Stateful Task Recovery
in the
UNIX Operating System

by

Robert A. Petrick

Submitted to the
Department of Aeronautics and Astronautics
in partial fulfillment of the requirements
for the degree of

Master of Science in Aeronautics and Astronautics
at the
Massachusetts Institute of Technology

May 1995

© 1995 Robert A. Petrick
All rights reserved

The author hereby grants to MIT permission to reproduce and to
distribute publicly copies of this thesis document in whole or in part.

Signature of Author

Robert A. Petrick
Department of Aeronautics and Astronautics, May 1995

Approved by

Richard E. Harper
Charles Stark Draper Laboratory
Thesis Supervisor

Certified by

Professor John J. Deyst
Department of Aeronautics and Astronautics
Thesis Advisor

Accepted by

Professor Harold Y. Wachman
Chairman, Department Graduate Committee

JUL 07 1995
LIBRARIES
Stateful Task Recovery
in the
UNIX Operating System

by

Rob A. Petrick

Submitted to the Department of Aeronautics and Astronautics
on May 12, 1995
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Aeronautics and Astronautics

ABSTRACT

This research presents a very general approach for recovering failed tasks
running on a Byzantine resilient fault tolerant parallel processor which runs the
UNIX operating system. Although general, this approach is believed to be
superior to other fault tolerant systems such as Tandem NonStop, Argus, and
TARGON/32.

In stateful task recovery, the subject of this thesis, one halts a UNIX task,
creates another task, copies critical state structures from the halted task, and then
writes the task state into the newly created process through implementing a new
version of the exec() system call. Two programs called test.c and draper_code.c
and one "h" file contain all the code which enables recovery. Test.c operates in
user mode and enters into kernel mode through the system calls read_state() and
write_state(). Read_state() and write_state() are system calls specifically created
for stateful task recovery and are linked into the kernel. Read_state() copies the
critical data structures from the halted application process, test.c saves the copied
states into files, and write_state() then calls my_exec(). The system call my_exec()
copies the halted task's saved state into the newly created process.

Recovery copies the text, data, and user stack into the restarted process.
The UNIX clock() function calculated the total CPU time for recovery to be .133
seconds. A simple counting program is the test application process used for
recovery. When recovery is initiated, the recovered process begins counting, not
at 1, but at the number where the application process was halted. At the present
state of development, the recovery process can not successfully copy critical
elements such as the program counter, stack pointer, and frame pointer;
therefore, the recovered process does not begin at the desired instruction.

Thesis Supervisor:  Dr. Richard E. Harper
                    Principal Member Technical Staff, C.S. Draper Laboratory

Thesis Advisor:    Dr. John J Deyst
                    Professor of Aeronautics and Astronautics, M.I.T.
Acknowledgments

First of all, I must definitely thank Rick Harper for all of the support and help he has given me. I was a lost soul approximately one year ago when Rick offered to be my thesis supervisor. I am sure he still regrets that decision, but I know I do not. I would have to say that the experience of working with someone so knowledgeable has given me much more than any educational institution could offer. I just have one last comment, Rick, “find . -name "*.csh" -exec grep “life” {} \; -printf I more.” In addition, my gratitude is extended to my thesis advisor at MIT, Professor John Deyst.

I would also like to thank Carol Babikyan and James Beaupre for their guidance through the anfractuous UNIX operating system and for their time they spent helping me with my research.

My thanks and congratulations go out to Dan Stine and Matt Lobner who have been very good company around the office, and who also are finishing their theses as well. Thank-you both for breaking up the boredom which comes with complete and utter confusion.

Finally, I would like to acknowledge my mom, dad, and two sisters for all the love and support they give to me. They have always been the most encouraging people in my life, and I hold them in the highest esteem.
This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under the US Navy Omnibus/FTM Contract number N00030-93-G-0074-EK12.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

Robert A. Petrick

Permission is hereby granted by The Charles Stark Draper Laboratory, Inc., to the Massachusetts Institute of Technology to reproduce any or all of this thesis.
# Table of Contents

## CHAPTER ONE

1.0 Introduction .......................................................... 13
1.1 The Fault Tolerant Parallel Processor, AFTA .................................................. 14
  1.1.1 Byzantine Resilience .................................................. 15
  1.1.2 AFTA Configuration .................................................. 16
  1.1.3 AFTA Configuration for Stateful Task Recovery .................................. 18
1.2 Motivation ............................................................. 19
  1.2.1 Advantages Over Memory Realignment ........................................... 21
1.3 Fault Tolerant Distributed Systems Using UNIX ........................................... 23
1.4 Outline ........................................................................... 26

## CHAPTER TWO

2.0 The UNIX Operating System .................................................. 29
2.1 UNIX Architecture .......................................................... 29
2.2 Kernel Architecture .......................................................... 30
2.3 Memory Management .......................................................... 33
  2.3.1 Virtual Memory .......................................................... 34
2.4 A Process Defined ............................................................. 36
  2.4.1 Process Creation and Execution ............................................... 37
  2.4.2 Process Data Structures ................................................... 39

## CHAPTER THREE

3.0 Existing Approaches to Process/Task Recovery ........................................... 43
3.1 Transaction-Based Recovery ...................................................... 43
  3.1.1 Tandem “NonStop” ....................................................... 44
  3.1.2 Argus ........................................................................... 45
  3.1.3 Transaction Processing Kernel (TPK) ........................................... 45
  3.1.4 Transaction-Based Fault Tolerance Implemented on MachTP ............... 45
3.2 Checkpoint-Rollback Based Recovery .................................................. 46
  3.2.1 Optimistic Recovery ....................................................... 46
  3.2.2 Pessimistic Recovery ....................................................... 47

