactions to determine how to change the current state of the filestore, effectively undoing the transactions one by one in the reverse order.

How does the shell implement redo? Again, as with undo, files created, altered or deleted by the original command must not be mentioned in any subsequent active transactions. However, there are additional restrictions to ensure that the redone command accesses the same versions of files as when it was originally issued. Suppose that the original command was issued at time `t`. If the command created a file without first deleting an older version, and an older version of the file is now current at time `t`, then the redo is invalid. For example, suppose the shell encounters the following situation: A user first executes two commands and undoes the second command. Then the user executes a new command and also undoes it.

```
1> write fileA
2> (copy fileA fileB)
4> (write fileB)
```

This sequence of commands creates the transactions:

```
1> Create fileA 1
2> (Read fileA 1)
3> (Create fileB 3)
4> (Create fileB 4)
```

In this case, the shell must prevent two undone commands (`copy` and `write`) from being active at the same time, because the behaviour of these commands depends on the non-existence of `fileB`. Therefore, once the `copy` command is redone, the `write` command cannot be redone and vice versa.

Another restriction is that, for any file which was read or deleted by the command, the version which was originally affected must match the version which currently exists at time `t`. For example, suppose we have the following situation:

```
1> write fileA
2> (edit fileA)
5> (copy fileA fileB)
```

with stored transactions:

```
1> Create fileA 1
2> (Read fileA 1)
3> (Delete fileA 1)
4> (Create fileA 4)
5> (Read fileA 4)
6> (Create fileB 6)
```

A request to redo the `copy` command is not valid, because the version of `fileA` (version 4) which the `copy` command was supposed to act on does not currently exist at time 4. This restriction ensures that a redo can always be implemented by manipulating file versions rather than by re-executing the command itself. In this example, the `edit` command needs to be redone before the `copy` command can be redone, perhaps inserting a command to save the old version of `fileA`, if desired.

The insertion of a command into the history list other than at the end is handled in a similar way to a redo request, except that the versions of the files affected can be taken to be the ones existing at the time of insertion. The shell may not discover that the inserted command is invalid until it is part-way through execution, but the failure can be reported and the partial effects of the command can be undone.

In the above, we have prevented undo or redo whenever a file is involved which is mentioned later. It is possible to be less restrictive. If two successive changes to a file involve disjoint portions of the file, the changes may be regarded as independent. This is the approach taken in systems like CVS [59] where multiple developers may work on the same program source file, the changes in the text are stored using RCS [89] and treated as independent whenever they don't overlap. In our setting, this would allow two non-overlapping changes to a file to be undone or redone independently of each other, and would in general reduce the amount of space used for saved versions of files.

7.1.3 Other algorithms

So far, we have only dealt with simple files. Other local state changes are dealt with in similar ways. Consider attributes of files, such as permissions and alias information, for example. A change of attributes can be handled correctly with the mechanisms already described by creating a new version of the file with the new attributes attached. However, the storage of multiple versions of the file contents can be avoided by adding an extra transaction type (`FPermit`) which records a change of attributes, and which can be undone by reversing the change. Also, some undo and redo operations can be allowed, which would otherwise have been invalid because of the change of file version, provided care is taken to ensure that the attribute changes do not affect the relevant commands.

File permissions really have great influence over the executability of commands. Changing them can prevent commands from functioning properly, allowing limited accesses to the prerequisite files. The fact that file operations require appropriate permissions restricts the use of recovery commands, which is necessary for a change of permissions to make sense in the behaviour of the system. For example, assume the user executed the following sequence of commands:

```
1> write fileA
2> chmod +r fileA
3> copy fileA fileB
```

for which `brush` stores the transactions

```
1> Create fileA 1
2> FPermit fileA 1 300 744
3> Read fileA 1
4> Create fileB 4
```

where the octal representations are used to denote old and new permission information of `fileA` at time 2. An attempt to undo the `chmod` command fails, because it cancels the `read` permission for the version of `fileA` which is needed by the `copy` command. This ensures that changes to permissions of file versions made by an undo do not affect any subsequent command.

Similar restrictions also apply to redo. For a redo to be able to run on a command successfully, the files that the command acted on must have suitable permissions, as well as their right versions. For instance, consider the following situation:

```
1> write fileA
2> (chmod -w fileA)
3> edit fileA
```

which creates the transactions

```
1> Create fileA 1
2> (FPermit fileA 1 300 100)
3> Read fileA 1
```

```
4> Delete fileA 1
5> Create fileA 5
```

where it is obvious that the `chmod` command was undone before the `edit` command was executed. The version of `fileA` (version 1) that the `chmod` command affected does not exist currently at time 6. Therefore, the `FPermit` transaction cannot be re-executed on the version of `fileA` unless the `edit` command is firstly undone. If `chmod` is redone after `edit` is undone, then it will not be valid to redo `edit` because the related version of `fileA` has no `write` permission. In other words, the shell cannot allow `chmod` and `edit` to be active at the same time. With this restriction, file versions are protected against unauthorised operations.

Directories are handled by adding transaction types (`Remove`, `Make`, `DPermit`, and `SLink`) which record changes to the directory structure. At first sight, it appears that multiple versions of directories need to be stored, as their contents change with time. However, this is not necessary, as the state of a directory at some previous time `t` can be inferred by working out the file versions which were current in it at time `t`, as described earlier. For example, suppose the history list contains the following commands:

```
1> mkdir dirA
2> cd dirA
3> write fileA
4> copy fileA fileB
```

which lead to the transactions

```
1> Make dirA 1
2> Path dirA 1
3> Create fileA 3
4> Read fileA 3
5> Create fileB 5
```

At time 4, `dirA` only contains the version of `fileA` (version 3). At later times, say 6, the version of a new file, `fileB`, (version 5) appears in the directory.

The effects of recovery commands on directory transactions are simple. The removal of an empty directory from the state represents an undo, and the re-creation of it represents a redo. Obviously, various situations can make undo and redo operations inapplicable. In the above example, an undo command cannot be performed for the `mkdir` command, since it is not empty. The only way to undo the command is firstly to undo the later three commands in reverse order (`copy`, `write` and then `cd`). In this way, when `mkdir` is undone, it can easily be seen that the other commands, which all depend upon it, cannot be redone. Many similar situations do not allow easy recovery, some of which will be addressed in Section 7.1.4.

As far as recovery is concerned, the transactions on directories are much less restrictive than ones on files. Moreover, the directory transactions are more rare in use in terms of both task-oriented and recovery-oriented commands. Undoing or redoing a command, whether it affects files or directories, does not require the current directory to be the directory in which the command has been originally issued. No matter what the current directory is, files and subdirectories which have been modified by commands issued in other directories can be restored. For example, even after the user extends the above situation with a new command that changes the current directory, it is possible to undo the `copy` command through the changed-to directory.

The `Path` transaction seems to be state-ineffectual, because it selects only branches in the tree-like structure of the file system. We want it not to put restrictions upon recovery operations. This does not cause any problem in practice. Whenever the directory that `Path` changed exists, it can be recovered. It does not matter what the current directory is. Thus the directory that undo changed may be reached by employing redo at any time and vice versa.

For the most part, external effects such as printing or communicating with remote facilities are simply declared as being outside the scope of the shell. Although these effects are detectable in principle by monitoring the system calls which access the relevant devices, in practice the effects are not reversible, or the intentions implicit in the device accesses are impossible to interpret. The user must be made aware that an undo command only undoes the local effects of a command, and other external effects are left to the user to deal with.

In addition, some local files may not be suitable for handling by the shell. For example, some files might carry with them special security or system integrity implications. To cover such cases, the shell provides a way of making specific files or directories exempt from the usual version handling, so that changes to them are treated in the same way as external effects which the user must handle. Database systems also often pose a problem. A database may be held on a separate disk or disk partition outside the control of the normal file system, or it may be held in a large file for which multiple versions might take up too much space. On the other hand, database systems often provide their own recovery facilities, so it makes sense to exempt them from the usual version handling too.

7.1.4 Implementation difficulties

As seen in the previous section, it is quite complex to define general conflict-detecting algorithms covering directory transactions. The complexity is because of the fact that the user can produce numerous unpredictable situations, especially issuing commands that modify directories and their access permissions. These situations may make the recovery process even for the restoration of a single file very long and expensive.

An unpredictable situation is associated with how much the user has modified the appearance of the file system until the time of recovery requirement for a command. The command may have been executed with arguments of partial or absolute pathnames some of whose components have been renamed or removed in between. Such pathnames have to be reconstructed to be able to carry out the checking algorithms and recovery. Given that many directories can contain subdirectories with the same name, this may lead `brush` to the case where the true pathnames may not be noticeable and so it may be difficult to implement recovery operations. Consider the following simple situation.

(1) A command is issued which affects some files in two separate directories,

(2) Both directories are renamed after a few more commands are issued.

Undoing the command in (1) requires `brush` to discover the renamings of both directories and check for conflicts, no matter which directory it has been executed in. If (1) has used partial names to specify files in the other directory, or (2) has been done through a different directory, it might not be possible to keep track of directory and pathname changes. In this manner, there might be problems with reconstructing the pathnames, resulting in the failure of recovery.

In fact, `brush` aimes to allow recovery as in many situations as possible. The minimum requirement is the existence of the directory with suitable permissions at the time of recovery. A directory which contains some files modified by a user command may not have the same name or version number during recovery if subsequent commands have operated on it. A change of name should not affect recovery from that command. Otherwise, the domain of recovery commands may become highly restricted. The version number of the directory can be changed in two ways; re-creating the directory after removal and renaming it back after its name is changed. The first way must prevent recovery, while the second one must not. It is easy to check for the removal of a directory. For a renaming command, a `Remove` transaction on a directory, whether empty or not, is always followed by a `Make` transaction, which means that the directory has been renamed. If `Remove` is not followed by `Make`, the directory removal has occured and any file restoration cannot be made to that directory. Checking for directory access permissions is also important because, for example, the absence of `execute` permission does not allow file transfers in that directory.

Below, we try to explain the above-mentioned difficulties via two special situations. For the first example, suppose the user has a fragment of the history list as follows.

```
10> cd dirA
11> write fileA
12> edit fileA
    ...
30> chmod -x dirA
```

which produce the transactions

```
10> Path dirA 9
11> Create fileA 11
```

```
12> Read fileA 11
13> Delete fileA 11
14> Create fileA 14
    ...
30> DPermit dirA 9 755 644
```

A request to undo the `edit` command cannot be serviced directly, even if it is assumed that the commands between times 8 and 30 do not modify `fileA`. This is because the `chmod` command, which has locked `dirA` against file accesses, prevents the movement of the previous version of `fileA` (version 11) into `dirA` for the representation of the effects of undo.

In the example, the user would probably expect it to be possible to be able to undo `edit`, which is actually necessary not to be too restrictive in the use of recovery commands. One way to do this is that the `chmod` and `edit` commands are firstly undone and then `chmod` is redone. The difficulty here is with the undoing of `chmod`, because a later command of the history could rename `dirA` or undoing `chmod` could create conflicts with some later command, complicating the recovery process. Note that it is legal to rename a directory with no `execute` permission.

For the second example, suppose that a fragment of command history contains the following commands.

```
10> cd dirB
11> edit fileB
    ...
30> move dirB dirC
```

which create the transactions

```
10> Path dirB 5
11> Read fileB 6
12> Delete fileB 6
13> Create fileB 13
    ...
30> Remove dirB 5
31> Make dirC 31
```

where we want to allow the user to undo the `edit` command. In this case, an undo command for `edit` must bring the prior version of `fileB` (version 6) back under `dirC`, not `dirB` in which it was possibly created originally. However, this undo might become complicated if later commands have operated on `dirC` or its permissions. It is not easy to discover such situations and perform undo.

The examples above point out that difficulties in achieving recovery potentially arise from directory renamings. `Brush` may not be able to cope with them to the extent that the user would like, although it finds remedies for some of them. When a directory's access permissions prevent recovery, `brush` changes them to the right ones, undoes the relevant command, and then sets the old permissions back, without handling the command (`chmod` in the example) that has caused the problem. The same is done for the redoing of the command. Nonetheless, changes on a directory in which a command has been executed do not prevent recovery from the command, as long as `brush` can keep track of them. The command is undone or redone in the new directory. As a result, the user must bear in mind that `brush` may fail to deal with some commands, even if their reversal seems conflict-free and is expected of the shell.

7.1.5 Pre-existing files and directories

The examples that we have given so far have contained files and directories which were created through `brush`. This was to avoid causing confusion about the representation of their version numbers. In practice, a user would have many files and directories which exist before `brush` is started, and desire to act on them in the shell. The existence of such files and directories may affect the applicability of undo and redo commands, since the conflict-detecting algorithms require them to have been created under `brush`'s control.

For the user to be able to use recovery commands most effectively, `brush` records all pre-existing files and directories by means of `Create` and `Make` transactions, respectively. The transactions are recorded by starting from the home directory and going down. This order is essential to allow the right discovery of conflicts between commands that are to be issued. Each transaction recorded is thought of as if the file or directory was created from the user's home directory. If the shell is initialised in a subdirectory, then an extra transaction, `Path` that represents change to the subdirectory, is recorded, which is used to identify the current directory. In the case of pre-existing files which are linked to other files or directories, `brush` uses the `Read` or `SLink` transactions, as well as `Create` and `Make`. The transactions that belong to the pre-existing links are recorded to the transaction list immediately before the `Path` transaction, to ensure that the transactions for files or directories that they point to must have already been recorded.

To show the record of such transactions explicitly, suppose that a user's home directory contains one file, `fileA`, one directory, `dirB` which contains one file, `fileB`, and a link, `fileC` which points to `fileB`. If the user initialises `brush` in `dirB`, the following transactions are recorded.

```
1> Create fileA 1
2> Make dirB 2
3> Create dirB/fileB 3
4> Read dirB/fileB 3
5> Create fileC 5
6> Path dirB 2
```

Each pre-existing file and directory is given a version number which represent its present contents.

These transactions basically help determine the validity of redo. The shell does not allow the user to undo or redo them. Therefore, note that the history list does not include commands that correspond to them.

7.2 Tracer Implementation

The `Tracer` is the module of `brush` that deals with control of file system accesses. It traces each command individually. This section specifies how the `Tracer` is implemented so that it can observe all file requests of commands securely. The tracing-related issues are then discussed, including the operating system restrictions and the `Tracer` layers.

7.2.1 Tracer structure

In our Solaris implementation, the `Tracer` monitors user processes using the **/proc** virtual file system [34]. This is the same method used by monitoring programs, such as *truss* or *strace*, which are also available on a number of other UNIX platforms, including Digital Unix, IRIX, BSD, or Linux. The System V **/proc** interface allows an individual process to be monitored and its events of interest to be exposed by operating on the file associated with a user process.

In particular the `Tracer` attaches to a subject process having a process identifier *pid* in the **/proc** directory. The **/proc** file is opened for exclusive read/write use as follows:

```
sprintf (processName, "/proc/%d", pid);
procfd = open (processName, O_RDWR | O_EXCL);
```

In this way the `Tracer` can avoid collisions with other tracing processes. Once attached, the `Tracer` uses *ioctl* system calls on the open file descriptor, `procfd`, to control the process. It can instruct the operating system to stop the subject process on a variety of events of interest. In `brush` there are two events of interest: system call entry into the kernel and system call exit from the kernel. `Brush` needs to examine system calls such as *open* and *creat* only on entry into the kernel. The following code indicates how the `Tracer` tells the operating system which system calls need to be stopped on entry.

```
sysset_t SysCalls;
premptyset (&SysCalls);
praddset (&SysCalls, SYS_open);
praddset (&SysCalls, SYS_creat);
ioctl (procfd, PIOCSENTRY, &SysCalls);
```

Some other system calls (e.g. *fork* and *vfork*) are stopped on exit from the kernel, to allow the `Tracer` to take the values returned from the operating system. For example, the result that the *fork* system call returns is the pid of a child process and the `Tracer` can use it to open its associated **/proc** file. Thus all child processes created by the subject process can be tracked.

```
premptyset (&SysCalls);
```

```
    praddset (&SysCalls, SYS_fork);
    praddset (&SysCalls, SYS_vfork);
    ioctl (procfd, PIOCSEXIT, &SysCalls);
```

As with the subject process, the operating system needs to find out tracing flags for each child process. System V provides an elegant way for the set of trapped system calls to be automatically inherited from parent to child. In order to make sure that child processes inherit all of the subject process' tracing flags, the `Tracer` should set the *inherit-on-fork* flag in the process. This makes child processes also stop whenever they begin a system call of interest.

```
    long flags = PR_FORK;
    ioctl (procfd, PIOCSET, &flags);
```

Until registering the set of system calls to intercept, the subject process must not start executing, so that the `Tracer` can be allowed to attach to it and initialize the tracing flags at the precise time. Therefore, the execution of the process is delayed by `brush` for a very short time.

Once events of interest are specified to the operating system, a new *ioctl* system call can be used to make the `Tracer` wait for the subject process to stop.

```
    prstatus_t *pstatus;
    ioctl (procfd, PIOCWSTOP, &pstatus);
```

At this stage, as soon as the stop takes place, the `Tracer` reads the registers and the status information (`pstatus`) for the process. It uses this to examine the parameters or the result of system calls. If it encounters an event for which it is necessary to hold recovery information, it passes control on to the `Recovery Manager` and waits for it to complete the task.

Finally, after having control back, the `Tracer` restarts the execution of the stopped process.

```
    ioctl (procfd, PIOCRUN, 0);
```

The process keeps running until it performs a new system call of interest. Eventually, when the subject process terminates or is killed, the `Tracer` detects

```
attach to subject process
register the set of system calls to intercept
while subject process is running do
    wait for process to stop on system call entry or exit
    examine system call number and its parameters/result
    call Recovery Manager for this system call
    if system call is fork then
        begin monitoring new child process
    resume system call
endwhile
```

Figure 7.1: Outline of the Tracer algorithm.

this and stops tracing the process. Figure 7.1 summarizes how the discussed functionality is used in the `Tracer`.

Conceptually, `brush` can be considered as an agent that implements recovery-supported versions of the system calls intercepted by the `Tracer`, but in practice they are not serviced directly. The **/proc** interface requires that the intercepted system calls always go through the kernel. It is possible to "abort" a system call on its entry to the kernel, intercept its exit from the kernel, and have `brush` service the call. In order to avoid reimplementing much of the existing operating system functionality, we didn't choose to follow this route in `brush`. Instead, we allow the system calls to exhibit their own original behaviours by reading the call's parameters, discovering the pending changes on the file system state and storing desired recovery information. For the *open* system call, which imposes the most complicated implementation on the `Recovery Manager`, these actions are associated with the analysis of the first two arguments, the file name string and the operation over the file.

