REVI : A RECREATABLE TEXT EDITOR

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

R. SRIDHARAN

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

January 1995

Contents

Acknowledgements

I take this opportunity to thank my project supervisor, **Dr. Jayant R. Haritsa**, Assistant Professor, Supercomputer Education & Research Centre, Indian Institute of Science (IISc), Bangalore, for his excellent guidance and persistent encouragement throughout the course of this work. I am fortunate to have had the opportunity to work in association with him and under his guidance.

I am grateful to the Chairman of our department, *Prof. N. Viswanadham*, for allowing me to work in SERC department for my project work. I am thankful to *Dr. S.K. Ghoshal* and *Mr. T.S. Mohan* of SERC department for providing valuable suggestions at various stages of my project work. I would also like to thank all the staff members of CSA department for their cooperation.

Several people have helped me in one way or the other and made my stay in the Institute a pleasant and unforgettable experience. I would like to thank SwathyPrabhu who has always cleared my doubts whenever I approached him, RajaMahesh for his pleasant friendship and Akhil Sahai with whom I have spent many enjoyable moments. I would also like to mention MBK, Balki, Gopi and gsri in this regard. I am leaving out several others for want of space.

Finally, and most importantly, I thank my parents, elders, my sister and my family to whom I dedicate this work.

Abstract

Anyone who has read a book has usually read it over a period of time. People typically read a set of pages in a book, set it aside and then pick it up again at a later time. To help them continue reading from where they had left off previously, bookmarks are used (usually a cardboard strip). The ScreenMark Project currently under way at the Indian Institute of Science is intended to provide a similar facility for users of shared computing resources. Such shared environments are typically found in educational and research laboratories wherein several workstations are connected by a network-based file system and users can log in to any workstation.

In shared environments, a user is required to log out from his workstation when he finishes his current session. When he next uses the computing lab, he may want to continue from where he left off the previous time. To do this, he has to manually recreate the earlier environment from scratch. This is both cumbersome and error-prone. Therefore, a facility that automatically reproduces the earlier screen context would benefit these users.

Reproducing the screen context has two aspects: syntax reproduction and semantic reproduction. Syntax reproduction refers to recreating the layout of windows, icons, and other graphical objects. Semantic reproduction, on the other hand, refers to recreating the *processes* that are executing in the windows. The goal of the ScreenMark project is to provide a syntax-cum-semantic screen–reproducing facility.

The first step in this project was to design a recreatable text editor and this is currently functional. The text editor **REVI** is recreatable in the sense that the exact state of the editor process at the time of logout is recreated when the user logs in later. We hope that the experience gained in this design will aid in extending the ScreenMark concept to other utility programs.

Chapter 1

INTRODUCTION

1.1 Motivation

Assume that you are preparing the final draft of a paper on a computer terminal in a shared workstation environment, that is, an environment in which users could work at the console of any computer connected over the network. You have opened several windows on the screen and are executing different processes in each of these. For example, in window 1, you are editing the final draft, in window 2, you are running a simulation program and so on. You have an important appointment and you realize that you have to leave. You expect to be back in an hour or so and would like to continue working from where you left off when you return. If you log out, this is not possible since you have to recreate your environment from scratch when you start working again. If you lock your screen, then that machine cannot be used by any other user. Both these options are not appealing and instead you would like to have a facility which automatically recreates your earlier environment without causing inconvenience to other users.

The goal of our project is to provide such a screen–reproducing facility. A recreatable text editor has been designed as part of achieving the goal of screen reproduction.

1.2 General Overview

A common computing environment consists of many workstations connected together by a high speed local area network. Such environments could be divided into two categories, namely, shared laboratories and personal workstations. In a shared lab environment (figure 1.1), workstations are shared by all the users as opposed to the personal workstation environments where each user has his own personal workstation. Sharing here means that users could work at the console of any workstation connected across the network and that the users' home directories are exported across the machines. A facility that could automatically reproduce an earlier screen environment would greatly help users who frequently log in and log out of machines each day. This behaviour is particularly seen in the Sun and Iris workstation labs at the Supercomputer Education and Research Centre (SERC) in the Indian Institute of Science.

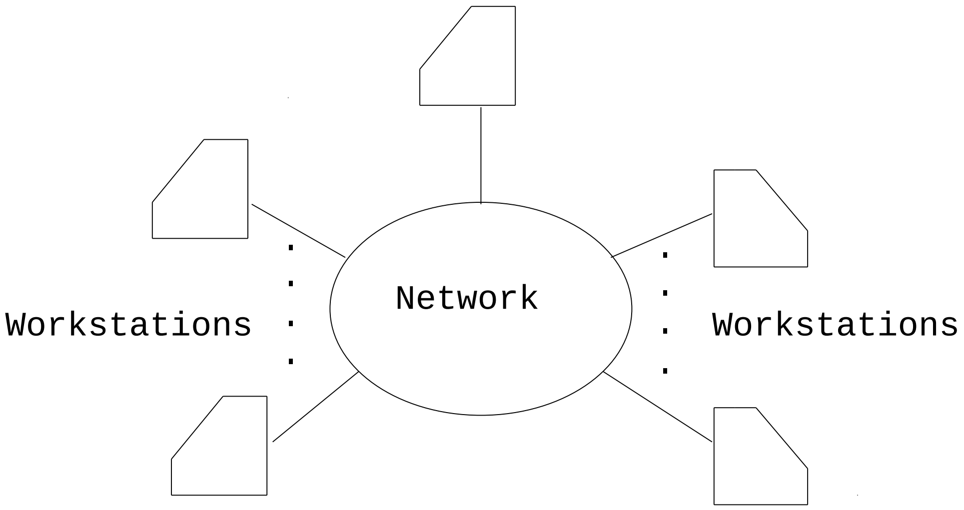


Figure 1.1: Network of Shared Workstations

A typical screen configuration consists of the layout of windows, icons and other graphical objects. The layout can be easily reproduced by using existing window system utilities. However, our definition of screen reproduction involves not just the layout of windows, icons etc. but also the *processes that were executing at logout* in each of these graphical constructs. In this sense, it is an extension of MIT's Project Athena [Arfman 92] where only file and resource access are made transparent to the user's location.

Another way to view the above functionality is in terms of the lockscreen [Sun 86a] command. In this technique, the user locks his screen using the lockscreen command. When the user types his passwd later, the screen is unlocked and the previous state is restored. Screen reproduction is equivalent to a lockscreen that is transported across machines. It is different, however, from the normal lockscreen in that it allows for continued user access to the original machine and it allows users to freely move across machines. In fact, it makes the normal lockscreen command redundant.

1.3 UNIX and Restartability

As mentioned earlier, screen reproduction involves restarting the processes that were running at the time of logout. The following is a brief account of the existing options for restartability supported by the UNIX[1] operating system.

A limited amount of restartability is provided by the UNIX operating system itself, through the use of suspend and continue signals. When a user types Control-Z, the currently running process is suspended and its state is saved by UNIX. The user can later restart the process either in the foreground or background by using the shell command fg or bg respectively. The major drawback of this scheme is that it works only within a login session and not across logins and logouts. This is due to the assumption made by UNIX that all the process-related resources are still in main memory. When the user logs out, all the resources such as page tables, pages etc. associated with the suspended process are released and hence the operating system would not be able to restart the process when the user logs in again.

Another option is to run all the processes using nohup before logging out. This is also not the ideal solution since in that case the system resources used by the processes are not released. This means that the user is still working even after he[2] logs out. Moreover,

[1] UNIX is a registered trademark of UNIX Systems Laboratories Inc.
[2] He should be read as he/she throughout this report.

interactive jobs cannot be nohup'ed since the user would like to return to the point at which he was working before.

Another facility available with standard text editors (such as vi) for restartability is the recover (-r) option. Suppose the system crashes when a user is editing a file. Then when the system later comes up, he can retrieve the file he was editing by using the -r option to vi. While this method recovers *most* of the file text, it *does not* reproduce the *exact state* of the original editor process. For example, if the user has saved some lines in an internal buffer (known as "yanking") (or) if he had undone some of his recent changes, then they are typically lost when the user recovers his file later.