## CHAPTER FOUR

4.0 Advantages of Stateful Task Recovery .................................................. 49
4.1 Proposed Design for Stateful Task Recovery ............................................. 49
4.2 Initial System Configuration ......................................................... 55
4.3 Process Setup on 486 .................................................................. 56
  4.3.1 The Defining Attributes of a Process ........................................... 59
  4.3.2 Phase 1 of Stateful Task Recovery ............................................. 62
  4.3.3 Phase 2 of Stateful Task Recovery ............................................. 64

## CHAPTER FIVE

5.0 Detailed Design of Stateful Task Recovery ............................................. 67
List of Figures

Figure 1.1 AFTA Physical Architecture .................................................. 17
Figure 1.2 FTPP Architecture .............................................................. 19
Figure 1.3 FTPP Configuration for Recovery ........................................ 21
Figure 2.1 System Architecture ............................................................ 30
Figure 2.2 System Kernel ................................................................. 31
Figure 2.3 File Descriptor Table, File Table, and Inode Table .................. 32
Figure 2.4 File System Layout ............................................................. 33
Figure 2.5 Memory Hierarchy .............................................................. 34
Figure 2.6 Process Memory Mapping ................................................... 36
Figure 2.7 User and Kernel Stack ......................................................... 38
Figure 2.8 Data Structures for Processes ............................................. 39
Figure 2.9 Process State Transition Diagram ....................................... 40
Figure 2.10 Components of the Context of a Process ............................... 41
Figure 3.1 Atomic Three-way Message Transmission ........................... 47
Figure 4.1 FDIR Message Transmission and Receiving ........................ 50
Figure 4.2 Normal Allocation of Processes on FTPP ............................. 51
Figure 4.3 Fault in Application Process on Processor 1 ......................... 51
Figure 4.4 Correct Application Processes Halted .................................. 52
Figure 4.5 Fork New Application Process ............................................ 52
Figure 4.6 Initiate My_exec() ............................................................. 53
Figure 4.7 Save Correct Application Processes States In Files ................ 53
Figure 4.8 Vote Saved States/Place in Processor 1 ............................... 54
Figure 4.9 Write Saved State into New Application Process .................. 54
Figure 4.10 Begin all Processes Simultaneously ................................... 55
Figure 4.11 Normal Application Processes .......................................... 56
Figure 4.12 Fault in Application Process #2 ....................................... 56
Figure 4.13 Correct Application Process Halted ................................... 57
Figure 4.14 Fork New Application Process ......................................... 57
Figure 4.15 Initiate My_exec() ........................................................... 57
Figure 4.16 Save Correct Application Process State in File .................... 58
Figure 4.17 No Voting Necessary With Single Processor ....................... 58
Figure 4.18 Write Saved State into New Application Process ................. 58
Figure 4.19 Begin Application Processes Simultaneously ..................... 59
Figure 4.20 User Area Addressable in Kernel ..................................... 60
Figure 4.21 Critical Data Structures .................................................... 61
Figure 4.22 Run-time Structure of the Kernel ...................................... 63
Figure 4.23 Process State ................................................................. 64
Figure 4.24 Recovery Architecture Using Kernel .................................. 65
Figure 5.1 Phase 1 of Test.c: Read_state() ........................................ 68
Figure 5.2 Virtual Address Space Arrangement .................................... 69
Figure 5.3 Recovery Architecture Using Kernel ................................... 71
CHAPTER ONE

1.0 Introduction

Increasing demand for highly reliable and highly available distributed systems has necessitated the creation of fault tolerant computers utilizing the UNIX operating system. On-line transaction processing (OLTP) and real-time applications utilize distributed systems. These applications rely on fault tolerance to guarantee recovery from unpredictable and hazardous system failures or system service unavailabilities.

Many financial institutions, such as banks and stock exchanges, employ on-line transaction processing systems because a system failure could cause havoc. The telecommunications industry also relies heavily on continuously available systems because a computer failure in the telephone system could cause phone lines to be down for extended periods of time, and important, life critical communications would not be able to transpire.

Real-time applications include but are not limited to computer systems used for guidance, navigation, and control systems on aircraft, space vehicles, ground vehicles and many other types of transportation vehicles which could endanger human life if a failure were to occur. Some notable systems which have required a highly reliable computer system were the Apollo Guidance, Navigation, and Control (AGN&C) computer used on the Apollo missions to the moon, the NASA/Navy F-8 Digital Fly-by-Wire (DFBW) system developed for military aircraft, the Space Shuttle's data processing system (DPS), and the automatic landing computers incorporated into the Lockheed L-1011, Douglas DC-10, Boeing 747, and the Boeing 777 aircraft as well [HL94]. In addition to transportation vehicles, many institutions such as hospitals, airports, and nuclear power plants need highly reliable computer systems since a computer failure could cost lives.

In order to avoid such catastrophic faults, people design systems to provide both hardware and software fault tolerance. Hardware fault tolerance is achieved through the use of several processors which can be connected via some communications medium [Bab90]. Replicated tasks run on each processor, and a failure of one site does not affect the other normally operating, replicated sites [CN90]. Software fault tolerance, however, poses a much more difficult problem
for any system, and the problem must be solved through both the hardware architecture and operating system environment.