7.2.2 Tracer discussion

With the `Tracer` mechanism, `brush` creates a personalized filestore where all past versions of a file are maintained. We can observe requests made to the kernel, allowing individual users to run `brush` on their own private operating system. Any user can use the "new" functionality without having to modify the original operating system or needing root access. Although the current `Tracer` primarily concentrates on intercepting system calls, it must also intercept and act on some signals. This allows for user-generated signals rather than ones delivered by the kernel to the traced process.

Other potential concerns with the `Tracer` mechanism are the execution of setuid programs and dealing with the premature demise of the `Tracer` process. In practice, these limitations do not pose any additional burden on the user [2]. Setuid programs always run under the identity of the owner of the executable file rather than under the identity of the user that starts them. The `Tracer` cannot control setuid processes, since the security policy of the operating system disallows user-level processes from attaching to other users' processes. An attempt to execute a setuid program causes it to be trapped on the first system call and stay trapped until it is killed or detached. The `Tracer` cannot restart it.

This is not seen as a problem in practice, since very few programs are installed as setuid, and those few, e.g. ps and rlogin, do not really require access to the users' own file system. In the current implementation, whenever the `Tracer` detects that a subject process is about to execute a setuid program, it simply stops tracing the process. The process continues to run normally, with correct credentials, but no longer under control of the `Tracer`.

All other processes to which the user's address space is allocated run under control. To maintain the monitoring of currently running processes, the `Tracer` uses a process table. It helps the `Tracer` control the execution of simultaneous processes and obtain desired information concerning them. Essentially, the table contains only a few fields, as shown in Figure 7.2:
Each time the `Tracer` attaches to a new process, which can be either statically started by the shell or dynamically created by a user command, the process is added into the table by setting its fields with corresponding values. The macro `MAXPROCS` specifies the maximum number of processes that the operating system allows to be traced at the same time.

Indeed, handling compound commands and complex application programs may involve tracing multiple processes simultaneously. To trace a set of pro-

```
struct processTable {
    pid_t pid;              /* process id */
    int procfd;             /* process descriptor */
    prstatus_t *pstatus;    /* process state information */
} procT[MAXPROCS];
```

Figure 7.2: The process table of the `Tracer`

cesses, the *poll* system call is used; this allows the Tracer to monitor the first interesting event to happen to any of the processes. The Tracer arranges to trace exit from the *fork* and *vfork* system calls. When the traced process forks, the Tracer adds the new process to its set of controlled processes by using the pid returned by the *fork*. It holds the child process in the set until it detects the *exit* system call made by the child[1].

As depicted in Figure 7.3, the Tracer, in fact, is composed of two layers: the Process Handler and the Syscall Handler. After a stopped process is attached for tracing, the Process Handler firstly resumes it with an *ioctl* system call. Controlling currently running processes, the Process Handler secondly waits for one process to be stopped by the kernel. Then it finds out the system call on which the process is stopped and passes control to the Syscall Handler.

The Syscall Handler takes information about the system call of interest from the stopped process and checks to see whether it is associated with the file system or not. The file system calls (e.g. *open*() and *creat*())are stopped on entry to the kernel, while the process creation system calls (e.g. *fork*() and *vfork*()) are stopped on exit from the kernel. The Syscall Handler forwards the file system calls together with their arguments to the Recovery Manager. If the incoming system call is creating a new process, the Syscall Handler attaches to the new process and directs it to the Process Handler.

[1]Instead of the *poll* system call, each process could be monitored by creating a new process in the `Tracer`, as adopted in [40]. But, we observed that this might cause some problems, such as rendering traced processes deadlocked.

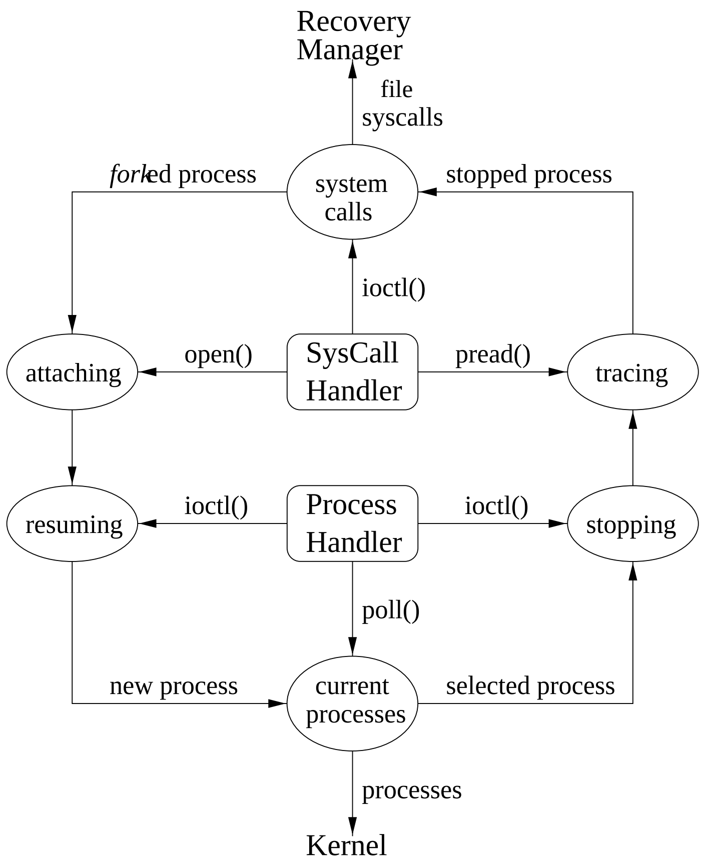

Figure 7.3: General structure of `Tracer`

7.3 Recovery Manager Implementation

The `Recovery Manager` is the module of `brush` that manages recovery situations. It employs some internal layers among which the stages of recovery are distributed. This section describes the configuration of the `Manager`, and the cooperation of its layers with each other and the other modules of `brush`. It also explains which tasks each layer implements.

7.3.1 Configuration

The `Recovery Manager` can be treated as an undo support sub-system that is constructed through `brush`. It deals with the collection of recovery information and the achievement of the recovery process. There are four basic layers which are directly used in the `Manager`:

1) `File Checker`, which is responsible for determining that it is necessary to store recovery information for a system call forwarded by the `Tracer`.
2) `Filestore`, which is responsible for storing procedural information about how to undo or redo a user command.
3) `Conflict Detector`, which is responsible for determining whether the undoing or redoing of a command forwarded by the `Shell` can create unexpected state changes or not.
4) `Undo/Redo`, which is responsible for performing the recovery commands and storing extra information needed to reverse them.

Figure 7.4 shows how these layers are basically configured. The `Shell` and the `Tracer` are two notable callers that need the assistance of the `Manager`.

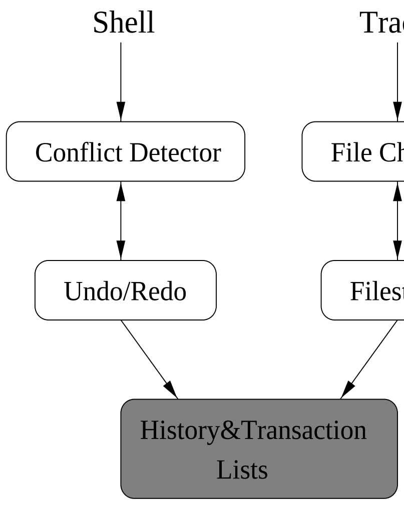

Figure 7.4: Basic configuration of `Recovery Manager`

`Recovery Manager` does not interact with the user directly. It is the command interpreter (the `Shell`) which handles the interaction between a user and the `Manager`. The `Shell` manages how recovery commands are specified and how the results of these commands are notified to the user. The recovery process is completed under control of the `Manager`.

The `Recovery Manager` is almost always in operation in `brush`. There are in general two phases that the `Manager` has to deal with, a preparation phase that records the information needed to undo a command and an execution phase when the command is undone or redone. The `File Detector` and `Filestore` operate in the preparation phase, while the `Conflict Detector` and `Undo/Redo` operate in the execution phase.

The flow of control in the whole system is switched on among the `Shell`, the `Tracer`, and the `Recovery Manager`. After the user issues a command and before the command is executed, the `Shell` is in operation. It reads each command from the terminal and determines whether it is a task-oriented or recovery-oriented command. For a task-oriented command, the `Shell` initiates the execution of the command and then passes control to the `Tracer`. Connecting to the process that runs the command, the `Tracer` monitors the file system activities of the process. When it encounters a system call of interest to the `Recovery Manager`, it passes control to the `Manager` which does the preparation for undoing. After finishing the preparation work, the `Manager` passes control back to the `Tracer` to continue its operation, waiting for another system call of interest to be traced. If the process is aborted because of an error, the `Shell` takes over control from the `Tracer` immediately. According to the user's request, the effects of the preparation for undoing that command is either removed or stored. Otherwise, after the process terminates, the effects of the preparation will be stored for possible later undoing. For a recovery-oriented command, the `Shell` passes control to the `Manager` which then deals with the recovery process.

When control is passed from the `Tracer` to the `Manager` to prepare for undoing, it accesses the `File Checker` layer which contains semantic knowledge and some filtering techniques concerning the meaning of the system call for that command. The `File Checker` layer will determine whether there is a need to store undoing information for that system call. If the system call is of interest, the `Manager` will access the `Filestore` layer which contains undo techniques concerning the efficient storing of the information. The `Filestore` layer will then produce necessary undoing information and update the transaction list by adding one or more entries for this system call [2]. Next, the `Tracer` takes control back until a new system call is intercepted, in which case the `Manager` will be in operation again. In this way they will go through the preparation phase for undoing. Finally, when the execution of the command ends, `Filestore` will update the history list by adding an entry for this command.

When control is passed from the `Shell` to the `Manager` to execute recovery, it accesses the `Conflict Detector` layer which contains conflict-discovering algorithms for the transactions of the relevant command. The `Conflict Detector` layer will determine whether it is safe to undo/redo the command. If there is no conflict that will occur, the `Manager` will access the `Undo/Redo` layer which contains the information of how the command is to be undone or redone. The `Undo/Redo` layer will then undo/redo the command, and update the history and transaction lists. Thus the execution phase is completed. Next, the `Shell` takes control back. As a result, the `Manager` cooperates with the `Tracer` in the preparation phase, and with the `Shell` in the execution phase.

7.3.2 File Checker

A user-issued command tends to use many system files (standard library files, etc.) during its execution, particularly for performing `read` operations. A small number of them are probably the user's own files. The `File Checker` layer checks system calls forwarded by the `Tracer` for the discovery of such files. Directories also have to be distinguished according to the ownership of a particular user.

Unfortunately, there is no effective way at the user level for `File Checker` to finds out if a component (a file or a directory) that is involved in the executon of a file system call is owned by the user. However, the following entities provided by Unix systems may be used to determine the owner of the component to some sufficient extent.

- device IDs
- access permissions

[2] In effect, entries are added to a local list held by `Filestore`. After the command is executed, they are assessed in a proper way and then placed into the transaction list as a whole (see Section 7.3.3).

- pathnames
- inode numbers

Library files and the user's components in Unix operating systems are often held on separate disks that are mounted to the system. Given that an intercepted system call operates on a component of the system, if the ID of the device containing a directory entry for this component is not equal to the ID of the device containing the user's home directory, then we can assume that the component does not belong to the related user (see Figure 7.5). Otherwise, some other entities of the component must be checked, since many users can share the same disk. If the pathname of the component embodies the user's home directory (usually username) or the user has write access to the component, it can be assumed that the user is the owner of the component. As discussed in Chapter 6, it is not easy to identify the component by using its pathname only.

One can think of comparing the inode number of that component to those of all components that the user owns, beginning from the present directory. Indeed, this solves the above problem precisely. But it is not efficient because it can require much processing time and so increase the response time of the shell. Instead, in the current implementation, `File Checker` uses device IDs to eliminate system calls that operate on library files. If both the IDs are identical, then it examines the pathname and access permission of the related component for a final decision. In this case, the user must be aware that `File Checker` cannot discover a component that resides on a separate disk or is linked by a component on it.

Sometimes a user may be allowed to modify some components that belong to the superuser or other users. If this is the case, the currently-used algorithm may cause `File Checker` to ignore the transactions associated with those components, no matter which operations are performed on them. Thus any modified component cannot be turned back to its previous state.

Such a distinction between components helps decrease the number of system calls which are to get involved in the recovery process. For example, it is possible to eliminate a reasonable number of *open* system calls which open library files for reading. With this function, `File Checker` allows the transaction table to keep track of transactions of system calls that affect only files and directories

```
int checkComponentOwner (char *component)
{
    char *home;
    struct stat buffer1, buffer2;
    home = getenv ("HOME");
    stat (home, &buffer1);
    stat (component, &buffer2);
    if (buffer1.st_dev == buffer2.st_dev)
        return 0;
    return -1;
}
```

Figure 7.5: Partial code for discovering a user's own components.

of the user's own. When it decides that a system call is of interest to store its recovery information, control of the preparation phase is passed on to the `Filestore` layer.

7.3.3 Filestore

The `Filestore` layer locally holds a list called the *transaction table* which contains transactions temporarily. When `File Checker` forwards a system call of interest, `Filestore` records its transactions in the transaction table. The recording continues until the end of execution of the related command. Once the command terminates, the transaction table is optimized, as will be seen in Section 7.5, and then the remaining transactions are stored permanently.

`Filestore` also manages the contents of the `.save` directory that is created by `brush` when it is initiated. The `.save` directory is composed of the history list, the transactions list, and copies of the file versions. For each system call that operates on the file system during the execution of a user command, it updates the transaction table with the corresponding transactions and stores necessary file versions. For the *exit* system call that signifies the termination of the command, it updates the transaction list with the optimized transactions of the table and the history list with an entry for that command.

It is important for `Filestore` to know whether a file system call runs successfully or not. This knowledge is used to avoid storing any redundant undoing information. Since file system calls are traced only on entry to the kernel for reasons of the performance of the `Tracer`, the values that are returned by them cannot be caught. However, the current state can provide adequate information to find out the possible status of the system calls. `Filestore` thus determines the executability of each system call by checking the existence and access permissions of the components affected.

`Filestore` produces factual information for the execution phase which will be used by both `Conflict Detector` and `Undo/Redo`. Each transaction in the transaction list has the form

time> *transaction status component version others*

where *status* shows whether the transaction is currently active (0) or inactive (1), which is used to check for the applicability of recovery commands. Each transaction of a file system call which has been executed successfully has an entry in the transaction list. The entry contains the following information

1) time of the transaction
2) transaction name
3) undo status
4) component name
5) version number of the component
6) some other useful information

The number of entries in the transaction list varies according to the type of the system call and the existence of the component connected to it. For some transactions, the entries can contain extra information; for example, an entry of `FPermit` has extra information concerning access permissions of the component that it takes.

Similarly, the number of file versions that will be stored also depends on the kind of the system call. The analysis in Chapter 6 shows that system calls can be reversible or irreversible. Irreversible system calls have been excluded from the set of undoable system calls and only reversible system calls have been considered to be of concern. There are examples of all three kinds of reversible system calls discussed in Chapter 6 in the undoable system call set that `Filestore` handles. However, some system calls can belong to more than one kind. Chapter 6 also discusses five recovery techniques. Effectively, the *partial checkpoint technique* has been used by `Filestore` to prepare the system for the undoable system calls. Concentrating on transactions, `Filestore` stores a copy of file versions whenever any system call will perform a `Delete` on them.

Finally, `Filestore` ties user commands with transactions which are created for them. Each command in the history list has the form

number> *command arguments*

where *number* is the *time* of the first transaction in the transaction list which belongs to the command. Using any two successive *number*s, it can be determined how many transactions a command has produced.

7.3.4 Conflict Detector

The `Conflict Detector` layer is the most difficult part of the `Recovery Manager` to implement. It handles the discovery of inconsistent state changes that are likely to arise during recovery. This involves looking through the past interaction. The analysis of the current state does not help discover all potential conflicts. Instead `Conflict Detector` analyses the transaction list that keeps track of the past interaction. To do so, it employs a complicated checking procedure (see Section 5.3.6) which covers transactions on files and directories.

The checking algorithms for the `undo` and `redo` commands operate on different sets of command transactions, as specified below:

- The transactions of a command to undo are compared to the transactions of all the subsequent active commands.
- The transactions of a command to redo are compared to the transactions of all the active commands.

Whenever a particular transaction conflicts with transactions of other commands, the undoing or redoing of the command that it belongs to cannot be allowed.

Conflicts can be created by many situations. To depict some examples of such situations, assume that a user has submitted a command which has the following transactions:

```
20> Read 0 fileA 11
21> Delete 0 fileA 11
22> Make 0 dirB 22
23> Path 0 dirB 22
24> Create 0 fileB 24
```

If the user executes some more commands subsequently whose transactions, for example, are

```
50> Create 0 fileA 50
51> Delete 0 dirB/fileB 24
52> Remove 0 dirB 22
```

then the first command cannot be undone, because three out of these later transactions individually tend to create a conflict. The line 21 clashes with the line 50 because a newer version of `fileA` now exists, the line 22 clashes with the line 52 because `dirB` has been deleted again, and the line 24 with the line 51 because `fileB` has been deleted.

Now, for an example of conflicts that arise during redoing, assume that the user has recently undone a few commands in some early stage of interaction that produced, for example, the transactions

```
20> Make 1 dirA 20
21> Create 1 fileA 21
22> Remove 1 dirB 10
```

Additionally, suppose the later stage of interaction contains a command which was undone before the above commands and has the following transactions

```
50> Path 1 dirA 20
51> Delete 1 fileA 21
52> Make 1 dirB 52
53> Create 1 dirB/fileB 53
```

Then the last command cannot be redone. The line 50 clashes with the line 20, the line 51 clashes with the line 21, and the lines 52 and 53 with the line 22. Some detailed examples of how `Conflict Detector` discovers the above clashing situations can be found in Sections 7.1.2 and 7.1.3.

In similar ways, `Conflict Detector` detects all conflicting situations. When a conflict is encountered, it notifies to the user the command which causes that conflict. It doesn't seek to find other possible commands which create conflicts since the policy is to complain about only one conflicting command at a time. If any conflict cannot be found, control of the execution phase is passed on to the `Undo/Redo` layer.