Yet another facility available with most workstations is the lockscreen [Sun 86a] command. This command locks the screen and the user can later unlock it by entering the password and the previous state is restored. This is an acceptable solution in personal workstation environments but not in shared workstation labs. The problem is that if a user locks his screen, then that machine cannot be used by any other user. Another problem with locking the screen is that the technique is not portable across machines. That is, if the user locks a particular screen, he has to come back to the same machine to continue working. This is restrictive in a Network File System (NFS [Sandberg 85]) environment where the user has transparent file and resource access to any networked machine.

Thus the ideal solution would be to provide a facility similar to lockscreen but which does not lock the screen and is portable across machines.

1.4 The ScreenMark Project

The ScreenMark Project [Hari 94] aims to provide a screen–reproducing facility which is transparent to the user and portable across machines. A recreatable text editor, **REVI (RE**creatable **VI)**, has been designed as a first step towards achieving the goal of screen reproduction. The editor is recreatable in the sense that if a user opens several windows and is editing different files when he logs out, the windows are recreated with their original positions and sizes and the editor processes are also recreated when he logs in again.

The recreatable text editor has been designed in such a way that even the internal buffer contents used by the editor are saved and an *exact* reproduction of the previous state of the process (including the position of the cursor in the file) is obtained. Moreover, the design is portable across machines in the sense that the user can log in to any machine and get back his previous logout state with respect to the text editor processes. Since even the cursor is brought back to its old position, an illusion that the user had never logged out from the original machine is created, i.e. "it is as if time had stood still in the interim period."

The recreatable editor uses the **checkpointing** mechanism. Checkpointing is the saving of the state of a program during its execution so that it can be restarted at any time and on any machine in the system. In our design, all the active editor processes are checkpointed at the instant of logout. Additional information regarding the window configuration such as the size of the windows, their **x-y** positions, etc. is stored. This information is later used to recreate the editor processes when the user logs in again.

The above design has been implemented on Sun-3 and Iris workstations. The future plans include extending the concept of recreation to other utility programs.

1.5 Limitations

There is a limit upto which the concept of recreation is applicable. For example, remotely executed commands (e.g. rlogin) cannot be recreated since they are dynamically dependent on the network state at that time.

Sometimes, it may be physically impossible to completely reproduce the screen. This can happen, for example, if a remote login machine has gone down in the period between a logout and the subsequent login.

In general, processes that communicate with other processes and those processes which take actions that depend on the knowledge of the execution environment (such as the process id), cannot be recreated from an arbitrary earlier state. However, there are very few applications that fall in this category and hence the design for recreation should be applicable to most applications.

1.6 Organization of Dissertation

The remainder of this dissertation is organized in the following fashion: Chapter 2 surveys published research on remote execution and process migration. Chapter 3 provides an overview of the design methodology and a discussion of the relevant issues. The focus of Chapter 4 is on the detailed description of the implementation and discussion of the overhead involved. Finally, Chapter 5 summarizes our work and outlines future research directions.

Chapter 2

OVERVIEW OF RELATED WORK

2.1 Introduction

Networks of workstations have increased in great numbers in recent years. Considerable work has been done to improve productivity in modern processing environments that consist of large collections of workstations interconnected by high capacity networks. In this chapter, we survey some of the work done in this area.

2.2 Project Athena

MIT's Project Athena [Arfman 92, Treese 88] was the first successful experiment at supporting distributed computing across a network of workstations. It was launched in 1983 to address the Institute's need to provide widely available computing resources to the campus. It was implemented on Berkeley UNIX 4.3 BSD and was portable across IBM and DEC platforms. However, its goals were limited to providing transparent file and resource access.

After Project Athena, there has been lot of work done towards effectively utilizing computing capacity in networks of workstations. This work has been conducted in three areas: the analysis of workstation usage patterns, the design of remote capacity allocation algorithms and the development of remote execution facilities.

In the first area of research, workstation usage patterns and their availability as sources of remote execution have been analyzed [Mutka 87b]. An analysis of a group of workstations over 5 months done by the authors showed that only 30% of the total capacity of the workstations was utilized. This makes workstations good candidates to serve as sources of remote processing cycles.

The second area of research is the exploration of algorithms for the management of idle workstation capacity [Mutka 87a]. In a system where long running background jobs are scheduled on idle workstations, it has been observed that some users try to acquire all the capacity available, while others only acquire capacity occasionally. The *Up-Down* algorithm presented by Mutka and Livny was designed to allow fair access to remote capacity for those who lightly use the system in spite of large demands by heavy users.

The third area of research, namely, the development of remote execution facilities is described in the next section.

2.3 Remote Execution Facilities

Research on development of remote execution facilities has been going on for several years. A number of papers have reported on the development of systems that allow for remote execution of jobs on idle workstations. These include the V-kernel, the Process Server, the Butler system and the Condor system. The following brief survey of these systems is based on [Theimer 85], [Hagmann 86], [Nichols 87], [Bricker 89] and [Litzkow 88].

- **The V System**: The V-system [Theimer 85] consists of a distributed kernel and a distributed collection of server processes. A functionally identical copy of the kernel resides on each host and provides address spaces, processes that run within these address spaces and network–transparent interprocess communication (IPC). Low–level process and memory management functions are provided by a *kernel server* executing inside the kernel. In addition, there is a *program manager* on each workstation that provides program management for programs executing on that workstation.

A V program is executed on another machine at the command interpreter level by typing:

<program><arguments>@<machine-name>

Any program can be executed remotely providing that it does not require low–level access to the hardware devices of the machine from which it originated. Hardware devices include disks, frame buffers, network interfaces and serial lines. A suite of programs and library functions are provided for querying and managing program execution on a particular workstation as well as all workstations in the system. Facilities for terminating, suspending and debugging programs work independently of whether the program is executing locally or remotely.

- **The Process Server**: The Process Server [Hagmann 86] is an experimental system for distributing some of the computation normally performed on a workstation to some collection of servers. It was built in an environment with powerful workstations, remote procedure calls (RPC [Birrell 84]), networking (including gateways and servers for names, authentication and files) and an open system running on the workstations. The design of the Process Server is such that workstation owners will be able to contribute some part of their machine for load sharing but also will be able to withdraw the machine or restrict its use by others.

  Three types of entities cooperate to provide the Process Server: the *Client*, the *Server* and the *Controller*. Any workstation can be a Client and can ask for computations to be performed. Some number of machines have Servers and a group of them form a *Cluster*. The Cluster is managed by the Controller, which is a distinguished Server.

  Collections of software, called *packages*, are entered into the cluster. Each package implements a set of commands, the execution of which is called a *request*. The front end of a command line interpreter gathers parameters for a request and the request is passed to the Client on the workstation. After checking with the local Controller, the Client contacts the Server. The Server fetches from a file server the code for the packages. The request is then passed to the package. During execution of the request, the Client is used by the Server to do file name resolution, to perform some file operations and to answer questions about the user's environment. Eventually the request returns from the Server and thus the call on the Client now also returns.

- **Butler System**: The Butler System [Nichols 87] is a set of programs running on Andrew workstations at the Carnegie–Mellon University that give users access to idle workstations. The system maintains a list of the workstations that are in the pool of available machines. When a user wishes to use one of these workstations, he types

  `rem` <*program*>

  The `rem` program finds one of the idle workstations and arranges to have the command executed on it. These facilities are also available to C programmers via a subroutine library.

  When a machine is donated to the pool of available machines, it runs a program called `butler`. This program is responsible for adding and removing the machine from the global free machine registry and for handling requests for remote execution on that machine.

  The `rem` program first contacts the registry to find a candidate machine to use. The registry returns the name and net address of the potential server machine. The `rem` program contacts the candidate machine and checks to make sure that it is available. If the machine is available, the `butler` marks it as in use and removes its name from the free machine registry. The `rem` program then sets up the remote execution environment, invokes the command and waits for it to exit. When the client is done, it releases the machine and its `butler` returns the machine to the free pool.

  When the machine is reclaimed by its owner, usually by logging in at the console, the `butler` warns and kills any guest processes that may be running, removes the machine's name from the global registry and exits.

- **Condor System**: The Condor System [Bricker 89, Litzkow 88] identifies idle workstations and schedules background jobs on them. When the owner of a workstation resumes activity at a station, Condor checkpoints the remote job running on the station and transfers it to another idle workstation. The checkpointing of a program is the saving of an intermediate state of a program so that its execution can be restarted from this intermediate state. This enables successful completions of jobs that consume months of CPU capacity. When a program is terminated, either due to the shutdown of a remote workstation, or intentionally by the remote workstation's owner, the program is resumed from its most recent checkpoint.