The UNIX operating system can provide many advantageous uses in ultra-reliable, mission-critical applications. For example, UNIX is a reasonably open system which facilitates its portability to an abundance of workstations. Additionally, UNIX has a standard user interface which makes application portability simple; in other words, users can port applications from one UNIX workstation to another UNIX workstation with ease. UNIX is a multi-user system which can run many processes while giving the appearance of running them simultaneously. Additionally, UNIX provides the user with necessary services such as a command line interpreter (shell), mail, text processing packages, and source code control systems to name a few. UNIX provides a hierarchical file system that allows easy maintenance and efficient implementation, and UNIX employs a consistent format for files, the byte stream, which is implemented for making application programs easier to write. The operating system provides primitives that permit complex programs to be built from simpler programs, and the operating system provides a simple, consistent interface to peripheral devices. Finally, UNIX is written in C, a high-level language which simplifies the system for many general uses [Bac 86]. Due to this open and standardized system, many prominent database management system vendors, such as Informix Software Inc., Oracle Corp., Relational Technology Inc., Sybase Inc., and Unify Corp. use UNIX for their specific applications.

The objective of this thesis work is to provide a general solution to the problem of restarting a failed task which is running under the UNIX operating system. In short, the research results will support hardware and software fault tolerance for replicated UNIX systems. The research will apply not only to Draper Laboratory's needs, but also to many applications, including both OLTP and real time applications, providing a fault tolerant environment for highly reliable and highly available systems.

1.1 The Fault Tolerant Parallel Processor, AFTA

The US. Army has supported and funded the project called AFTA (Army Fault Tolerant Architecture) which is a militarized version of the fault tolerant parallel processor (FTPP) developed by the Charles Stark Draper Laboratory, Inc. The FTPP was designed for computer systems which require ultra-high
reliability and high throughput. The FTPP is resilient to Byzantine faults [HBBLP94], and the architecture includes multiple processing elements in order to provide parallel processing capability. With these two design attributes, the FTPP achieves its ultra-high reliability and high throughput goals.

1.1.1 Byzantine Resilience

In order to understand the architecture of the AFTA, the concept of Byzantine resilience must be understood since it is central to the theory and operation of the AFTA. Byzantine resilience is an extremely high degree of fault tolerance which any system requiring cost-effective validatability and high reliability requires. Byzantine resilience is the capability of a computer system to withstand a Byzantine fault. A Byzantine fault is any arbitrary behavior of a failed component such as stopping and then restarting execution at a future time, sending conflicting information to different destinations, and other derelictions [HBBLP94].

Byzantine resilience leads to cost-effective validatability. For example, a Failure Modes and Effects Analysis based (FMEA-based) approach to validation posits likely failure modes of a system and their probable extents and effects. From this, each anticipated failure is then corrected by some sort of fault tolerant technique. Nevertheless, FMEA-based analysis is not very efficient:

The FMEA process is tedious, time-consuming, and extremely expensive. This is attested to by the seemingly contradictory trend of increasing costs of digital avionics systems even as the cost of hardware continues to decline. This is at least partially due to the fact that the cost of validating critical systems is completely overwhelming the cost of their design and construction. Software validation is a major component of this cost, and inappropriate fault tolerance-related architectural features can only aggravate its difficulty [HBBLP94].

In contrast, a Byzantine resilient system, or a system capable of tolerating Byzantine faults, does not have to rely on a priori assumptions about component misbehavior because the Byzantine system guarantees the toleration of all faults. Essentially, a component may fail or misbehave in any manner, and the system will tolerate these faults, thus, obviating the need for the more expensive and tedious FMEA-based test process.

The following is an analogy as to the origin of "Byzantine resilience":
Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement [LSP82].

The generals in the previous analogy correspond to processors in a redundant computer system, the traitors correspond to faulty processors, and the messengers correspond to interprocessor communications links. In short, a Byzantine resilient system has many powerful attributes which not only reduce software validations effort and cost, but also make the system extremely reliable for life and mission-critical applications. The following section describes the AFTA's architecture which enables it to be a Byzantine resilient computer system.

1.1.2 AFTA Configuration

Figure 1.1 is a diagram of the AFTA configuration. The following paragraphs describe the AFTA configuration [HBBLP94]. The AFTA architecture is made up of Non-Developmental Item (NDI) Processing Elements (PEs), Input/Output Controllers (IOCs), Power Conditioners (PCs), backplane/chassis assemblies, and specially designed hardware components called Network Elements (NEs). As seen on the diagram in Figure 1.1, there are clusters called Fault Containment Regions (FCRs) which are comprised of line replaceable modules such as PEs, NEs, IOCs, and PCs. Since the FCRs have independent sources of power, clocking, and dielectric and physical isolation from other FCRs, a fault in one will not cause a fault in another. A minimal AFTA configuration consists of a least four NEs and three PEs; a maximal system would consist of five NEs and forty PEs all depending on the specific system application.