7.3.5 Undo/Redo

The `Undo/Redo` layer copes with recovery from conflict-free commands that are filtered by `Conflict Detector`. It also stores necessary information for undo and redo requirements, which includes file versions, and updates the history and transaction lists. The `update` operation for undoing puts the related command in history lists in brackets and sets to 1 the undo status bit of the corresponding transactions in the transaction list. The `update` operation for redoing removes the brackets and sets the undo status bit of the transaction back to 0.

The `undo` of an active command is an operation which reverses all the transactions of that command. The `redo` of an inactive command is an operation which re-executes all the transactions of that command. A transaction is reversed or re-executed by a particular system call whose transactions do not have to be stored. The transaction list has a record of what kind of effects this system call is meant to cause.

During undoing a command, when `Undo/Redo` encounters one Create on a file in its transactions, one `Delete` is performed on that file and vice versa.

$$\text{Create file} \overset{undo}{\longleftrightarrow} \text{Delete file}$$

In order to implement the `Delete`, `Undo/Redo` cannot use an *unlink* system call because the `redo` command will require the file. Instead the *rename* system call must be issued which moves the file from the current state to the `.save` directory. The redoing of the `Create` (i.e. the undoing of the corresponding

`Delete`) is a matter of reversing this movement by a new *rename* that restores the relevant version of the file.

Undoing directory transactions is much easier. One `Make` on a directory is reversed by one `Remove` on that directory and vice versa.

Make directory $\overset{undo}{\longleftrightarrow}$ Remove directory

Path newdirectory $\overset{undo}{\longleftrightarrow}$ Path olddirectory

These effects are implemented by the *mkdir* and *rmdir* system calls. One `Path` is undone by another `Path`, using the *chdir* system call to change to new or old directories.

Changes to access permissions of files and directories are represented by `FPermit` and `DPermit`.

FPermit file $\overset{undo}{\longleftrightarrow}$ FPermit file

DPermit directory $\overset{undo}{\longleftrightarrow}$ DPermit directory

The `chmod` system call can undo or redo the effects of both FPermit and DPermit transactions.

Other transactions, `Read` and `SLink`, are not handled by the `undo` and `redo` commands, because they don't make any changes to the state. As seen in the previous section, they are important elements of the checking procedures applied for conflicts.

7.4 Shell Implementation

The `Shell` is the module of `brush` that interacts with the user directly and initiates the execution of user-oriented tasks. It is possible to use one of the existing shells to carry out the function of the module. This section discusses the usability of an existing shell that replaces the `Shell` and presents an implementation of such a use.

7.4.1 Developing a shell

In shell-based operating systems, shells are commonly used as command interpreters. A shell is a user interface that allows users to access operating system resources by running various commands. Likewise, `brush` is required to provide a new shell which cooperates with the `Recovery Manager` during both execution and undoing/redoing of commands. With the new shell, user commands, except `undo` and `redo` commands, are supposed to be interpreted in a similar way to those of an ordinary shell.

The `Shell` module of `brush` is still in its infancy. We wish to implement a *bash*-like shell. To achieve this goal, facilities that the readline library [38] presents would be of great use, e.g. smoothly reaching desired history lines and resubmitting those lines as well as calling `undo` and `redo` that are controlled by a single `Meta` character.

The goal might be achieved to some extent by tracing an existing shell, as in `Ufo` [2]. From the user's perspective, `Ufo` doesn't provide any new functionality. Since incorporating no new commands into the system, it doesn't need to totally keep the working environment under control. But it may not be possible that `brush` should be able to control the environment through the shell to the extent that it needs. There would be difficulties in working out issues such as definition of recovery commands and compatibility with different versions of a shell and so the use of such a shell would offer less portability.

In ordinary shell implementations, most user commands are executed with an *exec()* system call, which is invoked by a child process that the shell creates. Other commands (*built-in*s) are locally handled by the shell itself. In `brush`, therefore, it would be adequate that the `Tracer` monitors only child processes that call *exec()* and the `Recovery Manager` makes appropriate arrangements for these processes. Similarly, the `Shell` module should internally handle execution of and recovery from the built-ins as well, which don't require any information to be stored.

The implementation of `Shell` involves writing a substantial amount of code that is expected to provide at least known features of an existing shell. To show what a shell with undo support facility is like, we currently present a prototype implementation of `brush` below. The prototype takes advantage of utilities of a shell that is already in use.

7.4.2 A prototype implementation

Despite some difficulties, it is possible to revise `brush` to make it trace all existing shells, *csh*, *bash*, *ksh* and so on. Thus those shells may create an environment in which recovery operations are supported and the environment may provide all the features of the shell that it belongs to. The current implementation runs as a subshell under the standard shell (*sh*).

Initially, `brush` is connected to *sh*. It traces both *sh* and child processes initiated by *sh*. Each command that the user types in is served by *sh* itself. The prototype monitors all file accesses needed by the command and holds recovery information properly.

The prototype defines some additional commands and so it should check the command-line data to see if the command typed is one of those which are meaningful to itself, such as `undo` and `redo` commands. This can be achieved by using the *read* system call on the standard input performed by only the parent process (running *sh*). When the user submits one of these local commands, taking control of execution, the prototype stops the *read* on entry to the kernel. Then it reads the system call parameters and stores necessary information.

The *read* is also stopped on exit from the kernel to modify the argument that points to the command-line information. The argument is filled up with a null character, so that *sh* considers the information as if it consists only of a newline character and produces a new command line. With this approach, users can get the idea of what the shell with undo facility is like.

7.5 Optimization Issues

This section handles the optimisation of the `brush` components. Transactions that belong to each command are analysed in terms of their effects on the current state. The strategies which are adopted to store transactions and file versions optimally are discussed.

7.5.1 Optimizing transactions

The design has the potential to keep a non-trivial amount of transactions of the traced program whenever a system call of interest is intercepted. Programs such as the `Vi` editor, for example, tend to produce numerous transactions (in the case of `Vi`, all transactions are created by the *creat* system call). In order to throw these insignificant transactions away, we obviously want to discover which transactions are important and which ones are not. We also wish to keep out of concern transactions which depend entirely on unimportant ones.

Therefore, as soon as each user program terminates, several optimisations are applied to the system call transaction table before the contents of the table are stored as final transactions in the transaction list. The table may comprise a transaction that contains a `Create` and a `Delete` on a single file version. As an example, consider that the user edits a file named `fileA` from scratch via the `Vi editor`. During editing, if the user saves the file three times and quits the editor, the transaction table would consist of

```
1> Create fileA 1
2> Delete fileA 1
3> Create fileA 3
4> Delete fileA 3
5> Create fileA 5
```

As seen in the table, the transactions at times 2 and 4 can be considered to reverse the effects of the transactions at times 1 and 3, respectively. So these lines are removed from the table and the remaining transaction (5) is revised and held as follows.

```
1> Create fileA 1
```

The removal of transactions that undo each other leaves no side-effects on the current state.

The optimising algorithm basically sticks to the principle that transactions need only contain at most one `Read`, one `FPermit` and one `Create` or `Delete` for each file (see Section 7.1.1). Eliminating the pair of `Create` and `Delete` may affect an existing `Read` or `FPermit` on the same file version. As the second optimisation, if there is such a `Read` or `FPermit` in the transaction table, which always resides between a `Create` and a `Delete`, it is also removed from the table.

Thirdly, the table must be checked to see how many `Reads` or `FPermits` the transactions contain that are on a single file version. The third optimisation reduces the number of such `Reads` and `FPermits` to 1 each. For instance, a compiler that deals with the compilation of two or more source files at a time can make the transaction table hold this sort of transaction. Suppose that the `gcc` compiler deals with the source files `fileA.c` and `fileB.c` which locally reference a header file named `fileAB.h`, the version numbers of these three files are 10, 11, and 12, respectively. The table would probably have the following transactions

```
13> Read fileA.c 10
14> Read fileAB.h 12
15> Create fileA.o 15
16> Read fileB.c 11
17> Read fileAB.h 12
18> Create fileB.o 18
```

In this case, one of the transactions at times 14 and 17 is removed from the table. Note that the order of these transactions may vary from one version of the compiler to another.

The optimisation of `Reads` imposes some limitations on the implementation and the use of the `bmake` program. As far as discovering dependencies is concerned, it is the reason why `bmake` requires a compiler to compile only one file at a time (see Section 7.6).

On the other hand, the optimising algorithm is also applied to transactions on directories, even if a user command scarcely produces directory transactions which are unnecessary to store. As with transactions on files, the optimisation of directory transactions depends on the principle that transactions belonging to each directory contain at most one `DPermit` and one `Make` or `Remove`. Therefore, if any, the pair of `Make` and `Remove` on a single directory and all the other transactions which operate on that directory must be removed. Given that the transaction table includes, for example, the transactions

```
1> Make dirA 1
  ...
```

```
5> DPermit dirA 1 755 711
  ...
9> Remove dirA 1
```

Then the transactions at times 1 and 9, which perform opposite operations, will be eliminated. With this elimination, the transaction at time 5 seems to have operated on a directory that does not exist any more and thus it needs to be removed, also.

7.5.2 Optimising file versions

The optimised transactions can have caused new file versions to be held in the backup directory. During the optimisation, these versions normally need removing just as transactions associated with them are ignored. However, it is possible to keep intermediate versions as well and `brush` allows users to store them in a particular directory. This can be achieved by setting the appropriate flag at the user's disposal.

By an intermediate version, we refer to a file which is created and then deleted before the command ends. Since intermediate versions always reside in the `.save` directory they can be considered to be isolated from the current state. Like the eliminated transactions, they have no effect on undo and redo operations. A file version that survives the termination of the command forms an essential part of the state and recovery. Even if it is deleted by a later command, it can't be treated as an intermediate file.

While interacting with a program, users can reach previous versions of a file produced by the program and reload them for a new editing session, specifying proper pathnames (e.g. the `xfig` program and `emacs` editor allow this). However, there may be no way to take a previous version through the current session of some programs, because they may only allow control over the version of a file specified on the command-line. For this kind of program, users can see the relevant versions after terminating the program.

Programs can create various types of intermediate files. For instance, compilers behave differently from editors. The intermediate (temporary) files that they produce have no information important to the user. It would be undesirable to keep them and so `brush` directly removes them regardless of the request of users.

In addition to intermediate files, the backup directory can contain some insignificant files because of the possibility of storing the old versions of files that compilers bring up to date. Object files generated by a `C` compiler, for example, represent such files, which have the extension of ".o". On encountering a transaction that overwrites an object file `brush` does not store the old version of the object file.

In order to separate programs that produce crucial intermediate files from programs that don't, `brush` must use special recognition techniques. At first glance, it is easy to see that this separation can only made by looking at their types to accomplish a program-independent implementation. In the current implementation, `brush` does not takes the importance of the intermediate files into consideration.

There is another problem with executables. It may not be of interest to store the old version of an executable program that has been updated currently. However, it is not easy to recognize such programs. Users should specify the name of editors and compilers they use in `brush` before they are started. Alternatively, default names can be also defined.

7.6 `Bmake` Implementation

The `bmake` program is a tool that programmers can use with much less effort than the original Unix *make* program. It does the discovery of dependencies of a target, and the check of whether they are modified, automatically. This section introduces the issues that arise during the implementation of `bmake` and how they are resolved. Then it shows how `bmake` works by cooperating with `brush`.

7.6.1 Implementation issues

Our implementation aims to preserve the original behaviour of the *make* utility, but without requiring the provision of a `Makefile`. Compared to *make*, there is less burden put on programmers, reducing their involvement in the compiling process.

In the case of *make*, programmers use *Makefile* to specify how compilations are to be carried out. Our method expects programmers to start the compilation of each source file by typing the corresponding command line at least once. The most recent command line for a particular source file determines the way that *bmake* carries out later compilations.

`Brush` inherently associates a command line with the transactions that it executes. `Bmake` uses the transactions to determine dependencies. However, there are circumstances in which care needs to be taken. For example, suppose that a source file was last compiled by `bmake` itself, and that the execution of `bmake` involved several compilations. If all the transactions associated with the `bmake` execution are used to determine dependencies, too many dependencies will be found. To get around this problem, each compilation issued by `bmake` is stored as a separate command line in the `brush` history, as if it had been typed in by the user, and the appropriate transactions are associated with it.

The fact that compilation commands can be constructed in a wide variety of ways means that special attention has to be given to recompilations. For instance, a programmer can type in a command that produces several object files in one go. Care needs to be taken by `bmake` to compute the dependencies properly, and to ensure that the command is executed only once, no matter how many of the object files need to be recompiled.

Another issue concerns the order of recompilations. This causes no problem if an object file can always be produced independently of other object files. However, in many programming languages, modules must be compiled in dependency order. Thus `bmake` must carry out all required compilations in the right order, finally running the command that creates the executable.

The dependence on other files such as header files raises a new problem. If a source file is made dependent on a new header file, say, the new dependency is not among the ones computed by `bmake` from the last compilation command issued, so there is a danger of producing an object file with inconsistent contents. To discover this situation, the file is firstly compiled using the old dependencies, and then a comparison is made between the old and new dependencies. Then the proper action must be taken according to the new dependencies, perhaps involving compiling the file again.

A programmer who deals with the development of a program often encounters compilation errors. Most compilations generally start after opening all the input files for reading. Even if such a compilation fails to produce an object file, the transaction list contains all the dependencies that belong to the compilation and there is no problem. However, the compiler may fail before opening all the relevant input files, so that the dependencies computed from the compiling command are incomplete. This has no effect on the operation of `bmake`. The actions described in the previous paragraph also sort this problem out.

7.6.2 Running *bmake*

The fact that `brush` keeps past transactions makes the implementation of `bmake` fairly easy. Without the programmer needing to specify which compiler tools to use, it can deal with all compilers supported by the system. In other words, since transactions contain only the file system interactions, the implementation is compiler independent.

A compilation process can implicitly make use of many input files (dependencies). At first sight, it appears that dependencies can be determined from command lines. However, a command line does not include all the dependencies, for example it excludes files imported by source files, and in any case it may not be clear from a command line which arguments are actually file names. Thus, in `bmake`, all the input files required for an output file are taken from the transactions associated with running the compiler.

We assume that each object file created by a compiler needs all the files read by the compiler (regardless of whether they are read before or after the object file is opened for writing in the transaction list). This reveals an important requirement for source files which can be individually compiled. For `bmake` to discover dependencies that correspond only to a particular object file, programmers must type in a new command line for each such file.

Figure 7.6 indicates the relationship between the `bmake` and `brush`. Commands complete the process of compiling through a tiny tracing layer provided by `brush`. The layer, which is located between the system call interface and the kernel, catches file transactions and forwards the corresponding information (e.g. file name, file operation, etc.) to another (storing) layer. In the current implementation of `brush`, this information is stored in two separate file, namely *history file* and *transaction file*. Then the tracing layer communicates the pending system calls to the kernel without modifying them. Using the *history* and *transaction* files, it is possible to construct a *Makefile*-like file required by the original *make* utility. The `bmake` utility checks them for modifications to dependencies and resubmits appropriate commands.

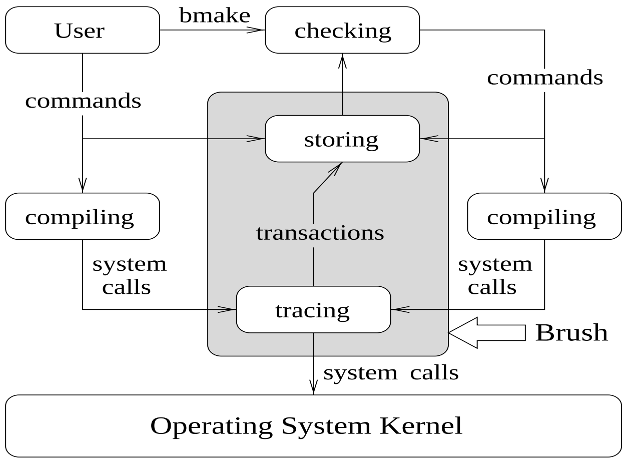

Figure 7.6: General structure of `bmake`

There is no difference between a single object file and the executable in terms of discovering dependencies. `Bmake` always takes the transactions of the last compilation associated with the object file or the executable into consideration so that it can check for modifications to dependencies properly. The desired comparison is made between these transactions and later ones, which includes both file names and version numbers.

As an example, assume that a programmer produces an object file named `fileA.o` by issuing

```
gcc -c -o fileA.o fileA.c
```

During the compilation, `bmake` ensures that all file transactions the compiler (`gcc`) makes which are associated with the programmer's own address space are stored. After carrying out some editing tasks, if the object file needs checking for recompilation, it is adequate to type in

```
bmake fileA.o
```

In this case, in order to complete the recompilation, `bmake` basically uses the following algorithm. Firstly, the transaction list is searched for the most recent transaction that involves a `Create` on `fileA.o`. Then `bmake` uses the history list to find out the command which created the relevant transaction. Using this command, it discovers the other transactions that were required to produce `fileA.o`. Of these transactions, ones involving files, including `fileA.c`, on which a `Read` is made determine the dependencies of `fileA.o`.

Now, `bmake` checks the transactions prior to the time when `fileA.o` is created, and sees whether the dependent files depend on some other files or not, using the above algorithm for each one. If there are more dependencies, the same algorithm must be used recursively for each of them and so on.

After discovering all the dependent files in this way, `bmake` needs to check for any changes made to them. For each dependent file, the checking process covers the transactions that reside between its last creation time before the creation of `fileA.o` and the current time. If a later transaction has affected at least one of the dependent files, specifically one transaction refers to a `Create` or a `Delete` or a `Utime` on a dependent file, `fileA.c` should be recompiled.

The editing tasks on `fileA.c` could cause the recompilation to create transactions which are different from ones created by the previous compilation of the same file, and thus to use different dependencies. Therefore, the `bmake` compares these two compilations to be able to discover new dependencies. Each new dependency can require a new recompilation of `fileA.c` after it is checked with the algorithm.

Compilation commands can make the steps involved in this algorithm quite complicated. For example, if the following linking command was initially submitted:

```
gcc -o program fileA.o fileB.o ...
```

the programmer can use `bmake` to update the executable program.

```
bmake program
```

This starts the checking procedure for each object file, in the same way as with `fileA.o`, and ends with a new linking command to create a new `program` file.