  When the remote execution facility is explicitly invoked in the Condor system, a *shadow* process runs locally as the surrogate of the process running on the remote machine. Any UNIX system call made by the program on the remote machine invokes a library routine which communicates with the shadow process. A message indicating the type of system call is sent to the shadow process on the local machine and can be viewed as a remote procedure call. When someone resumes using a workstation that is executing a remote job, the job is checkpointed and moved to another idle workstation.

None of the above systems deal with the handling of restartable processes effectively. For example, when a user reclaims a workstation in the Butler system, the remote job that is currently running on the station is terminated and all intermediate results are lost. Moving on to the Condor system, we find that it uses checkpointing for background jobs. Our principal improvement over the Condor system has been to apply the checkpointing technique to interactive jobs (e.g. text editor).

2.4 Process Migration

Process Migration is another area of research that has received considerable attention in the last few years. Process Migration refers to the capability in distributed computing environments to dynamically relocate processes between processors. The primary motivation for providing this facility is load balancing. When the workload is evenly distributed, other benefits such as greater throughput and reduced overheads are obtained. Process migration normally requires considerable cooperation and support from the underlying operating system software. It is performed by taking a "snapshot" of the process on a processor and reinstating the snapshot on a different processor.

A number of implementations have been reported (e.g. [Alonso 88, Mand 88]) but almost all of these involve modification of the operating system kernel to some extent. We have implemented our design of the recreatable text editor *completely outside the kernel*. Users can run the editor on a normal UNIX system.

2.5 Checkpointing

Checkpointing and rollback recovery have long been used as a standard technique for providing a fault–tolerant computing environment. With checkpointing, if a fault is detected by the computer system, the application is then rolled back to the most recent checkpoint where the state of the system is known to be correct. Checkpointing schemes can be broadly divided into two categories, depending on how the checkpoint information is maintained. *Full checkpointing* saves the entire state space of the application for every checkpoint. *Incremental checkpointing* creates major checkpoints and minor checkpoints. Major checkpoints save the entire state space and minor checkpoints save the difference between the current and previous state spaces.

Recently researchers have made use of compiler–based techniques for implementing full checkpointing. Examples include compiler–assisted static checkpoint insertion [Long 92], the CATCH compiler [Li 90] and compiler–assisted full checkpointing [Li 94]. However, the overhead incurred by the programmer in such schemes may be significant.

The next chapter focusses on the design of our recreatable text editor.

Chapter 3

DESIGN OF REVI

3.1 Introduction

A recreatable text editor, **REVI**, has been designed as a first step towards achieving restartability in the general case. The text editor was chosen for our implementation since it is commonly used and is one of the familiar utility programs for most users. This chapter discusses the issues involved and describes the design methodology in detail.

3.2 Two phases of Screen Reproduction

Ideally the user would like the system to automatically reproduce his earlier screen environment exactly. This could be divided into two distinct phases, namely, syntactic reproduction and semantic reproduction.

In syntactic reproduction, the window configurations are stored and later restored. Implementing this feature requires storing the layout of windows, their positions, width, height etc. This could be done in a simple manner using existing utilities in a window system. For example, in the X Window System [Scheifler 86], the utility xwininfo gives all the relevant information about a window.

The semantic reproduction involves storing the exact state of a process and restoring it later. The state of a process consists of the text, data and stack regions [Bach 92].

Since accessing these regions of a process from another process is difficult, implementing semantic reproduction is more involved when compared to implementing syntax reproduction.

If these two phases could be linked in a meaningful fashion, then recreation could be achieved. This ideal option is illustrated in figure 3.1. Investigating the feasibility of this ideal solution is the primary aim of our project.

3.3 Design Goals

The important design aims which have governed our implementation of recreation are outlined below:

- **Syntax Reproduction**: The syntax information would have to be reproduced. This means that the windows have to be recreated at their original positions with their original sizes.

- **Semantic Reproduction**: In addition to the syntactic information, semantic information would have to be reproduced. This implies restoring the exact state of a process.

- **Low Overhead**: Low overhead and low response time should be achieved. Overhead includes disk space needed to store the information needed for recreation. From the user's perspective, the response time should not be degraded. These issues are explored in greater detail in the next chapter.

- **Transparency**: The design should be transparent to the user. That is, if the windows are recreated with their original positions and sizes and the processes are also recreated when the user logs in again, then from his perspective, it should be as though he had never logged out from the original machine, i.e. an illusion that time had stood still in the interim period should be created.

- **No Source Modification**: Recreation should be implemented outside the processes, without any source code modification. But this leads to severe problems, as

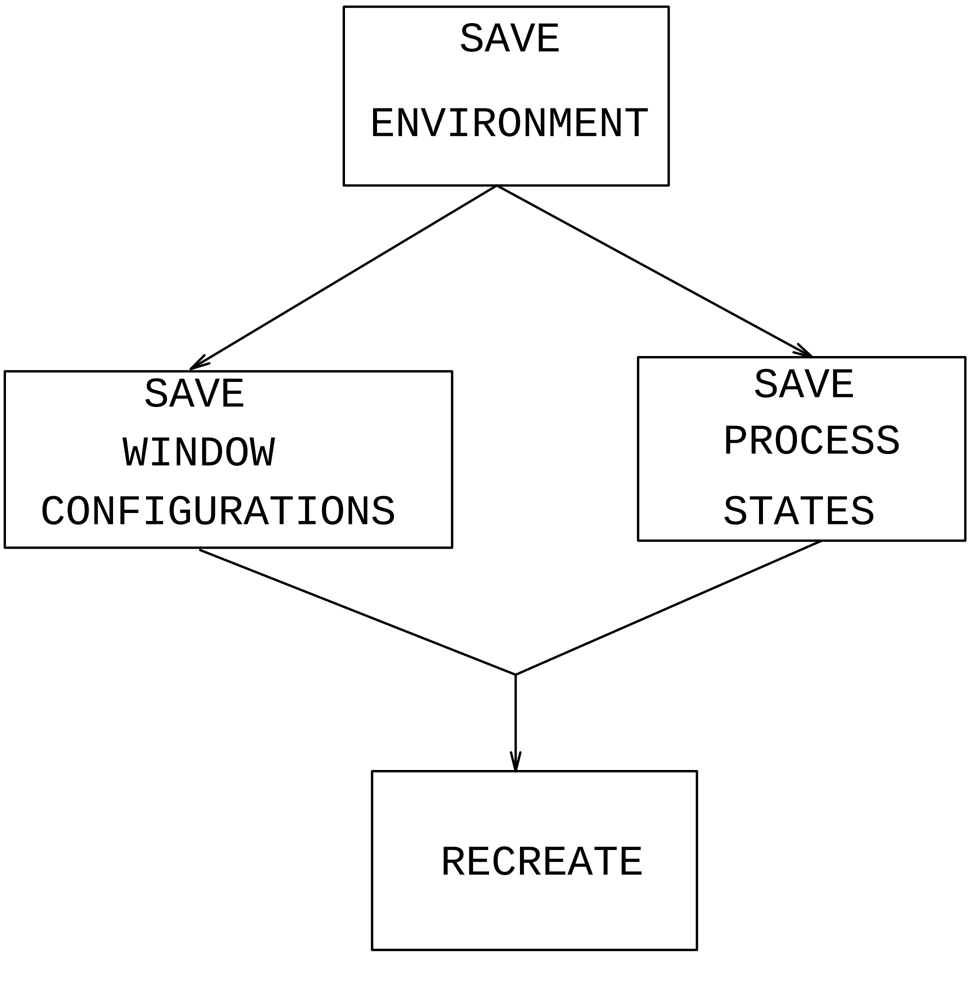


Figure 3.1: The ideal option

explained in Section 3.4.1.

- **User-Level Implementation**: Implementation of recreation should be done completely at the user level, without any modification of the operating system kernel.

3.4 Design Approaches

There are two possible approaches to designing a recreatable text editor, namely, Exo–process and Endo–process. These approaches are described in the following subsections.

3.4.1 Exo–Process(Outside the Process)

The first approach is to implement recreation entirely outside the process, without modifying the source code. However, it leads to several practical difficulties:

Firstly, the only way to control the execution of a process and to read its data and stack regions in older versions of UNIX is to use the `ptrace()` system call. But this requires the controlling process to be the parent of the controlled process. This means that the user should run all his processes as children of special processes which take care of recreation. Thus the goal of transparency is defeated and hence this method is not suitable.