The interconnection of the FCRs is done in a manner which makes transition from one standards suite to another very simple and with minimal hardware redesign. The FCRs are interconnected using one or more standardized backplane buses such as a VMEbus, SAVA SBBUS, PIbus, or Futurebus+ for example. Changes in any FCR backplane bus only affect the NE's
Fault Containment Region
- Independent Power
- Independent Clocking
- Dielectric Isolation
- Physical Isolation

Network Elements
- Voting
- Synchronization
- Message Passing
- Reconfiguration

Standard Bus
Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8

I/O Bus(es) (optional)
NE

High Speed Fiber Optic Network

Processing Elements
- NDI Components
- Application Software
- Ada Run Time System
- POSIX Run Time System
- Virtual Groups:
  S: Simplex
  T: Triplex
  Q: Quadruplex

Fault Tolerance Achieved by:
- Multiple processing elements, each in
- Separate fault containment regions
- Results voted via Network Elements over
- Fiber optic links

Figure 1.1  AFTA Physical Architecture
bus-dependent circuitry which is partitioned from its bus-independent circuitry allowing an easy transition when a change of backplane is needed.

The PEs resident in the AFTA are made up of a processor, private RAM and ROM, and miscellaneous support devices, such as periodic timer interrupts. The processor could be general-purpose or special-purpose such as a signal or image processor.

The NEs provide communication between PEs, keep the FCRs synchronized, maintain data consensus among FCRs, and provide dielectric isolation between the FCRs via fiber optic links. The NE implements the protocol requirements for Byzantine resilience [LSP82]. In short, the primary function of the NE is to exchange and to vote packets of data provided by the processors.

The IOCs connect AFTA to the outside world, and they must be compatible with the bus connecting elements of the FCR. They may have a programmable processor on board to drive the I/O, or they may require off-board processors for operation [HBBLP94].

1.1.3 AFTA Configuration for Stateful Task Recovery

The FTPP provides a superior environment to existing fault tolerant computer technology. First of all, its Byzantine resilience empowers the system with the ability to tolerate arbitrary failure behavior. Secondly, in order to enhance reliability in the system, the FTPP supports testability and redundancy management strategies which enable users to optimize the system for many different applications. This optimization is a very cost-effective aspect of the system. The testability of the system means that there exist extensive analytical models and predictive verification and validation techniques which allow any application designer to thoroughly test any configuration and guarantee that mission requirements will be met. Third, the architecture of the FTPP allows the addition of many processors depending on the application. A forth advantage of the system is that it is an open system built from Non-Developmental Items which reduce the cost and the risk of development while enabling simple modifications for different applications and upgrading. Finally, the open systems design philosophy was extended to the AFTA’s operating system and programming languages as well. A commercial off-the-shelf Portable Operating System Interface (POSIX)-compliant operating system was ported to the AFTA [HBBLP94]. The LynxOS/UNIX operating system necessitated the design for
stateful task (process) recovery, the topic of this thesis, in order to guarantee quick recovery from process failures.

The architecture used for Stateful Task Recovery is a simplified version of Figure 1.1. There are only four separate processors enabling redundant processing. Each processor has a network element enabling Byzantine-resilient synchronization, consensus, voting, and consistent ordering functions [HL91]. Each processor can communicate with its network element through its own device driver. In addition, each network element can then communicate amongst the other network elements in the system. Figure 1.2 illustrates the FTPP’s architecture.

Figure 1.2  FTPP Architecture

1.2 Motivation

Draper Laboratory has created fault tolerant systems which are used for many highly reliable and highly available system applications. This research is not to create a fault tolerant system but to improve upon an existent system.
In prior fault tolerant systems at Draper Laboratory, stateful recovery, also known as memory realignment, had been accomplished by simply suspending execution of the redundant computer set, voting the contents of the nonfaulty computer’s memory into the faulted computer’s memory, and resuming execution as a restored redundant group. This worked well for operating systems which could be guaranteed to have bitwise identical memory contents at all times, such as small real-time kernels running on clock deterministic hardware. However, stateful task recovery in a Byzantine resilient, fault tolerant computer running the UNIX operating system presents a much different problem. For example, UNIX is not a small real-time kernel, and the FTPP does not use clock deterministic hardware.

C1, an earlier FTPP, utilized memory realignment. A downloaded operating system, which was a deterministic system, guaranteed the developers that memory images were congruent (bitwise identical memory) on each processor at specific points in time, even though the FTPP was not clock deterministic.

UNIX, however, is a nondeterministic system which does not guarantee that memory storage will be identically congruent on each processor. Each process in UNIX has its own process ID (PID) number which differentiates it from any other process. Furthermore, each processor has its own disk, and the logical-to-physical memory mapping, represented, perhaps, by buffer cache structures, will probably point to different places on each disk. To be succinct, each processor’s buffer cache manager is unlikely to select the same disk block (physical memory) for memory storage of a given logical block. Quite often, a disk will have “bad blocks” where the computer can not store information. If one disk has information stored in a specific area of memory, and another disk has bad blocks at the identical space in memory, there is no possibility for the operating system to have bitwise identical memory at least in the data which maps logical to physical disk blocks. Also, modern UNIX operating systems identify and map out bad memory pages; this is not the same on all processors.

In addition to these problems, the fault tolerant parallel processor can not simply halt all executions at the same point in the instruction stream on all channels. In short, the FTPP is not a clock deterministic system, and it does not guarantee bitwise identical memory at specific points in time. A computer system’s hardware must be designed to achieve clock determinism which entails an extremely high cost of production.
Since the contents of each channel’s memory may never be bitwise replicas of each other, an approach is needed to halt just a replicated UNIX task on the nonfaulty channels, restart a UNIX task replica on a faulted channel, copy the task state from the nonfaulty copies to the restarted task, and resume task execution in synchrony with the nonfaulty copies. This method is called stateful task recovery, and it is the subject of this thesis. Figure 1.3 depicts the configuration for stateful task recovery which Chapter Four further explains.