Chapter 8

Concurrency

The Unix system model is based upon the two central concepts, files and processes. Files are managed by the file system, and processes are controlled by user commands. The use of processes is the only way that is presented to manipulate files. For any file operation, processes communicate with the file system through the kernel via a specific set of system calls. Subject to their use of system calls, they execute independently of each other and share system resources arbitrarily. The Unix system allows processes to be able to access shared data concurrently, which can cause non-determinism. In order to reduce the usual non-determinism problems caused by uncontrolled sharing of resources to a minimum, `brush` has to handle concurrency in a more sophisticated way than ordinary shells. Recovery requires the activities of processes to be carried out in a sequence controlled by `brush`.

This chapter aims to point out how concurrency and multiple users can complicate recovery mechanisms. It presents the relevant concepts for a multi-user undo facility, including users' and the system's requirements that it must meet. Several existing multi-user undo models provided in various applications are examined in their own contexts. The rest of the chapter deals with the extension of `brush` to support concurrency and multi-user issues. In this way, the problems that concurrency imposes on the current implementation of `brush` are discussed and some remedies are given. The degree that `brush` can allow a multi-user interaction is also analysed.

8.1 Multi-User Undo Concepts

In this section, we give general concepts of undo for multiple users. First, the functionality of multi-user undo and users' expectations of it are expressed. Then a number of methods which can be used to achieve undo safely in multi-user interactive systems are described.

8.1.1 Introduction

Like single-user undo examined in the previous chapters, multi-user undo is an important and useful feature in interactive multi-user applications. One reason is that having an ability to undo in a group environment may provide freedom to interact and experiment in a shared workspace. However, it is more difficult to control the interaction in systems with multi-user undo where other users may have performed actions on a shared object between the times when a user performs an incorrect action and then decides to undo it.

Multi-user undo/redo facilities are provided by very few multi-user programs. Many multi-user applications do not offer this feature, although it is crucial to users working in a group setting. Choudhary and Dewan [24] give the following reasons for this.

(1) Features available to users in the single-user case must also be available in the multi-user case.

(2) In the multi-user case, the potential cost of an individual user's mistake is multiplied many times because it can adversely affect the work of a large number of collaborative users.

(3) In a collaborative setting, the number of alternatives to be explored increases due to the presence of many users.

(4) Users of a multi-user program can make new kinds of errors by sharing the results of their commands with the wrong subset of users.

Compared to single-user applications, performing undo in multi-user applications provides technical challenges in the following areas: *selecting* the operation to be undone, determining *what* operation will result in a correct undo, and dealing with *dependencies* between different users' operations [73]. First, in a multi-user environment, there may be parallel streams of activities from different users. When work on a shared environment occurs in parallel, users usually expect an undo to reverse their own last operation rather than the globally-last operation, which may belong to another user. Some systems, e.g. groupware systems, need to allow selection of the operation to undo based on who performed it. Second, once the correct operation to be undone is selected, the operation to execute to effect an undo has to be determined. Simply executing the inverse of the operation to be undone may not work because of modifications done by other users to the environment. Finally, if multiple users interleave their work in the same region of an environment, it may not be possible to undo one user's changes without undoing some of the other users' changes. In this case, there are dependencies between the changes which need to be taken into account during an undo.

8.1.2 Functionality

Multi-user undo especially is implemented in several groupware applications which are built to support multi-user work on a shared document, including SASSE [7], Grove [32], DistEdit [52], CVS [59], and MACE [67]. These applications usually provide a global undo facility rather than a per-user undo facility. Supporting a simple form of per-user undo, MACE allows users to undo their own modifications made to a section provided they acquire a lock on the section prior to making modifications and do not release the lock prior to the undo.

In a preliminary characterisation of the design space of collaborative applications based on the notion of generalised multiuser editing., Dewan et al [30] defined multiuser undo/redo as a crucial collaboration function. A multiuser undo/redo model allows users to cancel/repeat previous actions for three main reasons:

- *Recovering from errors:* Undo/redo can let users conveniently recover from erroneous editing changes made by them. In addition, it can allow them to also recover from errors in specification of coupling, concurrency control, and access control. Moreover, it can let them recover from errors made by others. Finally, multiuser undo/redo can allow a system to recover from errors it makes in broadcasting editing changes. If the system does not use atomic broadcast, then the order in which editing actions are made on each user's display may be different. Often, the physical order of these actions does not affect the resulting display. However, when it does, the system can use the undo/redo mechanism to reorder the commands to ensure consistency.

- *Exploring alternatives:* Undo/redo can let users individually explore different alternatives. In addition, it can let them explore alternatives collectively.
- *Newcomers:* An undo/redo facility can also help a newcomer study/replay a log to understand the sequence of steps that lead to the current edited state.

8.1.3 Requirements

There are several requirements that a general multi-user undo/redo model must satisfy. It must be compatible with an existing single-user undo/redo model, give individual users autonomy in executing undo/redo commands, support undo/redo of remote commands and the remote effects of local commands, be independent of the coupling, multicast, and concurrency control model, and allow undo/redo of arbitrary commands. These requirements are highlighted by Choudhary and Dewan [25] as follows:

> **Compatibility:** Multi-user undo/redo should behave like undo/redo of a single user when only one user is interacting with the system, thereby reducing the overhead of learning a new undo/redo model, which in turn hinders usability. This is especially true for a feature such as undo/redo that is expected to be used when the user makes errors and is thus not in a relaxed frame of mind to try new things. Compatibility allows the user to incrementally learn the new features of a multi-user undo/redo model.
>
> **Independence:** Multi-user undo/redo model should not require attention or intervention of all users in a collaborative session. This allows a number of users to participate in a collaborative interaction without being

overloaded with the requirement of synchronising with other users their use of recovery commands. It also allows each user to concentrate on the work that is individually performed.

Semantic Consistency: When multiple users interact with an application simultaneously, a user's actions can affect other users. The undo/redo model must ensure that the effect of undoing a command is equivalent to the state the system would be in if the undone command had never been executed, thus guaranteeing that the degree of consistency ensured by the concurrency control policy adopted by an application is not compromised by the use of undo/redo.

Collaboration: The model must allow users to undo/redo commands issued by other users, thereby allowing the users to collaborate on their undo/redo. This is a direct analogy of the ability of users to collaborate by executing 'do' commands on behalf of other users. The model must also allow users to undo the remote effects of commands they issued, thereby allowing them to recover from mistakes made in their collaboration. This follows from the general semantic consistency requirement given above.

Genericity: The model should be generic enough to allow its use in a number of applications with a variety of coupling, concurrency control, and broadcast schemes, thereby promoting uniformity and automation.

Undo of Arbitrary Commands: The model must not be restricted to commands that manipulate the user-interface state. It must also support undoing of collaboration commands.

8.1.4 Methods for multi-user undo

There are various methods which ensure secure user interactions as a result of preventing different users from updating the same object. These methods also help keep system components consistent, controlling users' actions and increasing the functionality of multi-user undo. Below, we will describe four of them. More detailed information can be found in [1].

Locking: This method ensures that each system component or application object can only be updated by one user. The most prevalent form of lock is the *explicit lock*. A user obtains a lock on a section of a document, or a whole file, performs updates and then releases the lock. During the period the lock is held, other users cannot update the same section, but they may be able to look at it. If large sections are locked for long periods of time, however, collaboration becomes difficult. Systems which want to encourage low granularity cooperation, therefore, try to reduce the granularity of locking.

Implicit locks tend to operate at a much finer granularity. When a user starts to edit some portion of a document (paragraph or sentence), the system implicitly obtains a lock. When the user moves elsewhere or after a timeout (a short period with no typing), the lock is automatically released. Thus the users are unaware that there is any lock in operation unless they attempt to edit the same area simultaneously.

Roles: Some group authoring systems assign *roles* to users with respect to each object in the system. Depending on their roles, users may be able to read, write or add annotations to the objects. This is similar to the traditional idea of file ownership, where the file's owner has greater rights than others. The restricted access is not sufficient on its own to prevent contention, as often several users still have write access to an object. However, by making the rights of different users explicit in the system and to the participants, the likelihood of clashes which prevent undo is reduced.

Social protocols: Users cooperating over the use of resources often develop social protocols to prevent clashes whilst doing things to the shared objects. One of the most common dynamic social protocols is a simple baton passing protocol. One user creates a first draft of a document. This is then passed on to the second user who does some changes, who then passes it on to the next user, and so on. These baton passing protocols have been built into some groupware systems. Their importance is that they ensure a single thread of updates and thus make the meaning of undo clear. Within a single user's 'go' at the document, any action may as easily be undone as in a single user application. The user may even go on to undo the updates of the previous user.

Copying: Most traditional single-user applications do not act on the system's data store directly, but make copies of the data object (the document or program, for example), manipulate the copy, and then save the updated object. The copy is private to the application and thus there is no possibility of concurrent access. All commands internal to the application commute with all similar commands from other applications. In such a program, it is possible to undo the internal actions, but not the external ones. Such a system may be regarded as having two sorts of operations: **U** commands, such as editing the text, recalculating a spreadsheet, which are undoable; and **non-U** commands such as saving a file, which cannot be undone.

As with locking and role assignment, one has the problem that either the granularity is large, preventing cooperation at a fine scale, or the granularity is small, but the system cannot undo beyond these small boundaries. Copying without locking has an added problem that users may edit copies of the same object concurrently. This results in multiple versions which must be merged. Some form of modified version control system is required to deal with this resynchronisation problem.

8.2 Multi-User Undo Models

Multiuser undo/redo has been examined by several works. These works address many different issues arising in the design of multi-user undo/redo. This section presents a number of existing multi-user undo/redo models, as well as some of their related design and implementation issues.

8.2.1 General model

The general multi-user undo/redo model [25] is a simple extension of a linear single-user undo/redo model. To build the command histories of users, all commands are shared by every user and commands appear in the same sequence in all command histories. An undo/redo command can be executed by any user and causes the last command in every history to be undone/redone.

More formally, the model provides four operators *execute*, *undo*, *redo*, and *skip* which operate upon command lists. Let CL_i be the command list associated with the ith user interacting with a program. Let $CL = c_1, ..., c_{n-1}, c_n$. Then *execute*($CL$, c_{new}) sets

$CL = c_1, ..., c_{n-1}, c_n, c_{new}$, $CL_i = CL$ and $CurrentCommand_i = c_{new}$

for each user i. $undo_i$ executed by the ith user uses the same rule as the single user model for determining the target command. Thus $undo_i(CL = c_1, ..., c_{n-1}, c_n)$ results in setting $CurrentCommand_i = c_{n-1}$ and $c_n.status = Undone$ for each user i. The redo and skip commands can be similarly defined.

Choudhary and Dewan divided the undo/redo model into two submodels and developed first-cut versions of them. The semantic model determines how command histories of the users of a multi-user program are constructed, which commands are undone/redone by an undo/redo request from a particular user, and which user can undo/redo a command. It constructs the command history of a particular user by combining all commands (both local and remote commands), whose results were made visible to that user. It allows a user to undo/redo arbitrary commands in a command history including commands executed by other users.

The implementation model offers programmers a framework for implementing the semantic model given above for a particular multi-user program. It divides the task of implementing undo/redo between generic dialogue managers provided by the system and application programs written by programmers. The model classifies application programs into three increasingly complex classes according to how they respond to user commands; undo unaware programs, state caching programs, and undo aware programs. It requires increasing levels of undo/redo awareness from these classes of programs.

The general multiuser undo/redo model also deals with problems that arise in the communication of user's undo/redo operations to other users, which can be described as follows:

Corresponding commands: Some systems allow simultaneous execution of commands by multiple users. When two commands are executed by different users, such systems cannot guarantee that they are received in the same order by all users. Thus execution of commands c_p and c_q by other users may be received by users i and j in different order resulting in $CL_i \neq CL_j$. An undo/redo invocation may operate on different commands in different command lists, since the last command in different lists is not guaranteed to be the same. To solve this problem, the model provides a way to identify all copies of a command in different command histories with a unique label. If the last command in a local command history is undone, a request is sent to all other users to undo the *corresponding* command in all the other command histories.

Undo by reference: Consider a system that also allows execution of undo/redo commands concurrently with other command execution. In such a system, the undoing of the last command in the command history is requested, between the time the decision to invoke undo is reached and the undo is invoked, a new command may be executed by another user and become the last command in the command history, which causes the undo to operate inadvertently. This problem is rectified by allowing a user to explicitly mark the command to be undone. At most one command can be marked by a user. When an undo/redo command is invoked, it checks if a command in the command list has been marked. If a command is marked, that command is undone/redone. Otherwise, as before, the last command is undone/redone. Even if the referenced command is not the last command, it is undone, first undoing all intervening commands in the command list and then restoring them to their previous status.

Selective command sharing: Consider a system that provides non-WYSIWIS (What You See Is What I See) coupling[1]. In these systems, inserting all commands in every history is not applicable, since users expect that their command histories should be the state of their user interfaces. Therefore, the model inserts into users' command histories only those commands that are shared with the users. More precisely, it introduces the concept of a *sharing* function. Given two users i and j and a command c_n executed by user i, the sharing function determines if the command will be (immediately) executed by the user-interface of user j. Different sharing functions can represent different collaboration schemes. When user i executes the command c_n, it is inserted in CL_j for each user if and only if *shared*(i, j, c_n) returns *true.*

This model is applicable to multi-user programs offering a variety of functionality, coupling, concurrency control, and multicast schemes. In particular, it is applicable to multi-user text/graphics editors, spreadsheets, mail programs, and code inspectors; coupling schemes offering WYSIWIS and non-WYSIWIS interaction; concurrency control schemes; and multicast schemes supporting

[1]A system with non-WYSIWIS coupling does not require the results of all commands to be shared immediately with all users.

both atomic and non-atomic multicast.

8.2.2 Handle space model

The handle space model [1] is defined in the context of a shared or group editor. It assumes that the group editor is effectively centralised and that the effects of one user's actions are instantly available to all other users. This means that it considers problems of interleaving, but not true concurrency. The overall user history is an interleaving of commands issued by each user. There are two options for multi-user undo:

local undo, which operates on the user's own actions

global undo, which operates on the combined actions of all users

A system can support one or all of these undo capabilities. To make the system's interpretation of global undo and local undo clearer, consider the following sequence of commands issued by two users: The first user, *user1*, issues two `insert` commands which insert the text "hello" and "world" into a document. The second user, *user2*, issues a `select` command which selects the text "hello" and a `delete` command which deletes it. Now, the document has only the text "world". If either user executes a global undo, the text "hello" comes back to the document and remains selected by *user2*. For a local undo issued by *user1*, the text "world" is removed, resulting an empty document. A local undo issued by *user2* has the same effect as the global undo.

The global undo appears easier to implement from the system perspective. It does not matter to the system how many users are issuing commands because the script editing commands behave exactly as they do in the single user case. In the local undo case, the semantics of the script editing commands are complicated because the system must keep track of which user issues which commands. The model helps resolve the conflict between the system's preference for global undo and the user's preference for local undo.

Formally, the script model is described as follows: The set of possible user commands is denoted by the set C. In the multi-window work, each command is tagged with a *handle* which identifies the window to which the command is issued by the user. In the multi-user case, the handle indicates which user issued the command. The set of user handles is denoted by U. A history of user actions, denoted by H, is a sequence of pairs from U x C.

$$H \;=\; (U \text{ x } C)^*$$

In the above example, a valid history would have been

$$h = \{(\mathit{user1},\ \mathit{ins}(\mathit{hello})),\ (\mathit{user1},\ \mathit{ins}(\mathit{world})),\ (\mathit{user2},\ \mathit{sel}(\mathit{hello}))\}$$

The set of possible system states is denoted by S. Beginning with any initial state of the system, s_0, the current state is obtained using a state transition function *doit*.

$$\mathit{doit} : (S \text{ x } H) \rightarrow S$$

With the history h given above and initial state s_0 being the empty document the system would have

$$\mathit{doit}(s_0,\ h) = \boxed{\text{hello}}_2\ \mathit{world}\,|_1$$

The subscripts attached to the cursor and selection box above indicates that the system state knows the ownership.

Each user sees only a part of the whole state of the system. This is denoted by a display function for each user $\mathit{display}_u$ where u is the specific user's tag. The display functions map from the system state S to some different set, say D.

$$\forall\, u \in U \bullet \mathit{display}_u : S \rightarrow D$$

The earlier examples did not distinguish the current state from the users' displays. In a large document, these would be some small part of the document which would fit on the screen, and may be (depending on the form of groupware) different. For example, given the history h, the selection of text by the second user would not be visible to the first.

$$\mathit{display}_{user1}(\mathit{doit}(s_0,\ h)) = \mathit{hello\ world}\,|$$
$$\mathit{display}_{user2}(\mathit{doit}(s_0,\ h)) = \boxed{\text{hello}}\ \mathit{world}$$

As well as identifying the individual views, the model distinguishes that part of the state which corresponds to the permanent result of the interaction. In the example, this *result* function would extract the contents of the document but would discard the selection and insertion point information. Since the model is dealing with a shared document, the result is not local to the user, and so it does not represent individual result mappings as it did for the display. The result is, therefore, a mapping from the system state S to the result set R.

$$result\ :\ S\ \rightarrow\ R$$
$$result(doit(s_0,\ h)) = hello\,world$$

8.2.3 Script model

This model [72], based on history undo (see Chapter 3), is presented to provide undo facilities in groupware applications. It allows operations to be undone selectively and deal explicitly with location shifting and dependencies among users' operations. This has very similar characteristics to the handle space model, from a different point of view.

The model uses data structures similar to those used in history undo; in particular, upon an undo, the inverse of an operation is appended to the end of the history list. However, in a groupware application, since the last operation done by a user is not necessarily *globally* last (other users may have done operations subsequently), the model needs to allow undoing of a *particular user's* last operation from the history list. For example, consider the following history list, whether A_i's refer to operations done by one user, say Ann, and B_i's refer to operations done by other users:

$$A_1\ \ B_1\ \ A_2\ \ B_2\ \ B_3$$

Now, suppose Ann wishes to undo her last operation, A_2. One way is to undo the last three operations and then redo B_2 and B_3, but that can be disconcerting to other users of the system and may not even be correct if there are dependencies between A_2 and B_2/B_3. Note that Ann may not be aware that operations B_2 and B_3 have been carried out on the document, and other users may not aware of changes made by Ann. The model allows being able to undo A_2 without undoing/redoing B_2 and B_3.