The second technique of controlling a process is to use the file system interface to processes [Killian 84]. This is called the **/proc** file system and is provided in recent releases of most flavours of UNIX. In this file system, images of all the active processes are stored as files, with the filename being *XXXXX* where *XXXXX* is the pid of the currently running process. Special system calls are provided to access the regions of a process through this file interface. But accessing this file system leads to frequent system crashes (in SERC) and hence this technique is not reliable.

Moreover, interactive programs like the text editor have to initialize the terminal settings and tty modes for proper editing. If recreation is done outside the process, then if the terminal state has changed in the time between a logout and the subsequent login, the original terminal settings might no longer be valid and re-initialization is needed. For example, the *curses* package [Sun 86b] used by the text editor has to be initialized at the beginning of the edit session to achieve proper scrolling.

Hence implementation of recreation completely outside the editor process without any source code modification is practically infeasible at the user level.

3.4.2 Endo–Process(Minimal Source Modification)

If the source code of the editor program is available, then it could be modified at suitable places to facilitate recreation. One instance where this modification is needed is with respect to the input mode of editing. That is, if the user had inserted some text in the file just before logging out, then this text should be recreated when he logs in again. If source modification is not done, there is no way of knowing from outside the editor process what text had been inserted. Hence special functions have to be added to the source code of the editor program in order to save this information. Another instance where source modification is needed is the case of editing multiple files. That is, if the user is editing several files with a single invocation of the editor program at the time of logout, then during recreation, the user has to be brought back to the file he was editing just before logging out. This implies that some amount of source modification is needed to ensure reproduction of the *exact* state of the edit session as far as the user interface is concerned.

This approach, namely, a minimal amount of source modification, has been realized in our implementation of **REVI**.

3.5 Design Methodology

As described in Section 3.2, recreation involves both syntactic and semantic reproduction. The design methodology employed for reproducing both syntax and semantics is outlined in the following subsections.

3.5.1 Syntax Reproduction

The syntax information is obtained by running the X utility xwininfo on all the open windows and storing the necessary information in a file. This is done whenever the window size changes so that the latest information regarding the window configurations is obtained. The stored information includes the sizes of the windows and their **x-y** positions. This information could later be used to recreate the windows at their original positions with their original sizes.

3.5.2 Semantic Reproduction

Semantic reproduction is done through the technique of **checkpointing**. Checkpointing is the saving of the state of a program during its execution so that it can be restarted at any time and on any machine in the system. In this technique, a signal is sent to the editor process that is running and a core file is obtained. This core file is dumped by the kernel and contains the image of the process as it existed when it received the signal. The data and stack regions are obtained from the core file and the text region is obtained from the original executable. These are combined to produce a checkpoint file, which is itself executable. When the checkpoint file is executed later, it starts executing from the saved state of the process.

Our definition of semantic reproduction with respect to the text editor processes is that there is no perceptible difference to the user as far as the functions of the editor commands he had invoked before logout are concerned. This means that even though the *exact* state of the editor process (at the level of the register contents and the machine instruction being executed at the time of logout) is not being recreated, an illusion that the user had never logged out from the machine is created through the technique of checkpointing.

The link between the syntactic and semantic information is also stored in a file. This is implemented by storing the command name and arguments of the editor process and the geometry information of the window in which the process was originally running.

3.5.3 Recreation

The windows are first recreated based on the saved configuration information. The editor processes that were executing in those windows are then recreated by forking off the requisite number of processes and executing the checkpoint files, instead of the original executable file. Since the checkpoint file starts executing with the saved data and stack regions, the previous state of the process (including the cursor position) is exactly restored.

3.6 Advantages

Some of the important advantages of **REVI** are summarized below:

- The primary advantage of our design is that continued user access is provided to the original machine. That is, the machine is not locked once the user saves his environment and logs out. Any other user could log in to the machine and start working. This is the principal improvement over the lockscreen option referred to earlier.

- The system resources used by the editor process are released as soon as the user logs out. Hence there is no trace of the user's program state in memory. This is in contrast to the nohup option where the resources are not relinquished and the user continues to work, even after he logs out.

- Another important advantage is the portability across homogeneous machines. When a user logs out of a machine for a brief period, he should not be restricted to the same machine when he wants to continue his work later, since that machine might no longer be available. The user should be able to continue working on any networked machine. This portability is especially advantageous in an NFS environment, where the user has transparent file and resource access.

The implementation details of **REVI** are described in the next chapter.

Chapter 4

IMPLEMENTATION DETAILS

4.1 Introduction

Having presented the design goals and the design methodology of the recreatable text editor in the previous chapter, the implementation is described in greater detail in this chapter. The design has been implemented on Sun-3 workstations running UNIX SunOS Release 3.5 and supporting the X Window System and also on Iris workstations running IRIX Release 4.0.5. The description that follows is based on the implementation on Sun workstations. It was later ported to Iris workstations with minor modifications related to the operating system and addition of menu options for saving and recreation.

4.2 The Elvis Text Editor

As indicated in Chapter 3, recreation has been implemented with minor modifications to the basic text editor source code. The text editor elvis [Kirk 93] was chosen for this purpose since it is available as free software. Its interface is identical to that of the standard text editor vi.

4.3 Modifications to Elvis

The elvis source code was modified so that the temporary file used by it when the user edits a file is created in the user's home directory instead of the local **/tmp** directory of the machine. This is done to achieve portability across machines assuming that the users' home directories are exported to all the machines across the network.

A new option of -x was added to elvis. This option is used during recreation to do proper initialization. Initialization includes opening the original file which the user was editing with a new file descriptor since the old file descriptor would no longer be valid as explained in the previous chapter. A flag was added to indicate whether the current invocation of elvis is normal or whether it is being invoked during recreation. This information is needed to do proper clean-up when the user quits the editor since the new file descriptor has to be closed instead of the original file descriptor.

Whenever a window is resized, the operating system sends the signal SIGWINCH [Bach 92] to the process that is currently running in that window. A signal handler for the SIGWINCH signal was added to the basic elvis source code. This is needed to obtain the latest configuration information regarding the resized window.

A signal handler for the signal SIGHUP was also added to the elvis source code in order to store information about the current working directory of the editor process before the user logs out.

A function was added to store the id of the window in which the editor process is running. This is obtained from the environment variable WINDOWID inherited by the editor process. This information is later used by the signal handler for SIGWINCH to obtain the configuration information for the window using the appropriate id.

A function was added to determine whether the user was in input mode at the time of logout. This information is stored in a temporary file. This is later used to bring the user back to the input mode during recreation.

4.4 The SAVE Program

A program called SAVE is run at the time of logout from the user's login (or) console window. This program creates a file called *scrn_mark_file* in the user's home directory and saves the syntactic and semantic information about all editor processes that are running at the time of logout. It then destroys all the windows, kills itself and brings the user back to the console shell prompt. The user can now logout in the usual fashion. The way in which this program saves the relevant information is explained in Sections 4.6 and 4.7.

4.5 The RECREATE Program

The RECREATE program is run from the X Windows initialization file, namely, *.xinitrc*. This program is activated when the user invokes X Windows after logging in. The contents of the file *.xinitrc* are shown in figure 4.1.

```
twm &
/user7/csasrdn/proj/test-mult/RECREATE &
xterm -geometry  85x24+5+4
```

Figure 4.1: Initialization file for recreation

The initialization file starts up the default window manager, namely, `twm`. The RECREATE program which recreates the windows and editor processes running in them is then executed. Finally the terminal emulator program `xterm` is started up which acts as the console window (or) login window. The working of the RECREATE program is explained in Section 4.9.

4.6 Saving Syntax Information

The syntax information consists of the configurations of the open windows. This configuration information can be obtained by using the X utility `xwininfo` with the appropriate window id as its argument.

Whenever a window is resized, the editor program currently running in the resized window receives the SIGWINCH signal from the operating system and executes the associated signal handler. This signal handler catches the signal, runs `xwininfo` on the resized window and stores the information in a file in the **/tmp** directory. The name of this file is *wininfo_XXXX* where *XXXX* is the pid of the editor process.