![Figure 1.3 FTPP Configuration for Recovery](image)

1.2.1 Advantages Over Memory Realignment

The method of memory realignment used with C1 was inefficient for two main reasons. The first reason is that a processor’s entire memory, not just a specific process’ memory, had to be copied into the processor where the task had failed. Secondly, the method was extremely slow for all of the processors to reach memory alignment. The total time from a fault injection to complete recovery was 4.71 seconds [Babi90]. The proposed method of stateful task recovery in UNIX copies only the critical data structures from a nonfaulty process to the restarted task in order to achieve the exact state of where the failed process should have been. Of course if all processes are faulty, a worst case scenario, this method could take a long time also.

Another advantage other than speed is software fault tolerance. Since UNIX is nondeterministic, a properly designed, redundant, UNIX-based, fault tolerant computer can prevent both BohrBugs and HeisenBugs. BohrBugs are software defects that reliably crash operating systems given certain inputs.
These types of software faults are repeating faults which programmers can find and correct through running a process several times. The process will crash each time at the exact location making the task of correcting and recovering from the error relatively easy. A HeisenBug, on the other hand, is a software defect which is nondeterministic and intermittent. It is an elusive state and history-dependent bug that can crash the system on certain runs yet not on others [SW94].

The system on which stateful task recovery will be implemented, the FTPP (C3), has four identical processes which do not run in close synchrony and do not have bitwise identical memory; it is not a lock-step system. Basically, this means that the processes, although identical, may not complete instructions at the same time. For instance, if there were four processes A, B, C, and D, process A could be ahead, in its instruction stream, of process C which is ahead of process B which is ahead of process D. In addition to the gaps in time between each process’ instruction stream, the operating system sets up each process differently on the separate processors. For example, the timer may preempt each process at different places in their instruction streams, and the memory allocation and storage may be completely different on each processor. One could say that the “state trajectory” of each process is different, even though they execute the same instructions since they are identical programs.

Relating back to the four processes A, B, C, and D, if a BohrBug were to occur, one might occur, for example, in process A before any of the other processes. The system would detect and restart the failed process before all processes could crash because of the gap in time between each process’ instruction stream. In short, a BohrBug may not occur in all processes at the same time and fail the entire system. Even so, BohrBugs are unlikely to occur in a properly debugged program. However, HeisenBugs are more apt to occur since they are nondeterministic. Nevertheless, the system will be tolerant of HeisenBugs because each process will not be “directed” down the same path since they are set up differently in each processor.

In short, stateful task recovery on a nondeterministic replicated UNIX system can be superior to any other method of recovery currently in use. Where possible, we will achieve stateful task recovery without modifying the operating system code in order to maximize portability of the solution and to keep the solution simple. The ultimate outcome will be to achieve software and hardware fault tolerance and recovery in a distributed UNIX system through stateful task recovery.
1.3 Fault Tolerant Distributed Systems Using UNIX

In order to achieve fault tolerance, a computer must survive faults in both computer software and hardware. Users see fault tolerance as the availability and data consistency of the application [HK93]. The FTPP C3 achieves hardware fault tolerance through its Byzantine resilience. UNIX, nevertheless, is not an innate fault tolerant operating system; therefore, we need to build software fault tolerance into the system. Software implemented fault tolerance must detect errors, determine how to correct them, and finally make the corrections. Most faults in present-day applications have been due to software faults and operator errors, thus emphasizing the need for software fault tolerance [BW94].

There are many applications which use fault tolerant distributed systems in conjunction with the UNIX operating system such as communications, telecommunications, on-line transaction processing (data base management systems), factory automation, military and aerospace applications, instrumentation, and many more. The construction of RISC (reduced instruction set computer) microprocessors has enabled more CPU (central processing unit) power for many of the previously listed real-time and embedded applications. However the limiting aspect of these applications is the software implementation. Therefore, UNIX has had a rising popularity along with RISC technology. Because of the open system and easy portability of UNIX, people can simply port this operating system to workstations with RISC architectures. Interests grew in many areas that could use UNIX for their specific applications [LYNX 92].

On-line transaction processing (OLTP) applications, such as banking and financial uses, require efficient, not instant, recovery from failures. OLTP applications need to supply accurate and immediate up-to-the-minute information about businesses, store this information in a data base, and synchronize the data base with the real business. UNIX investigators have determined that the operating system had both strengths and weaknesses for OLTP applications. Some strengths of the UNIX operating system are its powerful program development tools, integrated terminal support, simple programming model, flexible basic file system, application portability, growing talent pool, and expanding library of application software. The weaknesses of the UNIX operating system, for OLTP, are that it has a high CPU overhead, it has no commercial file system, it is not designed for multiprocessing, it is not fault
tolerant, and it is weak on data recovery, availability and security [Hen86], [Sim89]. Nevertheless, many efforts have been made and are still being made in order to build a UNIX operating system which will be perfect for OLTP.

In 1986 Tolerant Systems Corporation had concentrated on the weaknesses of UNIX while capitalizing on its strengths in order to build a comprehensive OLTP operating system, TX, which has met the specific on-line transaction processing application requirements of high performance, data integrity, expandability, and continuous availability. In order to achieve the previous task, Tolerant systems employs multiple very large scale integration (VLSI) processors which operate in parallel. Each processor has a specific function to carry out. The Eternity System Building Block (SBB) is a three processor OLTP computer in which one processor executes user programs, and the other two perform operating system tasks. UNIX is distributed among the multiple VLSI processors within one of the System Building Blocks. Each SBB is self contained, yet it may access any other resource through an “invisible network” which connects the three SBBs [Hen86].