In the above example, the operation to be undone, A_2, is selected based on the identity of the user. More generally, the operation to undo could be selected based on any other attribute, such as region, time, or anything else. To allow such selection, each operation on the history list is *tagged* with appropriate selection attributes, such as user id and time of the operation. The operation to be undone is not necessarily the last one, but is selected using some attributes attached to the operation.

The model also provides support for both local undo and global undo. These undos operate normally on the last action of the interaction from the user's and system's perspective. An important issue in the design of group undo facilities is whether global undo is useful, because it can often be confusing for users. A user does not generally know what the globally last action is. Moreover, in the case of two actions done by two users in their own editors, no particular action can be guaranteed to be globally last. However, some scenarios where global undo is of some use and having predictable effect are:

- When the group as a whole wishes to go to a known earlier state of a document by discarding all recent changes, irrespective of who made it.
- When a group is working with synchronised views (effectively as one person) – everyone is looking at the same part of the document, changes are being made one at a time, and everyone is known to be looking at everyone's changes.

A reasonable way to handle such scenarios is to provide a special editing mode where a user acquires the undo rights of an entire group. For instance, it may be sufficient to provide a single undo button that does local undo for operations executed in the normal editing mode and does global undo for operations executed in a *lockstep* editing mode.

8.2.4 Direct selective model

The model [10] is presented in the GINA application framework [81] which is based on the idea to represent the interaction history as a tree of command objects in graphical interfaces. It uses a form of the history mechanism of the framework that is extended for multi-user applications. Multi-user applications allow spatially distributed users to work coupled or decoupled on a document.

It is assumed that the participants can communicate informally through another medium, such as an audio/video link.

Undo in multi-user applications may have different meanings. The normal undo function (global undo) goes one step back in the command history. If the users do not operate exactly synchronously, undo may also mean "undo my command" because users take only their personal actions into account. However, one user's last command may no longer be the last one in the history if another one has also submitted a command. The normal undo is not able to undo one without the other.

The problem is clarified below. The star points to the last command currently in effect. User `B` has submitted a command between two commands of user `A` (on the left). Suppose that `B` wants to undo this command. The normal undo facility cannot be used, because it would also undo the second command of user `A` (on the right).

$A \;\; B \;\; A_\star \qquad\qquad A_\star \;\; B \;\; A$

GINA provides a *selective undo* [11] that creates a copy of the command to undo, but with the inverse effect. Selective undo may be impossible because later commands have destroyed a prerequisite of the command. There is nothing the system can do about this. The users can find a solution for this conflict in a cooperative manner.

To solve the above problem, user `B` can use the menu entry "undo my command". This operation is internally implemented with selective undo and leads to the following history, where a command with an inverse effect of `B`'s command has been appended to the history.

$A \;\; B \;\; A \;\; B_\star^{-1}$

An open question is how subsequent calls to "undo my command" should be interpreted. Selective undo creates a new command, which becomes the last command issued by user `B`. A simple interpretation of "undo my command" would now reverse that command, so the user can only undo and redo the last command. Subsequent calls to "undo my command" might also undo the last command of the user that hasn't been undone yet. This would establish unlimited linear undo on the subset of commands issued by one user. The implementation is straightforward by providing every command created by selective undo with a reference to the command it reverted. The next command to be selectively undone can be found by searching backwards in the history beginning with the command preceding the referenced command.

Almost all the models depicted above are defined in multi-user text or graphical editors. The nature of operating system shells is somewhat different. The `brush` shell considers any user's command to operate on the entire contents of a file, because it handles the system's state rather than a file's state. Thus, we find it sufficient to provide only one kind of undo (local undo) in the multi-user setting. Users work on their own spaces allocated by the system. With concurrency, each user can submit commands running simultaneously, which involves modifying some policies adopted in `brush`. This issue will be addressed in the subsequent sections.

On the other hand, users can be allowed to work under a shared directory or on shared files, e.g. a couple of users can cooperate in a program development environment to build the modules of large programs. In this case, a user's command will probably depend upon that of another user and all users' commands as a whole will establish a consistent state. Recovery operations cannot be guaranteed to function correctly. An undo command issued by a particular user can create an inconsistent state. Furthermore, concurrency can cause a similar situation, allowing commands of different users to interleave and so to produce non-determinism. To avoid such situations, `brush` needs to put some restrictions on shared interactions, as will be seen in Section 8.5.

8.3 Concurrent Transactions

The Unix system is an operating system in which multiple users and processes are served concurrently. It can encourage processes to interfere with each other, generating a result of leaving the file system inconsistent. So careful attention must be paid to the granularity of concurrency for purposes of recovery. This section highlights problems that concurrent execution adds into the recovery situation. The problems are characterised by several classic examples and various ways to handle them are introduced.

8.3.1 General consideration

Concurrency causes some problems in determining the state of a system at a particular time correctly. The receipt of one system call may affect the response to a subsequent system call. In these situations, a system call received alters the state of the system, so the system must be evaluated in a single sequence of system calls. The state must be threaded through a sequential evaluation, the order of which is determined at run-time by the order of events, or rather the order of system calls received from concurrent processes. It doesn't matter when processes are initiated, because a process is not of interest until it makes a file-related system call.

Multi-processing operating systems generally serve processes in an event-processing loop. Each scheduled process runs during the time that is assigned by the kernel. The `Tracer` mechanism of `brush` *poll*s for events relating to the file system, and dispatches system calls to the `Recovery Manager` which then converts them to a series of transactions. When a system call issued by a process is stopped and forwarded to the `Tracer`, the operating system continues to serve other current processes. However, it will not service any other file request until the previously-stopped system call is resumed to run, in that such a request needs to be suspended and handled as well. This means that dealing with a selected system call in `brush` does not give any rise to non-determinism. The fact that the `Tracer` is waiting for a system call of interest that is performed by any running process involves associating the transactions which have been generated with the related processes.

For the most part, concurrency imposes on `brush` the burden of specifying the order in which events will occur, making it necessary to analyse pending transactions to determine the sequence of those which belong to a particular process. Therefore, we need more control over the traced system calls. The prototype of `brush` that has been implemented in the previous chapters can exhibit non-determinism, because program behaviour may depend on the accidental relative timings of system calls. Even if programs behave incorrectly, this can be discovered in `brush`, but the non-determinism can still affect the possibility of recovery, which will be discussed later.

8.3.2 Transaction-based interaction

On Unix systems, the execution of user processes is divided into two levels: user and kernel. When a process executes a system call, the execution mode of the process changes from *user mode* to *kernel mode*: the operating system executes and attempts to service the user request. Several processes may all run simultaneously in either mode. This does not influence our user-level implementation. The `Tracer` mechanism of `brush` can continue to deal with the user's own processes only in user mode. The simultaneous execution of processes in kernel mode is kept out of concern. Once the `Tracer` instructs the operating system to stop the file system calls on entry to the kernel, it is possible to predict how processes will behave in kernel mode, because the stopped system calls bear the complete knowledge of what the calling processes is to do in kernel mode.

System calls are the only mediator that causes a communication line to be created between user-level processes and the kernel. From the kernel's point of view, user programs consist of a particular set of system calls. Once the communication line is established by issuing a system call, the kernel handles the execution of the system call and returns a response to the calling process. If the system call is file-related, for example, the response is a confirmation that either the required modification is carried out on the file system or the requested information is made ready for the use of the process. At the kernel level each system call is executed as a whole, without being split in some other segments.

However, as seen in Chapter 6, we have divided system calls into a few suboperations, named *transactions*. The effects of user commands have been analysed through these transactions to build the recovery mechanism. One transaction can point out a small part or all of the effects that one system call is to have on the file system. Interactions that processes set up with the kernel, thus, can be assumed to be transaction-based (see Figure 8.1). In this way, a process that issues, for instance, one *unlink* system call will behave as if it created one `Delete` transaction. In our context, transactions of one system call cannot be split further.

The assumption that the communication between user processes and the kernel is based on transactions provides a way of analysing the granularity of concurrency in terms of recovery. It reduces the number of interleavings that come with concurrency to a reasonable amount because of the fact that most programs perform only a few transactions of interest. `Brush` acts as a shell that controls the flow of system calls into the kernel and converts them into transactions. While examining concurrency in the later sections, we will suppose that processes interact with the operating system kernel via transactions, rather than system calls.

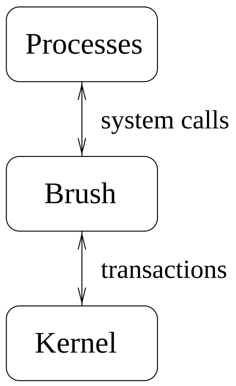

Figure 8.1: Pseudo transaction-based interaction

The way in which system calls are converted to proper transactions does not lead to fixed transactions. The number of transactions that belong to a system call varies according to the current state. Each group of transactions per system call must be considered in their entirety and transactions of each group must be stored in the right order. For example, if an existing file, `fileA`, whose version number is 5, is modified by a command, the following transactions are stored:

```
10> Delete fileA 5
11> Create fileA 11
```

These transactions do not interleave with any other transaction in concurrent execution in that they are produced in response to a request of the command via one system call. Note that the execution of commands can get involved in many groups of transactions in accordance with the type of system calls.

8.3.3 Transaction interleaving

When two or more processes execute concurrently, their file system transactions execute in an *interleaved* fashion. That is, transactions from one program may execute in between two transactions from another program. This interleaving can cause programs to behave incorrectly, or *interfere*, thereby leading to an inconsistent file system. This interference is entirely due to the interleaving, i.e. it can occur even if each program is coded correctly, and no component of the system fails. To understand how programs can interfere with each other, we will look at some examples of commands that are issued at the command-line.

In the following examples, we will also see how the command-line structure can complicate the recovery situation. For clarity, the Unix *cp* command is used[2]. Note that `fileA`, `fileB`, and `fileC` are already existing files whose current version numbers are, say, 2, 4, and 6, respectively.

Case 1: Processes which are started from a single command-line can run in isolation, without interfering with each other. Consider the following compound command:

```
$> cp fileA fileB & cp fileC fileD
```

where two concurrent processes are created in order to execute the command. Each process operates on the different components of the file system. As they do not overlap, the resulting state is always deterministic. On the other hand, the execution of the command can result in the interleaving of the transactions. For example, the order of the transactions that have been recorded in the transaction table can be like this:

```
10> Read fileA 2
11> Read fileC 6
12> Delete fileB 4
13> Create fileB 13
14> Create fileD 14
```

[2]The reason why we use the *cp* command to illustrate the problems is that it issues two file-related system calls of interest. It is the most convenient command-line command to show the interleavings.

where the transactions at times 10, 12, and 13 represent the first component of the compound command, ones at times 11 and 14 do the second component of the command.

This kind of concurrent execution does not create any problem, and so it does not need controlling. With the current implementation of `brush`, the transactions of the compound command can be undone easily, regardless of whichever process performed them.

Case 2: Processes which are started from multiple command-lines can run in isolation, without interfering with each other. Suppose the user executes the following two commands consecutively:

```
$> cp fileA fileB &
$> cp fileC fileD
```

In the execution of this sequence of commands, it is also possible to observe the interleaving of the transactions, especially when the files to copy are large. As an example, let's take the following situation that may be created by the commands.

```
10> Read fileA 2
11> Read fileC 6
12> Create fileD 12
13> Delete fileB 4
14> Create fileB 14
```

It is impossible that these transactions are shared between the commands in a command-independent way. Undo needs to know the transactions which belong to each command-line so that it can deal only with them. Consequently, the concurrent execution forces `brush` to determine which process produced which transactions. Although, from the above transactions, the first command appears to have ended before the second, the history list must be constructed according to the order of submission of the commands.

Case 3: Processes which are started from a single command-line can run in an interfering way. Assume that the following compound command is issued:

```
$> cp fileA fileB & cp fileB fileC
```

where the two processes created attempt to act on the same file (`fileB`; one is writing to it, the other is reading from it. This execution can easily result in the interference of the processes. Suppose a user has the following example transactions:

```
10> Read fileA 2
11> Read fileB 4
12> Delete fileB 4
13> Create fileB 13
14> Create fileC 14
```

The transactions show that the second component of the compound command has taken the previous version of `fileB`, not the current version, despite the fact that the user could probably expect the component to take the new version. This is caused by the process control system which constructs the interleaving by scheduling the processes for execution. Our design cannot ensure that files are passed with the correct contents to commands. Instead it concentrates on serialising file accesses. In this way, `brush` must convert concurrent accesses to a single file into serial accesses, so that file consistency can be preserved. From two processes accessing the same file concurrently, this means that one must be delayed until the other is finished with the file. As specified above, control of accesses to different files is not of concern.

On the other hand, it may be misleading to derive the behaviour of commands from their transactions. Note that the `Create` transaction at time 13 denotes only the creation time of `fileB`. The creation of the file with full contents may have occurred after the creation of `fileC` at time 14 (see Chapter 6). The same might be true for the other types of transactions. For example, the `Read` transaction at time 11 above might not signify the reading of the whole contents of `fileB`. The complete reading of it may have been accomplished only after a few later transactions. Thus, the user must keep in mind that transactions in this thesis are defined to represent the effects of commands, rather than their equivalent behaviour.

Case 4: Processes which are started from multiple command-lines can run in an interfering way. Consider the following two commands executed consecutively:

```
$> cp fileA fileB &
$> cp fileB fileC
```

The result of this execution may also contain the interference. For instance, the concurrent execution of these two commands might lead to the following execution of transactions:

```
10> Read fileA 2
11> Delete fileB 4
12> Create fileB 12
13> Read fileB 12
14> Create fileC 14
```

where it seems that the transactions do not interleave. Nonetheless, both commands might operate on `fileB`. That is, when the second command opens `fileB` for reading, the first command might not have completed the copying of `fileA` into it. This situation can be prevented by concurrency control, in which case the `read` and `write` operations on `fileB` are performed in a serial order.

In the preceding examples, the problems such as interference were caused by the interleaved execution of transactions from different commands performed by different processes. Similar problems can arise when a process creates many child processes and runs concurrently with them, too. To avoid interference and thereby avoid complications is the goal of concurrency control.

An interleaving comes into existence because the system executes different parts of a program at possibly overlapping times. The examples given above do not exhaust all possible ways that concurrently executing transactions can interleave (or interfere), but they do illustrate a few problems that can make recovery difficult, and emphasizes the importance of controlling the kinds of interleavings between transactions. For simplicity, the problems can be confined to the following two situations:

1. Concurrent access to files
2. Interleaving of transactions

A concurrent execution is required to leave the system recoverable under all interleavings. Recoverability does not depend ideally on the correctness of programs. Nevertheless, it may be affected by their incorrect executions. In other words, even if programs are correct individually, concurrency does not guarantee their correct execution and their recoverability, because it can force them to make simultaneous accesses to the file system at inappropriate times. Accordingly, the correctness of a program does not mean that the program will always behave correctly during its concurrent execution with other programs. It can be simply destroyed by the system which executes the program.

8.3.4 Storing concurrent transactions

In `brush`, the way in which command transactions are stored is primitive. Currently, transactions of a process in execution are kept in the transaction table temporarily, and whenever the process terminates the transactions are optimised and then stored in the transaction list permanently. `Brush` assumes that only one parent process[3], which can have child processes concurrently running with it, is in execution during a small cycle of the interaction, i.e. the creation of each parent process follows the termination of the previous one. The parent process's own transactions can easily interleave with those of its child processes. This does not introduce complications because all the transactions are bound to the parent process, as long as their potential concurrent accesses to files are controlled properly. An undo command is not allowed to reverse the transactions of a particular child process separately.

Conventionally, undo deals with the rolling-back of parent processes. The rollback of a process means the reversal of the effects of it and its child processes. Since concurrent executions of parent processes result in concurrent transactions, some complications can come into existence, as specified earlier, which affects the way in which transactions are stored in `brush`. For example, when there are two or more parent processes in execution at the same time, it is necessary that transactions which have been created must be distributed among them. To achieve the distribution, there is need to identify the relationships between parent processes and other processes (child, grandchild, etc.). In the implementation, this can be met by keeping lists which contains the *id*s of

[3]A parent process refers to the process which runs one complete command or the controlling component of a compound command typed in the command-line.

all the subprocesses of each parent process.

Concurrent transactions need to be stored in a convenient way so that parent processes can be rolled back one at a time efficiently. At first sight, it seems possible to store transactions of concurrent processes after any one of the following four situations occurs. However, the erratic behaviour of processes can make it inefficient to use them.

(1) Termination of all currently-running processes.

(2) Termination of each parent process.

(3) Termination of each child process.

(4) Termination of each transaction.

These situations are not self-contained and cause a couple of drawbacks: Firstly, some processes can run for a long time, and while such a process is in execution, the user may wish to be able to undo the effects of some short-running processes[4]. Secondly, it is possible for a process to create offspring and then proceed entirely independently of their existence or demise, and in some cases the parent can itself exit, leaving orphan processes. Finally, as seen in Chapter 7, if each transaction is stored permanently as soon as it occurs, some redundant transactions can arise.

To get around these problems, we need a new policy of storing transactions. The transaction table contains all (possibly interleaving) transactions of currently-running processes. It seems practical that transactions are stored, and later optimised, as with permanent ones in the transaction list when a parent process terminates together with all its subprocesses. If there are orphan processes left in execution after the parent's termination, the storing of transactions needs to be delayed until they all terminate. When a complete termination of the parent occurs, however, it is not adequate to store the transactions of that parent only. The current transactions of other parent processes must be stored which are still in execution at that time, because a transaction of the parent might depend upon that of another parent, both establishing a consistent state together. This kind of storing is repeated until the termination of all current

[4]The user must be aware that this attempt might affect the behaviour of other running processes.

processes, when the associated transactions in the transaction list can be optimised again. Note that the optimisation algorithms will have to be different from ones given in Chapter 7, due to the interleaving of various processes.

`Brush` represents processes' individual activities by means of a reasonable sequence of transactions which can be considered to generate transitions to the file system state. The order in which transactions are performed keeps track of transitional state[5] changes of the system which is ultimately brought from one consistent state to another. Concurrency allows transactions of different processes to cooperate with each other to make transitional changes to the state. In the case of desired control of concurrency, as will be seen in Section 8.4.2, a transaction, which makes a transitional change, always creates a consistent system state. This means that the state to which a process brings the system consists of a total of transitional states that are created by the transactions of that process.