The SAVE program run at the time of logout reads the temporary file containing window configuration information and extracts only the relevant information, namely, the position, width and height of the window. These are then combined to make up a string which is given as an argument to the terminal emulator program `xterm` during recreation. The format of the string is **WxH+X+Y** where **W** denotes the width of the window, **H** the height, **X** the x–position and **Y** the y–position.

4.7 Saving Process State

Recreating a process implies restoring the previous state of the process. This means that the process state has to be saved when the user logs out. The state of a process consists of the contents of its address space and the contents of hardware registers related to the process [Bach 92]. This state information is saved through the technique of **checkpointing**. This technique is described below in greater detail.

4.7.1 Checkpointing

To checkpoint a UNIX process, several things must be preserved. The text, data and stack regions are needed in addition to the register contents. The data, stack and register contents are available in a core file while the text is available in the original executable.

4.7.2 Creating the checkpoint file

The SAVE program sends the signal SIGHUP to all the editor processes that are running at the time of logout. A signal handler for this signal has been incorporated in the editor source code. This signal handler saves information about the current working directory of the process in a temporary file. The name of this temporary file is *cwd_file_XXXX* where *XXXX* is the pid of the editor process. The signal handler then sends itself the signal SIGTRAP, which results in a core dump. The SAVE program then combines the original executable file and the core file to produce a "checkpoint" file (figure 4.2). The checkpoint file is itself executable.

4.7.3 Structure of the Checkpoint file

The checkpoint file is constructed as an executable file with the format shown in figure 4.3.

In this format, the data and stack segments are obtained from the core file and all other information is obtained from the elvis executable. The elvis executable file consists of a structure called *exec* at the beginning of the file whose elements are defined in the header file */usr/include/sys/exec.h* and are shown in figure 4.3. The *exec* structure gives the sizes of the various segments of the executable. For example, the field *a_text* in the *exec* structure gives the size of the text segment and it is located immediately after the *exec* structure in the elvis executable. Similarly, the offsets and sizes of the other segments are determined and this information is used to extract these segments from the executable.

4.7.4 Structure of the Core file

In order to extract the data and stack segments from the core file, the structure of the core file should be known. The structure is defined in the header file */usr/include/sys/core.h* and is outlined in figure 4.4.

The core file consists of a header called *core* at the beginning of the file whose members are shown in figure 4.4. This header is followed by the images of the data and

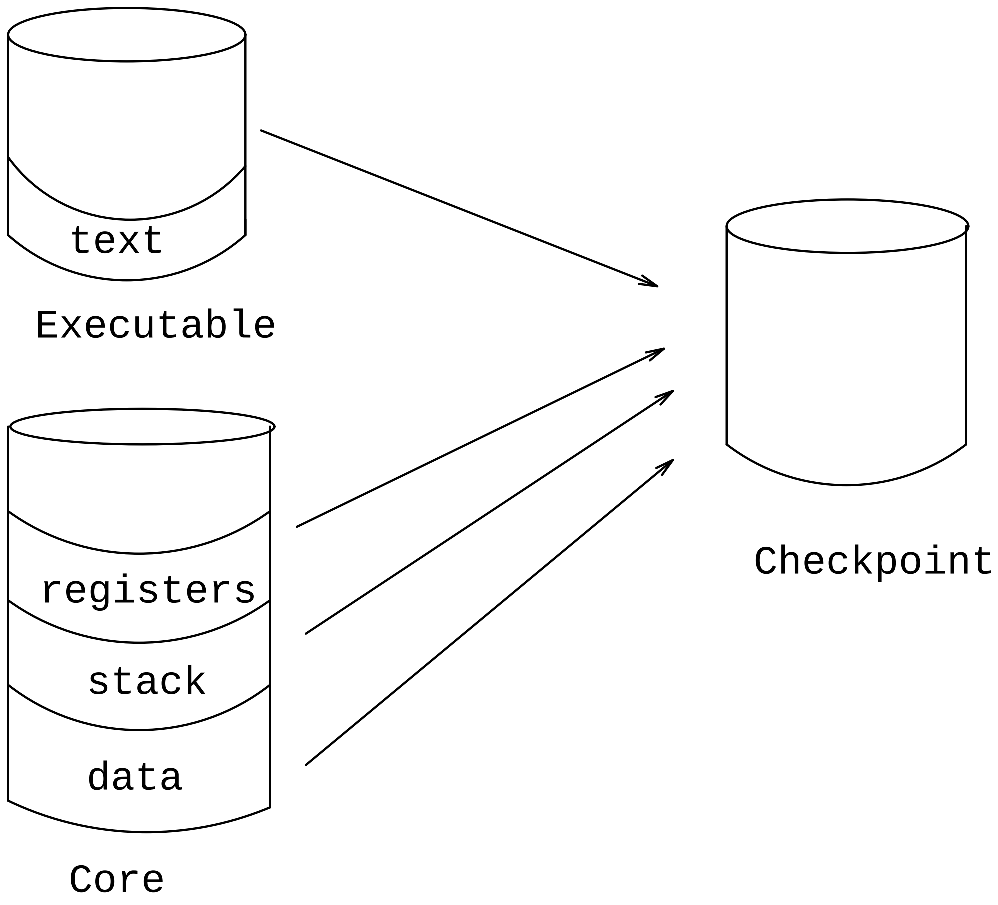


Figure 4.2: Creating a Checkpoint File

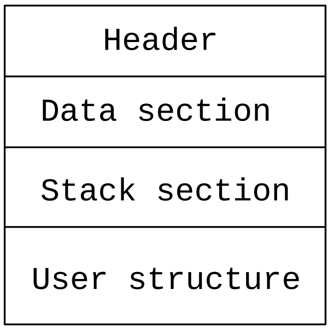
Exec Structure
Text Segment
Data Segment
Text Reloc. Info
Data Reloc. Info
Symbol Table
String Table
Stack Segment

FORMAT OF THE CHECKPOINT FILE

```
struct exec {
  unsigned short a_machtype;
  unsigned short a_magic;
  unsigned long a_text;
  unsigned long a_data;
  unsigned long a_bss;
  unsigned long a_syms;
  unsigned long a_entry;
  unsigned long a_trsize;
  unsigned long a_drsize;
}
```

The Exec Structure

Figure 4.3: Structure of the checkpoint file (in Sun machines)



STRUCTURE OF CORE FILE (IN SUN MACHINES)

```
struct core {
        int c_magic;
        int c_len;
        struct regs c_regs;
        struct exec c_aouthdr;
        int c_signo;
        int c_tsize;
        int c_dsize;
        int c_ssize;
        char c_cmdname[CORE_NAMELEN + 1];
        struct fp_status c_fpstatus;
        struct fpa_regs c_fparegs;
        int c_pad[CORE_PADLEN];
}
```

The Header Structure

Figure 4.4: Structure of the core file

stack regions of the process at the time of the core dump. Finally, the *user* structure of the process (defined in the header file */usr/include/sys/user.h*) is dumped in the core file. Based on the header information, the text size, data size and stack sizes of the process at the time of the core dump are determined. These are given by the fields *c_tsize*, *c_dsize* and *c_ssize* in the structure *core* respectively. This information is used to locate the data and stack segments in the core file. The segments are then extracted from the core file.

4.7.5 Writing the Checkpoint file

Before writing the extracted information into the checkpoint file, the *exec* structure of the original elvis executable is first read in. The data sizes stored in the *exec* structure are then updated based on the information gathered from the core file. The updated *exec* structure, the segments obtained from the elvis executable and the data and stack segments obtained from the core file are then written into the checkpoint file in the format shown in figure 4.3.

When the checkpoint file is later executed when the user logs in again, it starts executing from the saved data and stack regions and hence the previous state of the editor process is restored.

4.8 Linking Syntax and Semantics

The link between syntactic and semantic information is maintained by the SAVE program in the *scrn_mark_file*. A sample *scrn_mark_file* is shown in figure 4.5.

The format of the *scrn_mark_file* is as follows: Firstly, the number of editor processes that were active at the time of logout is stored. For each such process, the directory in which the process was executing and the geometry information about the window in which the process was running are stored. This information is necessary since the previous state restoration includes bringing the user back to the directory in which he was working before. In addition, all the command line arguments are stored since the processes have to be recreated with the same set of arguments. Finally, the name of the

```
2
11402
/user0/tmp/srid/condor-irix
newloop.c
90x22+13+138
xwsh
11375
/user0/tmp/srid/elvis-irix
new
107x22+178+543
xwsh
```

Figure 4.5: A sample scrn_mark_file

terminal emulator program in which the editor process was running is stored.