Not long after Tolerant System’s operating system, TX, many other software development corporations began research on how to make UNIX a better system for OLTP. On a joint project, AT&T and Tandem Computers combined their expertise and created the StarServer FT computer, also known as Tandem Integrity, which offered application-independent fault-tolerance and data protection on a UNIX operating system. This system then supplied many businesses and organizations with increased system productivity, lower operating costs, and expedited operations while promoting efficiency. The use of RISC processors allowed for faster execution of instructions and permitted performance additions such as cache controls and registers on the same chip set. The StarServer supports many high level computer languages such as C and assembly languages, C++, Micro Focus COBOL/2, standard Pascal, and ANSI standard FORTRAN-77. In addition, the StarServer FT is used for database management system products in corporations such as Oracle, Informix, and Ingres relational database management system products, along with Tuxedo and C-ISAM. The fault tolerant system employs three processors which independently execute the same instruction stream and send the results to self-checking voter logic. The voter logic compares the results of the three processors which enables this system to detect a faulty CPU and remove it from service before data corruption can occur. Furthermore, the StarServer FT contains a
feature which ensures data integrity through the use of error checking data items called check sums and through replicating the data in case of faults in the system. System crashes are greatly reduced through the StarServer FT's maintenance and diagnostic system which provides the ability to detect and correct physical faults. The StarServer FT also allows simple standard networking protocols and creates a very simple user environment which meets with great customer satisfaction [DLW90].

In 1991, the School of Applied Science, Nanyang Technological University in Singapore, created an experimental distributed system for highly available databases, called Crystal. The creators solely built this project in order to operate in a transaction processing environment. The project stemmed from the idea that replication of databases increases availability of databases in distributed systems so that each database can be maintained in the same state at the end of any transaction. While one machine acts as a backup database server, application servers, which perform the actual transaction processing activities, distribute the workload of Crystal. Thus, Crystal combines the concepts of Stratus, which requires dedicated hardware to support fault tolerance, and TARGON/32, which requires that all processors be available for productive execution in the absence of failure [LB91].

The telecommunications industry has also witnessed a widely increased usage of the UNIX operating system. AT&T Bell Telephone Laboratories built the duplex multi-environment real-time (DMERT) fault tolerant, process-oriented operating system in order to support both real-time and time-shared operations and to provide a versatile software base for telecommunication systems [HRS84]. Since then, the telecommunications industry has had a growing demand for a fault tolerant UNIX system. Integrated Micro Products (IMP), a specialist supplier of fault tolerant systems, had claimed that a UNIX based fault tolerant system would be cheaper than a single mini processor in the long run due to its fault free operation. However, IMP had problems swaying customers in telecommunications from buying from big name suppliers [Hun90].

Today, Isis Distributed Systems has been dedicated to the development, sales, and support of reliable distributed processing software technology which provides applications with communications guarantees and behavior consistency essential for building reliable systems. The world's top exchanges, banks, and brokerage firms from Wall Street in New York to Tokyo in Japan, use the Isis system. Isis supports many applications for many different customers. The New
York Stock Exchange and American Express use Isis Distributed Systems for their finance applications. Texas Instruments and Siemens employ Isis for VLSI chip fabrication systems for their factory automation. General Electric and SouthWestern Bell utilize Isis telecommunications applications for their distributed systems in conjunction with intelligent switching products and cellular services which require high performance through parallel processing, fault tolerance, and consistency within distributed databases. Finally, prestigious physics laboratories, such as Los Alamos National Laboratory and CERN use Isis for scientific computing, and Hewlett-Packard markets Isis products as part of their Cluster Computing program [Bir93].

The fact that fault tolerant distributed UNIX systems are becoming more and more important in today’s world is easily apparent from the previous examples and others too numerous to list here. The method of stateful task recovery will hopefully enable utilization of a fault tolerant UNIX system in many of the same ways, yet will offer advantages over the previous techniques due to the implementation of the UNIX operating system in a Byzantine resilient architecture.

1.4 Outline

Chapter One has presented a brief overview of typical fault tolerant systems and their applications. Additionally, Chapter One discusses the motivation for this research and gives examples of a few other projects where businesses employ the Unix operating system for fault tolerance.

Chapter Two briefly describes the UNIX operating system by discussing the architecture of the entire system and the architecture of the kernel, by explaining how the kernel manages memory storage, and by defining a UNIX process. Moreover, Chapter Two identifies what constitutes a process in UNIX, and why this research demands a complete understanding of a process in order to fulfill the objective of stateful task recovery.

Chapter Three introduces the existing approaches of stateful task recovery, the names of several systems used for recovery, and the applications for which the systems are used. Furthermore, this chapter explains why the FTPP can not assimilate any of the existing approaches of task recovery. In closing, the chapter explains why stateful task recovery is a much more general,
yet superior, recovery process as compared to the other systems described in the chapter.

Chapter Four continues with the advantages of stateful task recovery over existing systems and further develops the proposed design which will enable recovery. In addition, this chapter presents the initial test platform for recovery, a single 486 processor. Finally, the chapter discusses the two phases in which recovery is achieved.