Given recovery at the transaction level, recovery commands deal with the transitional states one by one to reach a previous state, reversing the effects of the transactions that cause them. The applicability of a recovery command requires each transitional state belonging to a process to be consistent. The order of the transactions has an important role in building the consistency. It also helps apply the conflict-detecting algorithms that are used to bring out clashing situations. Therefore, it is extremely important to store concurrent transactions in order in which they interact with the file system, so that undo and redo commands can be authorised safely.

As a result, concurrency involves modifying the form of the history and transaction lists given in Chapter 7. The transaction list needs a new entry for the process (or command) that each transaction belongs to. This entry can be used by the history list to tie the transactions to the related commands, instead of the enumeration in the transaction list. At Case 2 in the previous section, for example, suppose the two (parent) processes are `P1` and `P2`. The concurrent transactions can be recorded in the transaction list as follows:

```
Read fileA 2 P1
Read fileC 6 P2
```

[5]By a transitional state we refer to a state that is created by only one transaction.

```
Create fileD 12 P2
Delete fileB 4 P1
Create fileB 14 P1
```

A request to reverse the effects of P1 is serviced in the way that has been mentioned in the prior chapters; firstly, in order to see whether undo can be applied, each one of the associated transactions is compared to other processes' transactions that appear after it in the transaction list, if any. Then undo is possibly executed and the undone transactions are put in brackets.

8.4 Control Requirements

Brush must protect the consistency of the file system against both concurrent executions and recovery operations. It, therefore, requires more intimate control of file system interactions. This section discusses how simultaneous execution of programs can be controlled effectively and addresses the relationship between recovery and determinism.

8.4.1 Access control

One aim of brush is not to allow programs to execute file operations simultaneously which, when executed, are likely to become irrecoverable. Such operations generally arise when a few programs that need to access the same resource in the system are in execution at the same time, in which case their operations may not be able to be recovered. However, the command-independent implementation does not allow the discovery of the malicious programs of this kind. Now that each program is run by a separate process, a process-based limitation is sufficient to reach our purposes. Using an access control scheme, the brush shell puts some restrictions on the execution of processes. In this fashion, access control can impose process delaying and prevent unauthorised accesses to the components for a limited period of time.

As a result of not considering the interleaving of two or more commands which are to have effects on the same location of the system, the Unix interpreters may damage the integrity of the file system. For instance, the existing interpreters enable users to make alterations to the same system file in different windows simultaneously. Concentrating on the Unix editors such as Vi and Emacs, the user can edit or update the same file in two separate windows at the same time. As far as recovery is concerned, this is an undesirable situation. Although the editors themselves have recovery facilities (Vi supports one-step undo, whereas Emacs does multiple-step undo), once the user leaves them those facilities are not in use any longer in that the editors cannot be called back with the window parameters available at the point when they are closed. A file that is updated differently in two editors will have the contents in the window in which the last write command is issued. The modifications made in the other window will be lost. The best way to prevent this is to control access to files. In conclusion, `brush` must not allow the user to deal with the same file through two windows.

The access control scheme provided for a user working in an operating system environment must be flexible enough to allow the specification of desired access authorisations. The key point of the specification is related to how files can be protected from illegal access. In Unix, every file (or directory) has a set of permissions with it, which determine who can access the file. By changing the Unix file permission information, all users except the owner and super-user can be blocked from accessing any file. That means that accessibility can be prevented by the blockade of the locations associated with current processes of the system. In addition, we need to be able to block all processes from accessing a file, other than the one which currently has it open. This requires monitoring of open files and prevention of access by `brush` itself. With this protection strategy, `brush` needs to create a run-time environment to run each Unix program. This issue will be pursued further in Section 8.5.

8.4.2 Concurrency control

It is essential to prevent concurrent inconsistent changes to the editable data structures of an operating system, such that a recovery mechanism can operate properly. To accomplish this, concurrency control stops two or more events happening at the same time. It is very similar to access control because it may restrain access to data structures. The difference emerges in the way they function. Concurrency prevents authorised but inconsistent changes, while access control prevents unauthorised changes.

The fact that a user can work in several windows simultaneously reveals the importance of concurrency control. There are several issues in the design of concurrency control for recovery mechanisms. The most important is not to restrict concurrent behaviour of the system, in particular while the command interpreter interacts with the operating system. In a system in which multiple windows can be created, this implies that programs running in one window and recovery from them should not influence ones running in other windows. However, in some cases, a recovery mechanism may require suspending the concurrent behaviour in favour of determinism, which may thus destroy the independence among windows. These restrictions apply to concurrent accesses to the components of the mechanism (history list, etc.) as well. In fact, it does not matter how many windows the user is currently working in. A single window may also face similar restrictions for the prevention of the interference of two progams that run in it.

Generally, the control of concurrency does not ensure the applicability of recovery commands to all situations where programs interleave. Even if concurrency is controlled desirably, it may leave some programs irrecoverable, restricting the usability of the recovery mechanism. For example, consider the following situation which is led to by concurrent execution of two programs, `P1` and `P2`, entered in separate command-lines:

```
10> Delete fileA 5 P1
11> Create fileA 11 P1
12> Read FileA 11 P2
13> Create fileB 13 P2
14> Read fileB 13 P1
```

This does not enable both programs to be undone separately. Thus it is impossible to get back the old version of `fileA` (version 5). There is no elegant way to cope with this situation, because the user is currently allowed to specify only one command to undo from the history list at a time. However, the situation can be eliminated via the transaction list, considering the transactions of two programs as if they belong to one command. In the example, typing "undo 11

14" will bring the previous version of `fileA` back to the state. Therefore, the user can wish to be allowed to be able to undo a group of transactions from the transaction list directly.

To be able to control concurrent accesses efficiently, we need to define a data structure, named the *file table*[6]. The file table keeps a transient record of each file system activity of processes that are currently in operation. For our control of concurrency, it is required to contain the following information:

1) process *id*s
2) full pathname of open files

In the table, instead of the full pathname of an open file, a pointer to the inode of the file can also be used. There is no importance of the types of file operations (reading, writing, etc.) for which a process opens a file. Once the file is opened the table keeps a record for it. The record is removed from the table when the file is closed or the process terminates. The file that has a record in the file table cannot be opened by another process. In other words, the process which first opens a file has the priority to read or write the file until it is closed. The time of a file closure can be determined by instructing the `Tracer` to monitor the *close* system call. As with files, accesses to directories must be controlled, a discussion of which can be found in Section 8.5.

Monitoring the *close* system call is also important to the `Recovery Manager`. Since the `Manager` must ensure that recovery operations are not performed on a file which is currently open, it needs to discover the times of the opening and closure of files. The time at which a file is opened can be represented by `Read` or `Create` transactions on that file. In order to allow the `Manager` to find out file closures as well, we must define a new transaction, namely `Close`. This transaction must be inserted into the tansaction list whenever a file-related *close* system call is intercepted.

[6]The file table that is used by the `Tracer` should not be confused with the file table contained in the kernel. The kernel's file table keeps track of the byte offset in the file where the user's next *read* or *write* will start, and the access rights allowed to the opening process and so on.

8.4.3 Determinism and Recovery

In a computer system, any user would like everything to be under control during the interaction with the system. In order to keep all aspects of the interaction under control, the user basically needs to know the system state and the system behaviour after any performed action. Generally, the degree of this control depends on how exact the user's knowledge of commands the system supports is. An interactive system with the undo command enables the knowledge of the system state and behaviour to be increased. Thus, not only does undo allows the user to repair an erroneous state, but it also helps the user in handling non-determinism. In fact, the user's knowledge is concerned with the external system behaviour rather than the internal system structure. When the interpretation of a user-oriented command is differently made by the user and the system, from the user's point of view an error is likely to occur. In such a case the undo command may reduce non-determinism by providing the user with more information on the internal system structure [61].

As a very effective tool, undo is not as easy as it looks. It may force the system to behave differently from what the user expected. In a user interface into which undo facility is added improperly the user may be faced with much risk in interaction. For example, if undo is used for command-line data which embody a sequence of system commands, all the commands will be normally undone. But the user may have expected only the last command of the sequence to be affected by undo operations. The incomplete knowledge about how undo acts puts a burden on the user. Navigating through the system, the user has to make its behaviour foreseeable.

As long as the user knows the system state and behaviour after submission of commands the interaction is deterministic. However, the unexpected modification of the state and the unpredictable behaviour of the system may destroy determinism. This may be caused by accidental execution of a command. Moreover, non-determinism emerges not only from unintentional commands or undesirable states but may also occur due to concurrency. From the user's perspective, concurrent non-determinism arises from the fact that it is impossible to define an objective order of events, which are spatially separated from each other.

In `brush`, while a program is in execution the user can start out a recovery process for another program whose execution has already been completed. Even if priorities between these two programs with respect to the use of system resources are specified, deterministic concurrency may not be accomplished. Broadly speaking, `brush` allows the `Shell` module and the `Recovery Manager` module to deal with programs concurrently and thus allows the effects of their actions on the system to interleave. Allowing a command to be executed before an undo operation terminates may result in clashes, in which case one or all of the undo operation and the command will probably fail. In particular this happens when both tasks are associated with the same component of the system. Although control of the interaction belongs to `Shell` completely, `Shell` can co-operate with `Recovery Manager` in such a way that execution and recovery can be in effect simultaneously. This situation can be regarded as a special case of concurrency. It is the user's responsibility to stop such situations occurring.

On the other hand, concurrency enables many commands of the same kind to be performed simultaneously. By the same kind we refer to recovery and system commands. In terms of determinism, some restrictions must be imposed on the system. Concurrently executing two consecutive recovery commands (concurrent undoing or redoing) is unacceptable and Recovery Manager, therefore, must not be allowed to concurrently undo more than one command because of the possibility of dependence or interference. Unless a process created for an undo operation releases the control of system resources, the other process with a similar task can't be spawned to gain the control. In this way, it can be guaranteed that the interaction is deterministic. But, the concurrent execution of system commands must be allowed under a sophisticated control of `brush` so that the original system behaviour can be preserved.

As a consequence, recovery is associated with the reduction of non-determinism that users themselves cause. If recovery from concurrent situations can be achieved within `brush` by means of the system facilities entirely, satisfying the user's expectations, `brush` will behave as a shell that runs deterministically. Of course, non-determinism the existing system itself gives rise to will remain unchanged to a certain extent.

8.5 Other Concurrency Issues

In this section, we describe a number of complications introduced by concurrency and non-determinism, and propose some solutions. Concurrency arises from shell commands which create and control processes. Most of the problems involve concurrent access to shared data, which is discussed at length in the database management literature. The solution in databases is based on the idea of transactions and the ACID properties of transactions, as described by Date [29]. In our case, we can use the mechanism for monitoring and adapting system calls described in Chapter 6. The system call interface to operating system facilities which it uses is effectively transaction based, and many of the same principles can be applied.

The first issue we discuss is how to deal with single commands which involve several processes. For example, in Unix, a compound command `p1 | p2` creates two processes `p1` and `p2` which communicate via a pipe. It is easy to support undo and redo on the group as a whole, treating it as a single command, but it is difficult to undo or redo individual processes within the group. In the case of `p1 | p2`, the two processes can be dealt with separately if the information sent along the pipe is stored in a file, `tmp` say. Then the compound command can be treated in the same way as two commands `p1 > tmp` and `p2 < tmp` and the parts can be undone individually.

A second type of problem arises when commands share resources unnecessarily with the shell. For example, if a program uses the shell window for standard input and output, this leads to problems of arbitrary and confusing interleaving of text. To avoid these problems in the proposed shell, it makes sense to treat all commands, other than built-in shell commands, as processes which run concurrently. Each program should be run with a separate window being provided for it, where needed, for standard input and output. No special convention (such as the `&` symbol in Unix) is needed. As each program is started up, the shell immediately prompts the user for further commands.

There is then a problem if a program does not work properly and gets stuck in an infinite loop, for example. Conventionally, there is no clean and deterministic way to shut it down. Signals sent to the program (e.g. triggered by Control/C) may be ignored by the program. If a signal is used which cannot be ignored, the filestore may be left in an inconsistent state. However, there is a clean way to kill a rogue program using the undo facility. If the user undoes a previous program which is still running, the shell destroys the process or processes associated with the program, removes any new file versions created by the program, whether complete or partial, and restores everything to a state which is as if the program has never been run. If redo is subsequently used on the program, it must actually be re-executed from scratch, in contrast to our previous description of redo.

Long-running programs which run concurrently with the shell are likely to request resources dynamically. For example, a user may want to insert a file into a document while using a word processor. Problems arise if two programs request the same resource; which one succeeds, and what happens to the other? In conventional systems, this sharing of resources leads to unpredictable behaviour. For example, consider what happens if one program writes to a file `f` and a second program reads from it. The second program may "see" the old or new version of the file (or on some systems, something in between) in a timing-dependent way. Where file locking is provided, this can add to the unpredictability because the second access may succeed or fail, depending on the relative speeds of the two programs.

To solve this, we want to ensure that all dynamic accesses are correctly sequenced by the shell. In the example above, the two relevant shell commands will appear in the shell's history in one order or the other, as determined by the user's sequence of interactions. If the read request appears first, the reading program sees the old version of the file. If the read request appears second, the reading program sees the new version.

One way to achieve this is to treat a dynamic request as a double action. A command is given to the shell to provide the requested resource, and a command is given to the running program to access it. In a graphical setting, such a double action is quite natural. For example, the user may select a file icon using a file manager window belonging to the shell, and use a drag-and-drop operation to give the file to the program. If all dynamic file accesses are mediated by the user and involve an interaction between the user and the shell, then the shell knows what sequence these requests occur in, and any conflicts between programs accessing the same resources can be resolved. It may seem restrictive that all resource requests should involve the user, but we have already seen that when running foreign programs, it is desirable to ensure that every dynamic access is interactively sanctioned. It is not unreasonable to apply this to all programs, not just foreign ones, though some mechanism for allowing a program to have silent access to files which “belong” to it may be desirable.

Although a drag-and-drop interface is probably the most natural way of presenting this to the user, a textual version is also described here. For a request to read from a file `f` say, the shell can provide a command `give f n` where `n` is the time at which a previous program was started up, or some other identifier for the program. This makes file `f` visible to the program. An attempt by the program to read the file then succeeds. For a request to write a file `f`, a similar `take f n` can be provided. The original command to start up the program, and all the subsequent associated `give` and `take` commands have to be treated as a single group which are all undone or redone together.

We have been assuming up to now that an operating system transaction such as writing a file is indivisible. In practice, a program makes several system calls to write a file; one to open the file, several to write data into it, and one to close it. This raises the question of what happens if another program requests access to the same file while it is open. The answer is that the second program must be suspended and made to wait until the requested file becomes available, so `brush` effectively implements a locking mechanism. This raises the question of how we know whether it is the same file that is being accessed. As mentioned in Section 4.5, we need to find unique identifiers for files to get around problems with aliases.

For some long-running programs, particularly search programs, it is appropriate to stop them before they terminate when the results produced so far are judged sufficient. In this case, the user would want to save the partial results. This can be accomplished by a commit operation, also implemented as a double cooperative action between program and shell, which effectively splits the program run into two separate program runs.

A further question that arises is what happens if there are multiple users, or if a single user runs several shells. In order to avoid losing the deterministic properties of transaction sequencing, we propose the use of directory locking.

When `brush` accesses a directory for updating, the directory is locked, which prevents two users from updating the same file at the same time. Also, when a shell attaches itself to a directory to start reading files from it, we want it to see a consistent view of the directory for the duration of the attachment. This can be achieved reasonably easily, because the old versions of files which are kept allow a shell to "see" a directory at any particular time in the past, instead of the current state. A shell can thus keep a consistent view of a directory, corresponding to the time at which it attached. In the case of a directory which is being updated, a shell which attaches for reading sees the directory as it was before the updates started. In this way, a series of updates within a directory is treated as a single transaction from the point of view of other shells, so that the set of files as a whole is always seen in a consistent state. A commit command, equivalent to detaching and reattaching, can be added to allow finer grained control over events.

Chapter 9

Conclusions

The ability to undo and redo commands is a desirable facility of interactive systems which increases confidence and productivity. The work described in this thesis, on the whole, is concerned with selective undo facilities. Below it is summarised, and some conclusions and future directions are presented.

9.1 Summary

This thesis describes the design and implementation of user-oriented recovery facilities for operating system environments. It has used the Unix operating system to show how to achieve user-oriented recovery, prototyping recovery facilities in a shell environment. The argument of the thesis can be summarised as follows.

9.1.1 Operating system facilities

The main requirement of a recovery mechanism is to be able to notice the destruction that a user action is to cause on the state of the system beforehand. Modern operating systems provide various methods of controlling user actions. These methods allow capturing all operating system tasks requested by a user before they modify the state. In Unix systems, for example, there are two methods, the *ptrace* system call and the */proc* file system, which trace processes and obtain required information. The */proc* file system has more advantages compared to the *ptrace* system call, such as allowing the tracing of arbitrary processes and fine-grained control of their memories.

Users employ a particular program to communicate with operating systems. This program is often called a *shell*, which manages the interaction between users and the operating system. It prompts users for input, interprets that input for the operating system, and then handles any resulting output from the operating system. The shell is the most convenient user interface which can be equipped with recovery facilities at the user level, because it eliminates the need to modify the internal structure of the operating system. An additional advantage is that it makes the provision of the recovery mechanism for a single user possible and easy, without the help of the system administrator.

9.1.2 User-oriented recovery support

Interactive undo support has an important influence on how usable a system is. Although it does not change the functionality of the system, it enhances usability and learnability of the system. User-oriented undo support deals only with situations such as user abolishment of prior commands and the use of such a facility depends on the user's detection of the undesirable situation. There are some examples of situations where user-oriented undo can be used; repairing mode errors in editors, handling interactive deadlocks, helping explore new or unfamiliar functions of systems. However, recovery situations such as system crashes need to be handled by the system.

The functional characteristics of undo support are described by four properties; reversibility, inversibility, self-applicability, and unstacking. The property of reversibility states that the state of a system can be reversed to a previously existing state after a sequence of task-oriented commands have operated on the system. The property of inversibility states that the effects of undo support commands can be reversed. The other two properties classify a particular undo command. A self-applicable undo command can reverse the effects of only the last command and unstacking undo commands can reverse the effects of more than the last command. There is another property of undo support, thoroughness, which is a basic performance requirement.