This information is later read by the RECREATE program which recreates the windows and the corresponding processes that were originally running in them.

4.9 Recreation

The RECREATE program run from the X Windows initialization file first reads the *scrn_mark_file* in the user's home directory and determines the number of editor processes that were originally running. For each editor process, it first changes its own current working directory to the directory in which the process was originally running at the time of logout. It then forks off a terminal emulator program `xterm` with the editor process running inside it with the appropriate arguments. While recreating the editor process, the corresponding checkpoint file is executed instead of the original executable which ensures that the previous state (including the cursor position) is restored. Also, the geometry of the xterm window is specified based on the saved window geometry information.

This procedure is repeated for all the editor processes that were running at the time of logout. Hence, from the user's perspective, he comes back to the state at which he was working before and hence he could now continue working.

The sequence of steps involved in saving and recreation are illustrated in figure 4.6. An illustration of the state of a window being saved and recreated along with the editor process that was originally running in it is provided in figures 4.7 to 4.9. Figure 4.7 illustrates an editor process running inside an xterm window. Figure 4.8 shows the SAVE program being executed from the console (or) login window. Figure 4.9 shows the state of the window after recreation when the user logs in later. It could be seen that the previous state of the editor process (including the position of the cursor in the file) is exactly restored upon recreation except for some minor differences which are explained below: In order to emphasize that the position of the cursor has been restored, the status of the file after recreation has been shown by typing Ctrl-G explicitly. Also, according to the semantics of the xterm implementation, the name of the window is based on the process that is run at the time of creation of the window. Since the editor process is run as a child of the shell during recreation, the name of the window after recreation becomes sh instead of xterm.

4.10 Observations

Each editor process has an associated checkpoint file which is approximately the size of the address space of the process. Disk space **must** be available to store the checkpoint file.