Chapter Five delves into the specific methods of recovery. To be specific, this chapter presents a detailed analysis of the manipulation of kernel source code which enables process recovery. Chapter Five ends with a description of the specific critical data structures a recovered process needs in order to resume another application process.

Finally, Chapter Six presents the conclusions and recommendations of this research. It explains how certain structures like the kernel stack and the process control block can not be copied into the recovered process and the ramifications of this discovery. Furthermore, Chapter Six discusses the current working state of process recovery. The chapter then ends with some recommendations on a more robust recovery process which can handle open files, process to process communication, and pending signals.
CHAPTER TWO

2.0 The UNIX Operating System

As described in Chapter One and further described in Chapter Three, many methods exist that allow systems to detect and recover from failures. However, no-one has implemented the previous methods on a fault tolerant parallel processor running the UNIX operating system. This research focuses on the design of a very general solution called stateful task recovery, a robust procedure which restarts a failed UNIX task by copying critical data structures from currently running, non-faulty, identical processes. Thus, determining the critical data structures which we must copy in order to guarantee a global consistent state among the four identical processes running on the FTPP is essential to this research. In order to ascertain the critical data structures, we studied the design and architecture of the UNIX Operating System in depth to determine the essential components of a process; in short, what constitutes the context or state of a process.

2.1 UNIX Architecture

As shown in Figure 2.1, the very core of the UNIX operating system model is the hardware which the system kernel surrounds. The makeup of the operating system isolates the kernel from user programs and commands located in a layer above the kernel [Bac86]. The outermost layer consists of application programs such as language compilers like cc, the standard C compiler, utility programs, and user application programs. In order for application programs and commands to function correctly, they must invoke the kernel through special interfaces called system calls. These system calls, which operate in kernel mode, enable the non-kernel programs to pass data into the kernel and receive essential data from the kernel. The kernel interacts directly with the system hardware and can access registers which are essential for any process execution. When operating in a system call, the process is in a privileged execution mode called kernel mode and can receive information, such as register information, that it can not attain when operating in user mode. In short, the kernel is essential for the correct operation of the entire UNIX operating system. In fact, the kernel is the
basic building block of the entire UNIX system; without it, the system could not function.

Figure 2.1  System Architecture

2.2 Kernel Architecture

The UNIX operating system executes in two modes, user mode and kernel mode. User mode is the basic, nonprivileged protection mode of operation where a process, which is simply a program in execution, executes application code. In layman's terms, user mode is the normal operating mode for a user who is running a program. Kernel mode, however, is a privileged execution mode where a process seeks services from the operating system. A system call, hardware interrupt, hardware trap, and software-initiated trap or interrupt are the only initiating events which exist that enable entry into the kernel. This research deals only with system call entries into the kernel.

Two major components of the kernel are the file subsystem and the process control subsystem. Figure 2.2 illustrates the architecture of the UNIX operating system displaying the file subsystem on the left and the process control subsystem on the right side of the figure. The understanding of the file system and the process control system is a key concept in determining how to restart a
failed task. Maurice J. Bach best describes the file subsystem when he states:

The file subsystem manages files, allocating file space, administering free space, controlling access to files, and retrieving data for users. . . The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. Device drivers are the kernel modules that control the operation of peripheral devices. Block I/O devices are random access storage devices; alternatively, their device drivers make them appear to be random access storage devices to the rest of the system [Bac86].
The process control system enables the kernel to communicate between processes, schedule when processes should be executed, manage the memory allocation of all processes, and synchronize processes. The arrow between the process control subsystem and the file subsystem in Figure 2.2 indicates that the two systems communicate when loading a file into memory for execution. Looking at the process control subsystem on the right side of Figure 2.2, the interprocess communication module handles several forms of interprocess communication, such as asynchronous signaling of events and synchronous transmission of messages between processes. The scheduler module allocates the CPU to processes which run in turn until the process “gives up” its CPU time or until the kernel preempts the process when its run time exceeds a kernel-specified time limit. The scheduler determines which process will run by choosing the highest priority eligible process. The memory management module determines if the system has enough memory for all processes and ensures that every process will execute by swapping processes between main memory and secondary memory. Located on the bottom of Figure 2.2 is the hardware control module which handles interrupts and communicates with the machine.

Within the file subsystem, there are kernel data structures which define the internal representation of a file. One of the most important structures is the index node or inode which contains a description of the disk layout of the file data and other information such as the file owner, access permissions, and access times. Every file has an inode. The kernel contains two other data structures, the file table and the user file descriptor table. The file table determines where the

![Diagram](image)

*Figure 2.3  File Descriptor Table, File Table, and Inode Table*
user’s next read or write will start by determining the byte offset, and it also
determines the access rights allowed to the accessing process. The user file
descriptor table contains the information on how may files a process has opened.
The file table is a global structure while the user file descriptor table is allocated
each time a process is created. When using files, the kernel reads the inode table
from the file system into an in-core inode table. The three kernel structures,
shown in Figure 2.3, maintain the state of the file and the user’s access to it.

Most UNIX systems use disks to store regular files and directories. These
disks are usually partitioned into several file systems in order to facilitate the
management of data storage. The kernel treats the file system as a logical device
with a logical device address while the disk is treated as a physical device with a
physical device address. The disk driver makes a conversion between the logical
and physical device addresses. The file system consists of a boot block which
occupies the beginning of a file system and is used to initialize or boot the
operating system. Next, the super block describes information such as the file
size, the number of files the system can store, and the location of free space on
the file system. System administrators specify the inode list size. This structure
is a list of inodes and it follows the super block. The last structures are the data
blocks which contain file data and administrative data. Figure 2.4 shows the file
system layout.