The models proposed for undo support fall into two groups, primitive undo model and meta undo model. The primitive undo model has the properties of self-applicability and reversibility, and provides two kinds of undo; single undo and multiple undo. A single undo command reverses the effects of the last user-issued command which can be either task-oriented or undo. A multiple undo command can reverse a specified number of previous commands. The meta undo model has the properties of unstacking and multiple reversibility, and also provides two kinds of undo; linear undo and selective undo. A linear undo command allows undoing and redoing of a sequence of task-oriented commands. A selective undo command allows undoing of an arbitrary command.

9.1.3 Adding undo in operating systems

The general context of this research was dedicated to investigating how to provide operating systems with user-oriented recovery facilities. Recovery mechanisms need to find out commands' effects on the system. An operating system can have a large number of user commands, which can be classified into three categories in terms of their effects on the state; ineffectual commands, reversible commands, and irreversible commands. An ineffectual command has no effect on the system state. A reversible command has an effect on the state which can be reversed. An irreversible command has an effect on the state which may not be reversed. The domain of the recovery mechanism only covers reversible commands.

In operating systems, sufficient information about which part of the system data each command is to modify can be obtained in several ways. For example, part of the filing system or the file manager that is an interface to it can be re-implemented and replaced. However, some operating systems like Unix ensure that commands interact with the file system via system calls. Concentrating on the Unix system and its system call interface, a recovery facility was proposed for shells, whether textual or graphical, providing a mechanism for recovery from accidental loss of files through unintentional user commands. The facility is convenient, and has a clear meaning for the user in terms of a visible command history. The ability to select arbitrary commands to undo or redo, and to insert commands at arbitrary points in the history, makes it possible to recover any previous state or any collection of desired files.

9.1.4 Design of a recovery facility

The approach to recovery from user commands concentrates on the provision of a user with full satisfaction of existing facilities as well as undo support facility, with the production of code which maintain a system-like working environment. Hence, in shell-based operating systems, a recovery model involves a rational saving mechanism for retaining required checkpoints, a command interpreter for supporting system commands and a recovery mechanism for executing recovery commands. In our prototype, the saving and recovery mechanisms construct the `Recovery Manager` module. The command interpreter is implemented in the `Shell` module. Besides, the prototype employs another module, `Tracer`, which controls file system accesses of system calls and captures recovery information.

Each system call that is intercepted by `Tracer` are converted into a number of transactions which are used for recovery operations. Transactions such as `Read`, `Create`, and `Delete` are defined for files. `Read` represents the reading of an existing file, `Create` represents the creation of a file from scratch, and `Delete` represents the removal of an existing file from the file system. For example, the effects of the *create* system call on an existing file can be denoted by one `Delete` and one `Create` on the old and new contents of the file, respectively. Additional transactions, such as `Make` and `Remove`, are defined for the system calls that operate on directories.

9.1.5 Prototype of the shell with undo

A prototype called `brush` has been constructed to demonstrate the algorithms described. It is a textual shell implemented under Unix, and it uses the Unix `/proc` mechanism to intercept the system calls made by commands and programs. Alexandrov et. al. [2] use the same mechanism to implement direct access to remote file stores by intercepting file system calls and altering their arguments. In our case it is not necessary to alter the arguments to a system call before it goes ahead, merely to insert some extra processing. Also, where file handling is concerned, we need only intercept `open` and `close` calls, and not `read` or `write` calls. In practice, with typical use of the shell, the time overhead involved in managing file versions is acceptable.

The prototype maintains two separate lists of commands and their transactions, called history list and transaction list. Using system calls that are issued by each command, the corresponding transactions are recorded and associated with the relevant command. The kind of recovery technique used in storing recovery information depends upon the characteristics of the file system calls. Considering the three types of system calls, self-invertible, invertible, and destructive, the most appropriate technique was employed. For purposes of efficiency and redundancy, the transactions and recovery information that belong to a command line are optimised before they are stored permanently.

Recovery in `brush` is carried out by two recovery commands, selective undo and selective redo, which can be treated as ordinary shell built-in commands except that they are not added into the history list. These commands allow the user to reverse the effects of an executed or undone command arbitrarily. The use of selective undo and redo is restricted in some conflicting situations, in order to protect the file system consistency. The conflicting situations for both recovery commands are detected through the transaction list. If a file was created or altered by the command, and the new version of the file is mentioned in the transactions of any subsequent active command in the history list, then the undo cannot be used because it would lead to a version inconsistency. The same is true for the redo command. However, selective redo is more restrictive than selective undo and needs an analysis of the transactions of the previous commands as well.

9.1.6 Multi-user undo and concurrency

Undo facility is not only a feature desired within single-user environments, but it is also desirable within some multi-user environments. Users can expect an interactive system with multi-user undo to allow them to undo both their own commands and other users' commands. This makes control of multi-user undo more complicated than single-user undo. Because of concurrency, since their commands can interleave or interfere with each other easily, it can be difficult to achieve control of interactions and the detection of dependencies between commands. Some methods, such as locking and roles, can be used to make commands avoid interfering. In the current literature, there are several undo models defined for multiple users.

In a concurrent system, processes can be created statically or dynamically, which perform concurrent transactions. In either case, concurrent access to the file system can occur and such a situation can cause non-determinism. On the other hand, the kind of process creation affects recovery situations. Static processes are generally separate ones which require an individual recovery procedure to be employed. Dynamic processes (e.g., in Unix systems, process creation by *fork* is dynamic) construct a group of processes and thus a recovery operation must be applied to them in their entirety. Even if programs are executed from the command line sequentially, similar problems may arise. We proposed some effective solutions to reduce these kinds of non-determinism to a minimum, ensuring the ordering of concurrent transactions and preventing process interference. The current prototype of `brush` allows for concurrency in a limited way, using the techniques described in Chapter 8.

9.1.7 Program maintenance

In this thesis, we have also presented the design and implementation of `bmake` which carries out compilations of source files to build executable programs. It has almost the same behaviour as the Unix *make* program, except that it does not need an auxiliary file (often named *Makefile*). The approach adopted in `bmake` is based on command tracing that is provided by `brush`. All compilations are made under `bmake`'s control to find out the file system transactions, and thus to discover the dependencies of each object file. Not having to specify dependencies for a particular compilation enables programmers to focus entirely on the development of programs.

The way that `bmake` works is compiler-independent; it does not need to know which programming language a programmer uses. Once the programmer compiles a particular source file by specifying the entire structure of the compiling command, `bmake` issues that command for the subsequent re-compilations of the source file. The dependencies of the target are taken from the transactions which have been recorded at the time of the last compilation of the source file, no matter whether it is made by the programmer or `bmake`. Checking the recent transactions, if any, for modifications made to each dependency, the requirement of reproducing the target is determined.

9.2 Evaluations and Conclusions

In this study, we aim for the provision of a user-level recovery facility for operating systems. Much effort has been carried out to make this facility easier to design and use. This section presents some important evaluations and conclusions which can be drawn from the analysis of recovery issues pointed out by this thesis.

9.2.1 `Brush` evaluation

`Brush` keeps all command executions under its control, taking advantage of tracing facilities provided by the **/proc** file system. As with conventional shells, it gives a prompt to the user and waits for a command to be typed. When the command is issued, its execution is handled. The difference is that, while the command is being executed, `brush` attaches itself to the process that is created to run it, and monitors the process to trace a particular set of system calls, particularly file system calls. In this way, selected system calls are stopped on entry to the kernel and necessary items of recovery information (e.g. file versions, command transactions, etc.) are stored. Then the stopped system call is resumed. The user needs no knowledge of what's going on in `brush`. Therefore, `brush` is as easy to use as a conventional shell.

As an example, suppose the transaction list for two files, `fileA` and `fileB`, created from scratch (e.g. via an *emacs* editor) is as follows:

```
1> Create fileA 1
2> Create fileB 2
```

where each number that is given to the filenames represents the current contents of `fileA` and `fileB`. At this stage of the interaction, if the following standard Unix command is executed

```
cp fileA fileB
```

Then `brush`, monitoring the execution, intercepts a *read* system call on `fileA` and a *creat* system call on `fileB`, and thus stores new transactions in the transaction list:

```
3> Read fileA 1
4> Delete fileB 2
5> Create fileB 5
```

Also, it stores the old version of `fileB` (version 2) in the `.save` directory, since a `Delete` transaction has operated on it. These are a maximum of overheads that the *cp* command imposes.

In order to explore the practicality of using `brush` further, suppose the user has issued a sequence of 10 Unix commands given in Figure 9.1. The figure also indicates the transaction list which corresponds to these commmands.

History List	Transaction List
1> emacs fileC	1> Create fileC 1
2> cp fileC fileD	2> Read fileC 1
	3> Create fileD 3
4> mkdir dirA	4> Make dirA 4
5> mv fileD dirA	5> Delete fileD 3
	5> Create dirA/fileD 5
6> cd dirA	6> Path dirA 4
7> emacs fileD	7> Read fileD 5
	8> Delete fileD 5
	9> Create fileD 9
10> cat ../fileC fileD > fileE	10> Read ../fileC 1
	11> Read fileD 9
	12> Create fileE 12
13> ls	
13> rm fileD	13> Delete fileD 9
14> mv fileE fileD	14> Delete fileE 12
	15> Create fileD 15

Figure 9.1: An example interaction: Commands and Transactions.

As seen in Figure 9.1, the effects of 10 commands are represented by 15 transactions. For each command, the transactions created vary between one and three, except for *ls* which does not creat any transaction. The `.save` directory contains four files called `3`, `5`, `9`, and `12`, representing three versions of `fileD` and one version of `fileE`, as a result of the effects of `Delete` transactions.

Examining the files in the `.save` directory, we see that the files `3` and `5` have the same contents. It is not possible to find out this by only using the transaction list, because `brush` does not keep a record of whether a `Delete` transaction is derived from the system calls *rename*, *creat*, or *unlink*, which are made by the commands *mv*, *emacs*, and *rm*, respectively. Therefore, `brush` can store file versions redundantly. The redundant storing is generally caused by the *rename* system call.

For bigger commands (or application programs) such as *xfig* and *pine* through which the user can interact with the file system dynamically, a larger number of transactions needs to be recorded. If any `Delete` in these transactions is encountered, the corresponding file versions are also stored in the `.save` directory.

Similarly, recovery commands may also give rise to overheads. For example, if the user undoes the last command (*mv*) shown in Figure 9.1, the current version of `fileD` (version 12) is stored in `.save`. This is because `undo` performs a `Delete` transaction in order to reverse the effect of the `Create` transaction at time 15. In this case, the `.save` directory contains four versions of `fileD`. To complete recovery operations, `undo` also moves the file `12` in `.save` to `dirA` by renaming it as `fileE`, which may get overheads down.

In `brush`, on the other hand, commands such as compilers are handled differently. Old versions of files (object files, executables, etc.) that these commands generate are not stored, because those versions can be re-generated (see Section 9.2.6). This reduces their potential overheads to a minimum.

Transactions are substantially recorded for pre-existing files and directories when `brush` is initialised. As seen in Chapter 7, these transactions hold information of version numbers which represent the current contents of pre-existing files and directories and are used by recovery commands. The number of them is normally equal to the number of files and directories that exist during the initialisation. Besides, if `brush` is started from a directory other than "HOME" and there are some symbolic links available, further initial transactions are recorded. However, `brush`'s initialisation does not require any file version to be stored in the `.save` directory.

Another problem with `brush` is that some state-ineffectual commands can create conflicts. Representative examples of such Unix commands are *cat*, *grep*, and *sort*, which perform `read` operations on the file system and thus only have `Read` transactions. To illustrate, consider the following commands:

```
edit fileA
cat fileA
```

With the current version of `brush`, the `edit` command cannot be undone because `fileA` would no longer be available at the time of execution of `cat` command. Given that `cat` is state-ineffectual, the user could expect `edit` to be undone independently. This is caused by a command-independent design policy adopted in `brush`: every command that executes at least one transaction is assumed to be state-effectual. However, the problem can be avoided by making `brush` ignore the commands that have only `Read` transactions whenever conflicts are checked, which is the scope of future work.

9.2.2 System and user features

`Brush` provides features which are clear and convenient for the user, and which integrate well with existing operating systems.

Many interactive systems have based their recovery capabilities upon the notion of sequential undo. The undo models provided in these systems require that only task-oriented commands are executed under control. Recovery commands do not usually have to be kept under control. In contrast, selective undo has to control the execution of both task-oriented and recovery commands.

Users expect an interactive system to provide recovery facilities in terms of multiple reversibility and inversibility. An erroneous command may not be noticed as soon as it is executed. Besides, the commands issued until the user notices the erroneous command may be too many in number and of great use in importance. The user would not probably want to deal with them before reversing the effects of the erroneous command. Selective undo facility saves the user from such inconvenience.

The recovery mechanism for operating systems should provide a command history recording executed and undone commands. This is necessary to be able to establish the interaction between the user and the mechanism. Whether irreversible commands should appear in the command history may be determined by the user. Since irreversible commands are not in the domain of the mechanism, it may be desirable to hide them from users. Likewise, ineffectual commands, such as ones that display state information can be hidden from appearing in the history. However, both irreversible and ineffectual commands are in the domain of the `resubmit` function and are displayed in the history to allow the user to use them again. Mistyped commands also appear in the history to allow the user to re-edit them. On the other hand, for the system, a transaction history recording the effects of commands on the state is provided.

It is quite important to employ the most efficient recovery technique for storing information required for each user command. Among the five recovery techniques summarized in Chapter 6, the complete rerun and full checkpoint techniques are generally not suitable for building the recovery facility in operating systems. This is because the file system that belongs to a particular user may contain as much data as in a database system. The other techniques may be used for operating systems, whose functionality varies according to the kinds of commands.

9.2.3 Consistency features

`Brush` provides a greater level of consistency than many previous systems.

In the literature, we did not find a clear discussion of the consistency properties of undo/redo, even for sequential undo system. Suppose that the user submits three task-oriented commands, two undo commands, and a new task-oriented command, respectively, resulting in the history list being $[c_1, c_2^{-1}, c_3^{-1}, c_4]$. When c_4 is undone, the next command to undo appears to be c_3^{-1}, which means the redoing of c_3. If c_3 depends upon c_2, then the single undo mechanism must not allow undoing c_3^{-1}, discovering the dependency. Past applications have avoided using such a model of sequential undo, probably due to this kind of problem. The example shows that sequential undo may be as complex as selective undo and require dependency checking. In our model, before allowing an arbitrary command in the history list to be undone, `brush` checks for safe execution of the undo.

Besides, if two commands are working simultaneously on the same component, the shell ensures that their modifications to the component do not overlap, e.g. through the delaying of one of the commands, which is essential to make the system call deterministic.

In general, although it is essential that recovery mechanisms do not depend on the correctness of programs that they deal with, the `proc` file system used by our mechanism does not make this possible. There are some problems with executable files produced by tools such as compilers. For example, a user must ensure that an executable file is opened and closed properly and ended with proper exit code. This is required so that `brush` can control concurrency, using the *close* system call. Note that, in Unix systems, when a process is to be terminated, all the files that have been remained open are closed in the kernel level and thus `brush` cannot catch these closures. Already-existing commands and programs obey these rules and so there is no problem with handling them. In the design of `brush`, we assumed that each program is correct in the sense that if it were executed in our shell, then its result is the desired one. It is the responsibility of the programmers to ensure that their programs are indeed correct.

9.2.4 Extension with undo

Undo constitutes a way of extending interactive systems. `Brush` ensures that undo remains only as an extension. It does not degrade the usability of the system and confuse users who have already been acquainted with it. That is, when a system is extended by adding undo, the original system behaviour is preserved within the new extended system.

In a system that will be extended with undo, it is not practical that all user commands are in the domain of recovery. A general recovery facility for an arbitrary interactive system is feasible if it is restricted to a supporting file system [4]. Such an interactive system would be required to store all of its modified state in the file system. Although this can be perceived as a drawback, the implementation of the recovery facility in this way is simpler and more reliable than incorporating it directly in the interactive system. This fact has motivated `brush` to handle only the file system calls.

It seems more difficult to realize a recovery mechanism after a system is built. However, the user-level facilities provided by the system can ease the realization of the mechanism. Even it can be possible to provide a recovery mechanism that functions independently of user commands. In such a case the mechanism does not need to be extended for new commands added to the system; they can be handled by the mechanism automatically. This thesis has proved that a command-independent recovery mechanism is achievable at the user level. In conclusion, for an operating system to be extended with undo, it is adequate to concentrate on modifying user interfaces (shells in the case of Unix-like operating systems), without affecting the internal structure of the system.

9.2.5 Recovery evaluation

The notion of undo in operating systems has many differencies from the notion of undo in editors. This makes `brush`'s task harder. The success of our approach can, however, be evaluated by comparing the provision of recovery in these two environments.

- In operating systems, user-oriented recovery facilities have to be a part of the kernel neither at the beginning nor afterwards. The user-level implementation is always achievable, using modern techniques that allow control of the working environment. In editors, it is quite difficult to gain smooth control of the editing environment. This makes it impossible to embed a recovery facility in editors transparently. Therefore, editors must be designed with recovery facilities at the outset.

- A recovery mechanism must be able to handle all commands that interact with the file system. Given that commands may vary from one operating system to another and new commands may be included to the system, the mechanism must be command-independent. This is also necessary because the user can write executables in any programming language that affect the file system. Editors are distributed with a fixed set of commands, which enables the mechanism to be able to focus on commands directly.

- In both environments, partial checkpoint is the most favourable recovery technique to save recovery information. Operating systems contain three kinds of components associated with the filing system; files, directories, and access permissions, which all have to be handled by the recovery mechanism. A file usually holds a whole unit of data and needs to be stored in its entirety against all the destructive file operations. For the other two components, it is adequate to store a small amount of information. Editor commands operate on various segments of a single file and only the segment that a command is supposed to act on is generally stored, possibly with information of its insertion point which will be used during recovery.

- Recovering an operating system command involves determining both which version of a relevant file to restore and which directory to put it in. The directory in which a file restoration is to be made can only be discovered through directory renaming commands held in the command history. Sometimes it may be difficult to trace directory changes and this may result in the failure of recovery. In editors, recovery is equivalent to the insertion of the relevant file segment back to the current state.

- Operating system commands operate on the components as a whole or can be assumed to do so. Thus it is easy to define atomic transactions which correspond to the effects of each command. On the contrary, editor commands modify the same single file and cannot be easily converted into transactions, which does not allow a command-independent implementation of undo in editors.