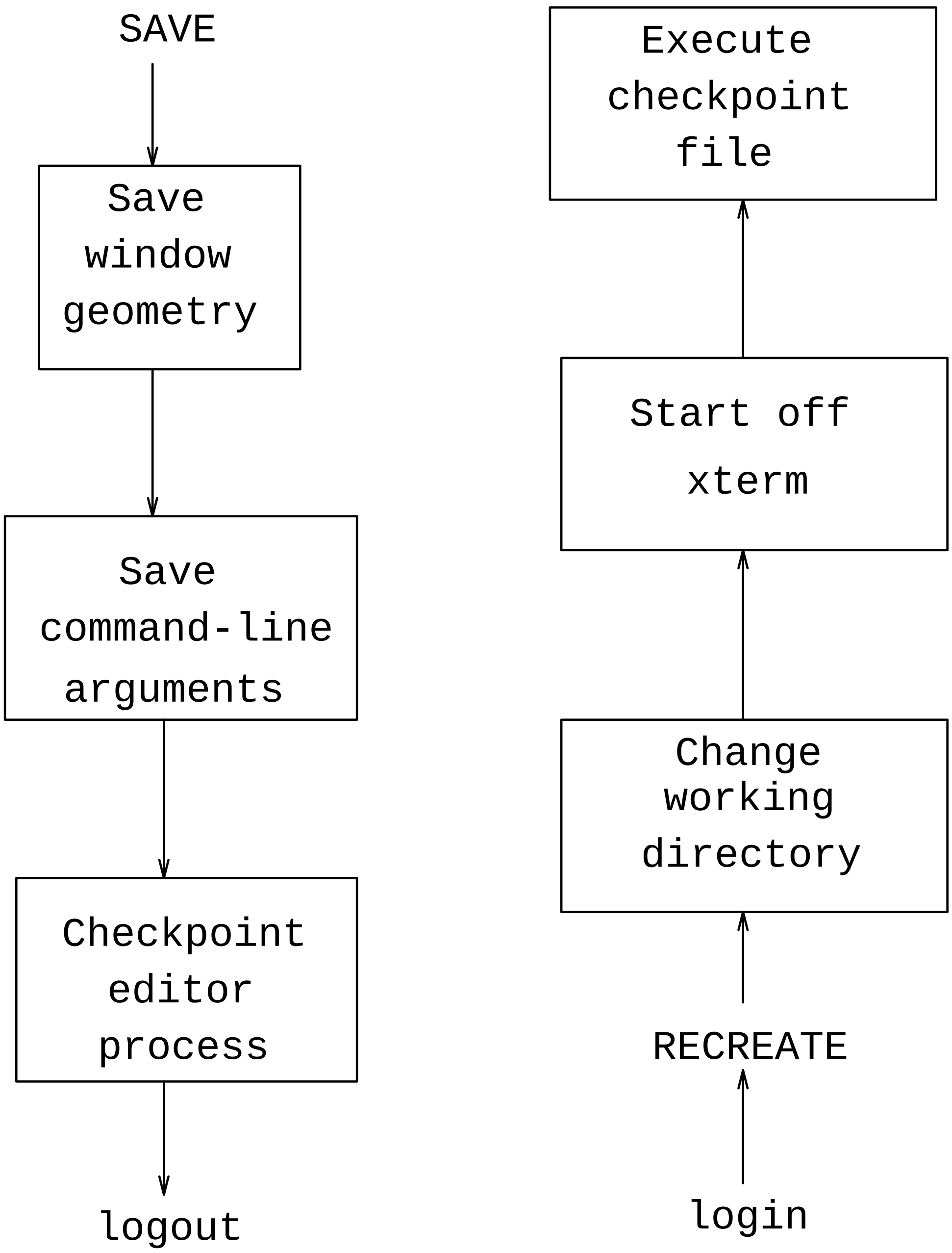


Figure 4.6: Saving and Recreation

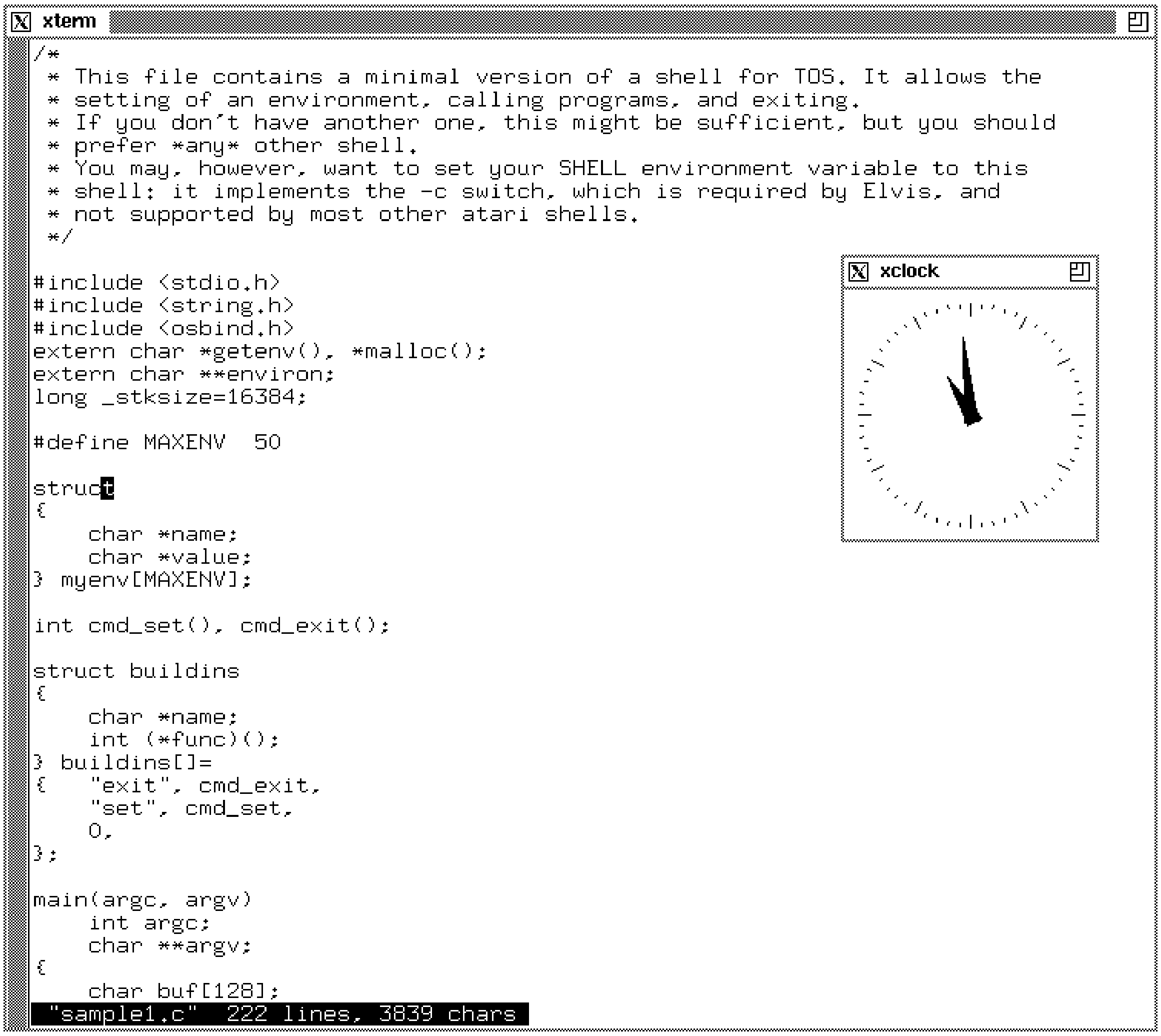


Figure 4.7: State of an xterm window before logout

4.10.1 Compression

As mentioned earlier, the size of the checkpoint file depends on the size of the address space of the process. This in turn depends on the size of the file the user is editing. We have observed that the average size of the checkpoint file is about 1/2 megabyte when the user is editing files of sizes 1000–5000 bytes. Hence, compression has been used to save disk space. Using the UNIX utility compress, the checkpoint file size is reduced to about one-third of its original size [Welch 84]. For example, if the checkpoint file size is 400 KB, after compression, the size gets reduced to about 150 KB. Compression is at the control of the user. The user could choose to compress his checkpoint files by giving the -c option to the SAVE program. The RECREATE program searches for a checkpoint file ending with the suffix .Z in the directory in which the editor process was running before. If it finds one, then it first uncompresses the file using the UNIX utility uncompress and then uses it for recreation of the editor process. If it does not find one, then it uses the checkpoint file directly for recreation.

4.10.2 Response Time

The response time, as seen by the user, depends on the size of the file he is editing and the system workload at that point in time. Typically, when the user is editing a 4000–byte file and with no other users on the system, the SAVE program takes 10 seconds of elapsed time (wall-clock time) to save the information pertaining to a single window. The RECREATE program takes 5 seconds of elapsed time to recreate the window with the editor process running. This time proportionately increases with the number of windows that are active at logout.

If the user specifies the compress option (-c) to the SAVE program, the response time is degraded since it involves running the compression algorithm on the checkpoint file. Hence there is a trade–off between disk space overhead and response time.

4.11 Salient Features of REVI

Some of the salient features of REVI are outlined below:

- Editing of multiple files is supported, i.e. if the user is editing several files with a single invocation of the editor program at the time of logout, then upon recreation, the user is brought back to the particular file he was editing at the time of logout.

- Multiple saves and recreations are possible. Suppose the user has saved the state of his editor process and has logged out. When he logs in again, the state of the edit session is recreated. If he makes some more changes to the file he was editing and again saves and logs out, then, when he logs in next time, the recreation is done with respect to the most recent saved state of the editor process.

4.12 Requirements to be satisfied by the user

Saving and recreation of text editor processes works under certain conditions (or) requirements imposed upon the user:

- The RECREATE program assumes that the *scrn_mark_file* exists in the user's home directory and that it follows the format given in figure 4.5. If this file is modified (or) deleted, then recreation would not be achieved. Similarly, the checkpoint file used to recreate the previous state of the editor process should not be altered.

- The initialization files for the X Window System should be properly defined to support saving and recreation. For example, the RECREATE program should be executed from the *.xinitrc* file in the user's home directory in order to achieve automatic screen recreation with respect to the text editor processes.

- The implementation is based on dumping a core image of the process in the current working directory. Hence the user should have write and execute permissions in the directory in which the editor program is invoked, as well as sufficient disk quota to store the checkpoint file.

- Since the code was designed to be portable across homogeneous machines, the directory in which the user invokes the editor program should be an NFS–mounted directory. If the directory is local to a particular workstation, then portability would not be achieved even though proper recreation would be achieved.

4.13 Limitations

The following are some of the limitations of the recreatable text editor that have been observed:

- Nested editor sessions are not supported. That is, if the user invokes another editor session inside a currently running editor session, then the second session would not be recreated.
- Saving and recreation works only for those editor processes which are running in the foreground at the time of logout. For example, if the user has suspended the editor process (using Ctrl-Z), then such processes would not be recreated. This is because of the fact that the process ceases to receive any signal other than the SIGKILL signal once it is suspended and hence a core image of such a process cannot be dumped.