- Concurrency is a potential threat to the achievement of recovery in operating systems. Being the cause of command interference, it complicates the process of storing necessary information for each command. Similar problems also arise with carrying out recovery operations. The recovery mechanism must have interfering command transactions performed serially. Many editor environments for single users, whether text or graphic, are free from concurrency's complications.

- As far as multiple users are concerned, some restrictions must be applied in operating systems. An operating system component can be allowed to be accessed by only one user at a time. Editor environments are more complex to control than operating system environments. A user needs to work on a document simultaneously together with other users. The recovery mechanism must put all users' actions in a sequence so that it can check for dependencies between them correctly, which may not always be possible because of reasons such as slow communication lines.

9.2.6 `Bmake` conclusion

Our approach of designing `bmake` has made it necessary to be able to differentiate a compiler from an editor. `Bmake` needs this to relate a file to the right dependencies. For example, suppose that a programmer, currently working on a source file, calls up another file in the editor. In this case, `bmake` must not treat the second file as a dependency, otherwise changing that second file would cause `bmake` to re-run the editor. This can be dealt with either by declaring certain programs to be editors, and thus not to cause dependencies, or by declaring certain types of files to be source files, and thus never to need re-creating.

This information means that the `bmake` program can work cooperatively with `brush` to help avoid saving old versions of files. The use of `bmake` allows the system to determine which files are object files. Thus old versions of them need not be kept by `brush`, because they can be reconstructed whenever necessary. This works even in the presence of undo, because old versions of object files can be reconstructed from old versions of source files.

For situations where the usual *make* utility is more useful, it is possible for `bmake` to construct a description file. For instance, if an application program which has been developed under `bmake` is distributed, users who receive it may wish to use `make` for re-producing the program. This is often desirable when they have no history of commands which `bmake` requires. In order to allow using *make*, `bmake` can generate the relevant description file, which can then be included in the distribution.

9.2.7 Research achievements

In this thesis, despite some shortcomings of `brush` given in Section 9.2.1, we have reported the following achievements:

- It is possible to import a `brush`-like system into any operating system with suitable facilities (for minimum requirements, see Chapter 5),
- Process tracing facilities are the most convenient techniques to build a recovery mechanism in Unix systems,
- The recovery mechanism eliminates the need to adapt the kernel, and deals with a dynamically changing set of user commands,
- Monitoring the relevant system calls made by commands, necessary recovery information is stored and the system calls are recorded as a series of transactions,
- The dependencies between commands are determined, so it is easy to find out which commands get involved in which versions of which files,
- The conflicts arising in the use of undo and redo are discovered and some support in getting around them is provided,
- It is proved that `brush` can present some further facilities, such as execution of programs under control and protection against malicious programs,
- A `bmake` utility is implemented which offers automatic dependency-checking and re-compiling facilities,
- The ideas to design a sophisticated operating system shell are developed.

9.3 Future Research

Our prototype of `brush` is an important step in implementing the selective undoing and redoing of Unix operating system commands. There are many aspects of the prototype which must be developed further. In this section, we outline some of them.

The prototype currently keeps all versions of all files, without compression. This has an obvious space overhead, and the efficient storage of multiple versions of files needs to be addressed. Also, methods are needed for determining exactly which versions need to be kept and for how long, for example by detecting which files are source files and which files are generated files so that only source files need be kept. In the long run, a shell which has much more high level information may be able to cooperate with the user in managing a user's file space better than at present.

In `brush`, in order to determine if the file system calls run successfully, we did not need to stop them on exit from the kernel and to look at their status (i.e. the returned values). Instead, we have assumed that it is adequate to examine the status of the relevant files and directories. For a system call that is intercepted, `brush` checks the existence and access permissions of the files or directories that the call is going to operate on, and stores the corresponding recovery information. The assumption is really useful under normal circumstances, because access permissions have significant effects on the abortion of system calls. On the other hand, the Unix system allows a user process to have only a restricted number of files open, which may affect the status of the *open* and *creat* system calls. Therefore, whenever a process issues either *open* or *creat*, it would also be necessary to compare the number of open files that belong to that process to the maximum number allowed by the system.

Concurrency is considered in a limited way in the prototype. Chapter 8 has addressed many concurrency-related issues and proposed various solutions to the identified problems to ensure determinism. It has also examined to what extent user-oriented recovery can be provided for multiple shells and users. Immediate work will concentrate on these issues.

The only system calls which are intercepted are file system calls, and process control calls in order to monitor all the processes created by a program. No attempt is currently made to monitor external effects or communications or signals. Thus further work needs to be done on monitoring and handling programs with external effects, including programs which cooperate with remote filestores or other services, and on security issues.

Other work is being carried out on adding further facilities based on the ability to monitor commands centrally. These facilities may be both sophisticated versions of already-existing programs and new ones, which `brush` enables to be developed. For example, `bmake` has been implemented in this thesis as an automatic version of the Unix *make* program.

The `bmake` facility requires some additional work to be done. As far as a compiler independent implementation is concerned, there is a problem in allowing `bmake` to be called without any arguments. It can then be difficult to find the name of the final executable file (or files) to start the checking process. All these problems are the scope of future work.

Bibliography

[1] G.D. Abowd and A.J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

[2] A.D. Alexandrov, M. Ibel, K.E. Schauser, and C.J. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[3] J.E. Archer Jr. and R. Conway. COPE: A cooperative programming environment. Technical Report TR81-459, Department of Computer Science, Cornell University, Ithaca, N.Y., June 1981.

[4] J.E. Archer Jr., R. Conway, and F.B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6:1–19, 1984.

[5] E.H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, Spring 1998.

[6] M.J. Bach. *The design of the Unix operating system.* Prentice-Hall International, Inc., 1986.

[7] R.M. Baecker, D. Nastos, I.R. Posner, and K.L. Mawby. The user-centered iterative design on collaborative software. In *INTERCHI'93 Conference Proceedings*, pages 399–405, Addison-Wesley, 1993.

[8] T. Berlage. Recovery in graphical user interfaces using command objects. Arbeitspapiere der GMD, Sankt Augustin, Germany, 1993.

[9] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Human Computer Interaction*, (1):269–294, 1994.

[10] T. Berlage and A. Genau. A framework for shared applications with a replicated architecture. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Atlanta, GA, Nov 3-5, 1993), pages 249–257, 1993.

[11] T. Berlage and A. Genau. From undo to multi-user applications. In *Proceedings of the Vienna Conference on Human–Computer Interaction* (Vienna, Austria, Sept 20-22, 1993), pages 213–224, 1993.

[12] T. Berlage and M. Spenke. The GINA interaction recorder. In *Proceedings of the IFIP WG2.7 Working Conference on Engineering for Human–Computer Interaction*, (Ellivuori, Finland, Aug 10-14, 1992).

[13] B. Betts, D. Burlingame, G. Fischer, and et al. Goals and objects for user interface software. *Computer Graphics*, 21(2):73–78, 1987.

[14] B. Blackmore. Unix shell differences and how to change your shell. http://www.looking-glass.org/.

[15] M. Blasgen. The recovery manager of the System R database manager. *Computing Surveys*, 13(2):223–242, 1981.

[16] S.R. Bourne. *The Unix system.* Addison–Wesley Publishing Company, 1983.

[17] J.S. Briggs. Generating reversible programs. *Software - practice and experience*, 17(7):439–453, July 1987.

[18] C.M. Brown. *Human-computer interface design guidelines.* Intellect Ltd., 1999.

[19] R. Bubenik and W. Zwaenepoel. Optimistic make. *IEEE Transactions on Computers*, 41(2):207–217, 1992.

[20] E.C. Bueche, M.T. Franklin, E.R. Holley, H.F. Korth, and G.C. Sheppard. Oonix: an object-oriented unix shell. In *Proceedings of the Twenty-Second*

Annual Hawaii International Conference on Software; System Sciences Track, Vol.II, pages 928–935, 1989.

[21] A. Carasik. *Unix secure shell.* McGraw-Hill, 1999.

[22] S.K. Card, T.P. Moran, and A. Newell. *The psychology of human-computer interaction.* L. Erlbaum Associates, Hillsdate, NJ, 1983.

[23] J.M. Carroll and S. Mazur. Lisa learning. *Computer*, 19(11):35–49, 1986.

[24] R. Choudhary and P. Dewan. Multi-user undo/redo. Technical Report TR125P, Computer Science Department, Purdue University, 1992.

[25] R. Choudhary and P. Dewan. A general multi-user undo/redo model. In *Proceedings of European Conference on Computer Supported Work*, pages 231–246, 1995.

[26] A. Cooper. *About face: the essentials of user interface design.* IDG Books Worldwide, Inc., Foster City, CA., 1995.

[27] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers* C-13, 6:531–540, 1982.

[28] J. Darlington, W. Dzida, and S. Herda. The role of excursions in interactive systems. *International Journal of Man-Machine Studies*, 18(2):101–112, 1983.

[29] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, 1995.

[30] P. Dewan, R. Choudhary, and H. Shen. An editing-based characterization of the design space of collaborative applications. *Journal of Organizational Computing*, 4(3):219–240, 1994.

[31] T. Duff. Rc–A Shell for Plan 9 and Unix systems. In *UKUUG Conference Proceedings*, pages 21–33, Summer 1990.

[32] C.A. Ellis, S.J. Gibbs, and G.L. Reln. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[33] D. Endler. Intrusion detection: Applying machine learning to Solaris audit data. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 268–279, 1998.

[34] R. Faulkner and R. Gomes. The process file system and process model in UNIX system V. In *Proceedings of the 1991 Winter USENIX Conference.* USENIX Assoc., Berkeley, CA., 1991.

[35] S. Feldman. MAKE–A computer program for maintaining computer programs. *Software-Practice and Experience*, 9(4):255–265, Apr. 1979.

[36] J.D. Foley and V.L. Wallace. The art of natural graphic man-machine conversation. *Proceedings of the IEEE*, 62(4):462–471, 1974.

[37] G.S. Fowler. The fourth generation MAKE. In *Proceedings of the 1985 Summer USENIX Conference*, pages 159–174, June 1985.

[38] B. Fox and C. Ramey. GNU Readline Library. Free Software Foundation, Inc., July 1994.

[39] A. Goldberg. *Smalltalk80: The interactive programming environment.* Addison–Wesley Publishing Company, 1984.

[40] I. Goldberg, D. Wagner, R. Thomas, , and E.A. Brewer. A secure environment for untrusted helper applications–Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium.* USENIX Assoc., Berkeley, CA., 1996.

[41] R.F. Gordon, G.B. Leeman, and C.H. Lewis. Concepts and implications of undo for interactive recovery. In *Proceedings of the 1985 ACM Annual Conference*, pages 150–157, 1985.

[42] M. Gray. Wish–A window-based shell for X. Technical Report CSD-88-420, Department of Computer Science, University of California, Berkeley, May 1988.

[43] P. Haahr and B. Rakitzis. Es: A shell with higher-order functions. In *Proceedings of the 1993 Winter USENIX Conference.* San Diego, CA., pages 53–62, January 25-29, 1993.

[44] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4):287–317, 1983.

[45] S.J. Hanson, R.E. Kraut, and J.M. Farber. Interface design and multivariate analysis of Unix command use. *ACM Transactions on Office Information Systems*, 2(1):42–57, March 1984.

[46] G.G. Helmer, J.S.K. Wong, V. Honavar, and L. Miller. Intelligent agents for intrusion detection. In *IEEE Information Technology Conference*, pages 121–124, 1998.

[47] I. Holyer and H. Pehlivan. A recovery mechanism for shells. *The Computer Journal*, 43(3):1–9, 2000.

[48] I. Holyer and H. Pehlivan. An automatic *make* facility. Technical Report CSTR-00-001, Department of Computer Science, University of Bristol, Jan. 2000.

[49] A. Hume. Mk: A successor to make. Technical Report 141, AT&T Bell Laboratories, Murray Hill, NJ, Nov. 1987.

[50] M.B. Jones. Interposition agents: Transparently interposing user code at the system interface. *ACM SIGOPS Oper. Syst. Rev.*, 27(5):80–93, Dec. 1993.

[51] B.W. Kernighan and R. Pike. *The Unix Programming Environment.* Prentice-Hall, Inc., 1984.

[52] M. Knister and Prakash. A. DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 343–355, Los Angeles, California, October 1990.

[53] W.H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *Computing Surveys*, 13(2):149–183, June 1981.

[54] D.G. Korn. KSH–a shell programming language. In *Proceedings of the 1983 Summer USENIX Conference,* Toronto, pages 191–202, 1983.

[55] A.P. Kosoresow and S.A. Hofmeyer. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, 1997.

[56] P.A. Lee. Exception handling in C programs. *Software-Practice and Experience*, 13(5):389–405, 1983.

[57] G.B. Leeman. A formal approach to undo operations in programming language. *ACM Transactions on Programming Languages and Systems*, 8(1):50–87, January 1986.

[58] P. Lemmons. A guided tour of Visi On. *Byte*, 8(6):256–278, 1983.

[59] M. Loukides and A. Oram. *Programming with GNU software.* O'Reilly and Associates Inc., 1996.

[60] R.L. Mack, C.H. Lewis, and J.M. Carroll. Learning to use word processors: problems and prospects. *ACM Transactions on Office Information Systems*, 1(3):254–271, 1983.

[61] R. Mancini, A.J. Dix, and S. Levialdi. Dealing with Undo. In *Proceedings of Interact'97,* Sydney, Australia, Chapman and Hall, 1997.

[62] C.S. McDonald. fsh-A functional Unix command interpreter. *Software-Practice and Experience*, 17(10):685–700, October 1987.

[63] N. Meyrowitz and A. Van Dam. Interactive editing systems: part 1. *Computing Surveys*, 14(3):321–352, 1982.

[64] A. Monk. Mode errors: a user-centred analysis and some preventative measures using keying-contingent sound. *International Journal of Man-Machine Studies*, 24(4):313–327, 1986.

[65] C. Morgan, G. Williams, and P. Lemmons. An interview with Wayne Rosing, Bruce Daniels, and Larry Tesler. *Byte*, 8(2):90–114, 1983.

[66] B.A. Myers, R.G. McDaniel, and et. al. The Amulet environment: new models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[67] R.E. Newman-Wolfe and H.K. Pelimuhandiram. MACE: A fine-grained concurrent editor. In *Proceedings of the ACM/IEEE Conference on Organizational Computing Systems (COCS 91)*, pages 240–254, Atlanta, Georgia, November 1991.

[68] D.A. Norman. Design rules based on analyses of human error. *Communications of the ACM*, 26(4):254–258, 1983.

[69] D.A. Norman. Stages and levels in human-machine interaction. *International Journal of Man-Machine Studies*, 21:365–375, 1984.

[70] A. Oram and S. Talbott. *Managing Projects with make*. O'Reilly and Associates Inc., 1993.

[71] R. Pike. The text editor Sam. *Software - practice and experience*, 17(11):813–845, 1987.

[72] A. Prakash and M. J. Knister. Undoing actions in collaborative work. In *Proceedings of the Fourth ACM Conference on Computer–Supported Cooperative Work* (Toronto, Canada, Oct 31-Now 4), pages 273–280, 1992.

[73] A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. Technical Report CSE-TR-125-92, Computer Science and Engineering Division, The University of Michigan, Ann Arbor, March 1992.

[74] B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing system design. *Computing Surveys*, 10(2):123–165, June 1978.

[75] S.P. Reiss. PECAN: program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, 1985.

[76] E.L. Rissland. Ingredients of intelligent user interfaces. *International Journal of Man-Machime Studies*, 21:377–388, 1984.

[77] W.R. Rouse and S.H. Rouse. Analysis and classification of human error. *IEEE Transactions on System, Man, and Cybernetics*, SMC-13(4):539–549, 1983.

[78] C. Rutkowski. An introduction to the human applications standard computer interface, Part 1: theory and principles. *BYTE*, 7(10):291–310, 1982.

[79] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction.* Addison-Wesley, 1987.

[80] D.C. Smith, C. Irby, R. Kimball, and B. Verplark. Designing the Star user interface. *Byte*, 7(4):242–282, 1982.

[81] M. Spenke and C. Beilken. An overview of GINA — the generic interactive application. In *User Interface Management and Design* D.A. Duce et al., Eds., pages 283–303. Springer-Verlag, Berlin, 1990.

[82] R. Stallman. *GNU Emacs manual.* Version 17, Free software foundation. Inc., 1986.

[83] W. Teitelman. *INTERLISP Reference Manual.* Xerox PARC, Palo Alto, Calif., Dec. 1975.

[84] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14:25–33, April 1981.

[85] L. Tesler. The Smalltalk environment. *Byte*, 6(8):90–147, 1981.

[86] H. Thimbleby. Character level ambiguity: consequences for user interface design. *International Journal of Man-Machine Studies*, 16:211–225, 1982.

[87] H. Thimbleby. *User Interface Design.* Addison-Wesley, 1990.

[88] R. Thomas and J. Yates. *A user guide to the Unix system.* Osborne/McGraw-Hill, Berkeley, California, 1982.

[89] W.F. Tichy. RCS–A system for version control. *Software-Practice and Experience*, 15(7):637–654, July 1985.

[90] H. Toriya, T. Satoh, K. Ueda, and H. Chiyokura. UNDO and REDO operations for solid modelling. *IEEE computer graphics and applications*, 6(4):35–42, 1986.

[91] J.M. Van Eekhout and W. B. Rouse. Human errors in detection, diagnosis, and compensation for failures in the engine control room of a supertanker. *IEEE Transactions on System, Man, and Cybernetics*, SMC-11(12):813–816, 1981.

[92] J.S.M. Verhofstad. Recovery techniques for database systems. *Computing Surveys*, 10(2):167–195, June 1978.

[93] J.S. Vitter. US&R: A new framework for redoing. *IEEE Software*, 1(4):39–52, 1984.

[94] M.I. Vuskovic. R-shell: a UNIX-based development environment for robotics. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation, vol.1*, pages 457–460, 1988.

[95] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[96] G. Williams. The Lisa computer system. *Byte*, 8(2):33–50, 1983.

[97] Y. Yang. Undo support models. *International Journal of Man–Machine Studies*, (28):457–481, 1988.

[98] Y. Yang. *User-oriented design of undo support.* PhD thesis, Heriot-Watt University, Department of Computer Science, October 1989.

[99] Y. Yang. Experimental rapid prototype of undo support. *Information and Software Technology*, 32(9):625–635, November 1990.

[100] C. Zhou and A. Imamiya. Object-based nonlinear undo model. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference, COMPSAC'97*, pages 50–55, 1997.