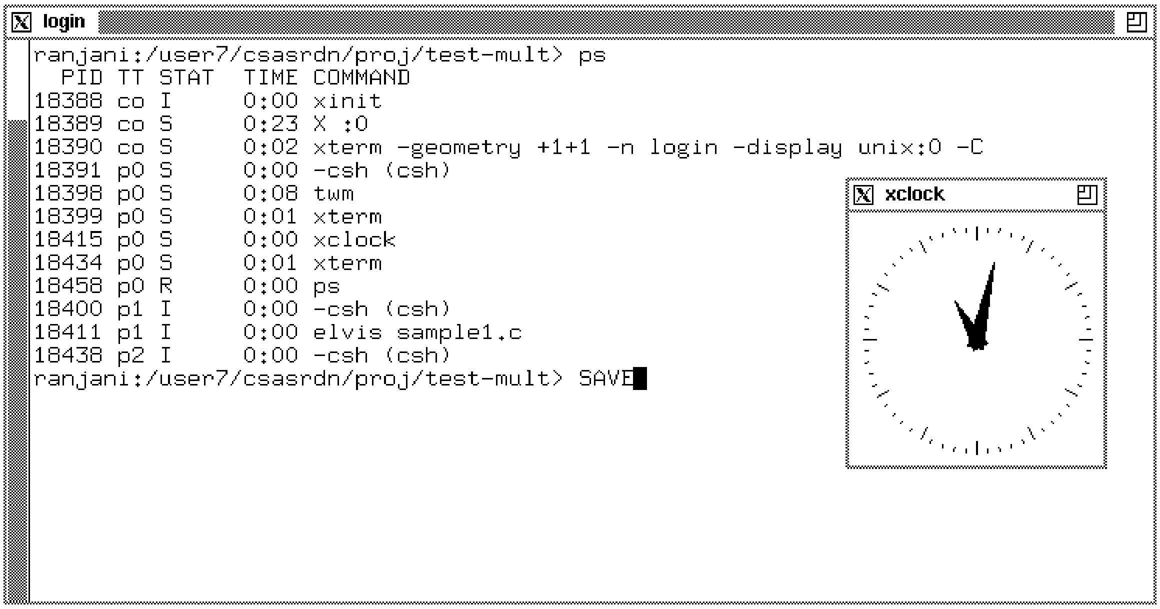


Figure 4.8: State of the console window at logout

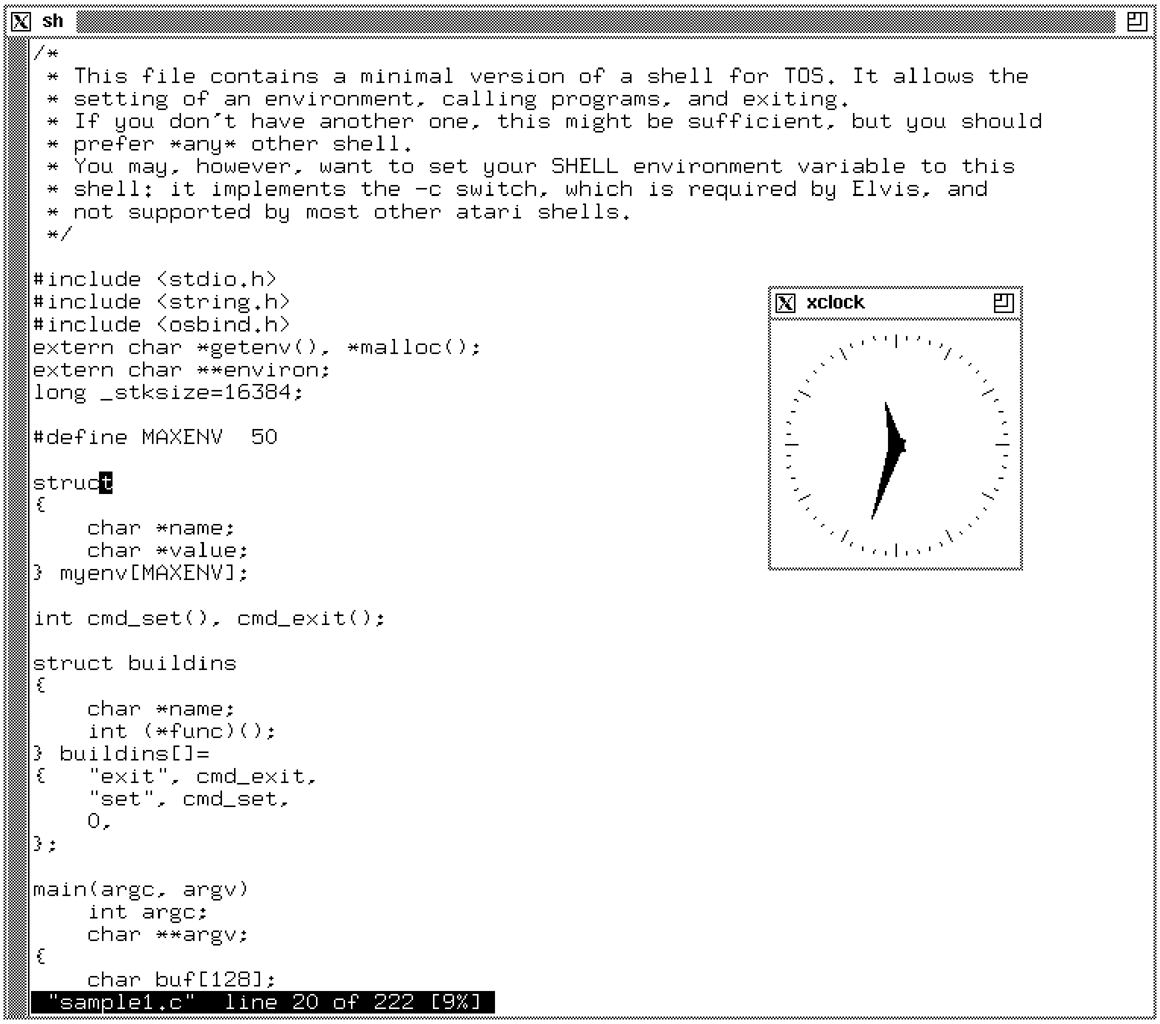


Figure 4.9: State of the xterm window after recreation

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Networks of powerful workstations have become prevalent in modern computing environments. These networks of workstations provide the possibility of expanding the capacity available to a user beyond his/her local workstation to that of the entire network. A facility that reproduces an earlier screen environment would greatly benefit the users, especially in shared workstation labs where the users could log in to any workstation and where transparent file and resource access are available to the users.

In this project, a recreatable text editor has been implemented as part of investigating the feasibility of the aforesaid facility. The editor recreates the exact state of its previous execution including the position of the cursor in the file the user was editing.

5.2 Future Work

Further work is required in a few areas to improve the applicability and effectiveness of the system. Our current interest is in extending the concept of recreation to other utility programs and later to arbitrary processes. This could possibly be done through two different approaches:

- Special stub programs could be designed which store the state information of a process from time to time so that the process could be restarted from its most recent saved state. These stubs could be activated by a special library linked with the user programs. The advantage of this approach is that it does not require modification of the user program although it requires the user to relink his code with the special library.
- Another approach would be to access the pages, page tables and other auxiliary information about a process by reading kernel virtual memory, physical memory and the raw disks. This information could be stored on disk and could be used later to reinstate the pages back in memory when the process starts up again. This approach would be transparent to the application but requires modifications to the operating system kernel since accessing the pages and reinstating them is not possible at the user level.

Other possible extensions to our work include reducing the cost of checkpointing and recreation and trying to recreate processes that have subprocesses or those that use communications sockets.

Bibliography

[Alonso 88] Alonso, R. and Kyrimis, K., "A Process Migration Implementation for a UNIX System", *Proceedings of the Winter 1988 USENIX Conference*, USENIX Association, February 1988.

[Arfman 92] Arfman, J.M. and Roden, P., "Project Athena: Supporting Distributed Computing at MIT", *IBM Systems Journal*, Vol.31, No.3, 1992.

[Bach 92] Bach, M.J., *The Design of the UNIX Operating System*, Prentice Hall of India Private Limited, New Delhi – 110 001, 1992.

[Birrell 84] Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol.2, No.1, February 1984.

[Bricker 89] Bricker, A. and Litzkow, M., "Condor Technical Summary", Computer Sciences Dept., University of Wisconsin, 1989.

[Hagmann 86] Hagmann, R., "Process Server: Sharing Processing Power in a Workstation Environment", *Proceedings of the Sixth IEEE Distributed Computing Conference*, May 1986.

[Hari 94] Haritsa, J. R., *ScreenMark Project Description*, SERC, 1994.

[Killian 84] Killian, T.J., "Processes as Files", *Proceedings of the Summer 1984 USENIX Conference*, USENIX Association, 1984.

[Kirk 93] Kirkendall, S., *elvis-1.7 documentation*, Public Domain Software, 1993.

[Li 90] Li, C.-C. and Fuchs, W.K., "CATCH: Compiler Assisted Techniques for Checkpointing", *The Twentieth Annual International Symposium on Fault-Tolerant Computing*, June 1990.

[Li 94] Li, C.-C., Stewart, E.M. and Fuchs, W.K., "Compiler–assisted Full Checkpointing", *Software – Practice and Experience*, Vol.24, No.10, October 1994.

[Litzkow 88] Litzkow, M., Livny, M., and Mutka, M., "Condor — A Hunter of Idle Workstations", *Proceedings of the Eighth International Conference on Distributed Computing Systems*, June 1988.

[Long 92] Long, J., Fuchs, W.K. and Abraham, J.A., "Compiler–assisted static checkpoint insertion", *The Twenty-Second Annual International Symposium on Fault–Tolerant Computing*, 1992.

[Mand 88] Mandelberg, K.I. and Sunderam, V.S., "Process Migration in UNIX Networks", *Proceedings of the Winter 1988 USENIX Conference*, USENIX Association, February 1988.

[Mutka 87a] Mutka, M.W. and Livny, M., "Scheduling Remote Processing Capacity in a Workstation–Processor Bank Computing System", *Proceedings of the Seventh International Conference on Distributed Computing Systems*, September 1987.

[Mutka 87b] Mutka, M.W. and Livny, M., "Profiling Workstations' Available Capacity for Remote Execution", *Performance'87, Proceedings of the Twelfth IFIP WG 7.3 Symposium on Computer Performance*, December 1987.

[Nichols 87] Nichols, D.A., "Using Idle Workstations in a Shared Computing Environment", *Proceedings of the Eleventh Symposium on Operating System Principles*, November 1987.

[Sandberg 85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B., "Design and Implementation of the Sun Network Filesystem", *Proceedings of the Summer 1985 USENIX Conference*, USENIX Association, 1985.

[Scheifler 86] Scheifler, R. and Gettys, J., "The X Window System", *ACM Transactions on Graphics*, Vol.5, No.2, April 1986.

[Sun 86a] *lockscreen Manual Pages*, Sun Release 3.5, Sun MicroSystems Inc., 1986.

[Sun 86b] *curses Manual Pages*, Sun Release 3.5, Sun MicroSystems Inc., 1986.

[Theimer 85] Theimer, M.M., Lantz, K.A. and Cheriton, D.R., "Preemptable Remote Execution facilities for the V–system", *Proceedings of the Tenth Symposium on Operating System Principles*, December 1985.

[Treese 88] Treese, G.W., "Berkeley Unix on 1000 Workstations: Athena Changes to 4.3 BSD", *Proceedings of the Winter 1988 USENIX Conference*, USENIX Association, February 1988.

[Welch 84] Welch, T.A., "A Technique for high–performance data compression", *IEEE Computer*, Vol.17, No.6, 1984.