<table>
<thead>
<tr>
<th>boot block</th>
<th>super block</th>
<th>inode list</th>
<th>data blocks</th>
</tr>
</thead>
</table>

Figure 2.4 File System Layout

2.3 Memory Management

The memory-management system handles the organization of memory on
a computer. The following illustration, Figure 2.5, shows the hierarchical layout
of memory beginning with the top level, which is the CPU, and then finishing
with physical storage devices such as disks. The CPU has the fastest cycle time
of any element on a computer. The memory-management system can access
memory stored in the CPU or in the registers very quickly. The cache also has a
very fast process or cycle time which is only a bit slower than the register cycle
time. The third block down in Figure 2.5, main memory, is slower than the CPU.
yet much more fast in its access time than the physical disk devices located below it [LMKQ89], [HB84], [Sch94]. A computer network, as illustrated by the arrows in Figure 2.5, interconnects each component which makes up the memory storage on a computer.

2.3.1 Virtual Memory

Most UNIX machines have an important abstraction called virtual memory which enables the computer system to readily handle processes which require large amounts of memory. In fact, a machine which employs virtual memory, a virtual machine, presents the illusion that the system has much more memory than in actuality. The system creates this illusion because it operates in virtual address space. This space is a range of memory locations which the system references independently of the physical memory addresses. Hence, virtual addressing allows the loading of programs into any address location without the need of changing any position-dependent addresses. Consequently, the programmer and compiler can choose any address for memory storage rather than having the system hardware restrict certain addresses. Furthermore, the operating system does not require the entire virtual address space of an
executing process to be resident in main memory. This empowers the virtual machine with several advantages over a system which does not support virtual memory. First, if a machine does not have enough main memory to run some large processes, virtual memory will enable the machine to run large processes since only part of the program will have to be resident in main memory while the unused sections of the program reside in virtual memory. Second, since unused sections of a program and its data space need not be present in main memory, each process needs less space in main memory which enables more programs to access main memory in order to compete for CPU time. Last of all, programs are able to begin much more quickly since they require only a small amount of main memory before beginning execution [LMKQ89]. The memory hierarchy, as illustrated in Figure 2.5, enables virtual memory to exist. The parts of a program not located in main memory during execution are located on secondary storage devices such as disks, and the system can page in the memory when needed.

A virtual memory system is dependent on the hardware architecture of the specific machine. The memory-management unit (MMU) is part of the CPU hardware which translates the virtual memory addresses into references to physical memory addresses. The memory-management unit is a critical component of the computer system hardware in that it ensures the protection of each process' physical memory and corresponding virtual memory so that one process cannot alter the contents of another process [LMKQ89].

Furthermore, the memory-management system utilizes a concept of memory organization called demand paging. The system stores a process' memory in units called pages which are usually about 4000 bytes each. When process execution needs a part of its memory which is not resident in main memory, the memory management system simply searches for the memory needed, swaps out another segment of memory which it does not need, and swaps in the required memory page(s). Since this research employs computer processors which are virtual memory machines, we must be careful with memory addressing during coding since virtual and physical addressing are completely separate entities. Also, different processes and the kernel use different virtual memory spaces.

A final topic that needs mentioning is the memory management of main memory. The core map is a data structure which handles all main memory management. It is part of the operation system, however, it is not hardware related like the MMU. There exists a one-to-one correspondence between the

35
core map and the page clusters of main memory; therefore, the core map easily locates any page cluster, and the system uses the core map for referencing portions of main memory. The core map enables physical to virtual address translation, indicates available clusters of physical memory, locks and synchronizes on a per-cluster basis, and locates free pages which contain useful data such as pages of a pure text segment. The recovery process created for this research utilizes the core map.

2.4 A Process Defined

A key concept for stateful task recovery is understanding the exact operations of a process. For instance, we must discover what exactly makes up a process. We must determine the inner-workings of a process so that we can copy essential process components to another newly created process. In order to determine a solution, we have studied, in depth, the process subsystem of the UNIX operating system.

![Figure 2.6 Process Memory Mapping](image)
One can think of a process as an execution of a program. It consists of three structures called the text, data, and stack. These three pieces make up the process’ virtual address space which the kernel provides. Figure 2.6 illustrates the architecture of the virtual address space layout and how the operating system maps virtual memory to physical memory [And90], [Sch94].

The interaction between virtual memory and physical memory is an important concept for this research. As illustrated by Figure 2.6, the region table assumes contiguous virtual memory for processes. The operating system maps the user text, data and stack and the kernel text, data, and stack to specific entries in the page tables. Each block of memory specified in the region table maps to one page table entry. The page tables enable the translation from a virtual memory address to a physical memory address as long as physical memory contains the specific page. If the page referenced by virtual memory is on disk, then the MMU must swap that page into physical memory and swap out another page not currently used by that process or another process.

2.4.1 Process Creation and Execution

The fork() system call creates all processes in the UNIX operating system. The only process not created by the fork() system call is Process 0, which the system creates during boot time. The kernel uniquely identifies each process created by fork() with a process ID (PID) number.

The child process created by the fork() system call is an exact replica of the parent process which called fork(). Essentially, fork() duplicates the address space of the process which called it. In addition to duplicating the address space, fork() assembles all essential components for process execution. For example, fork() allocates the kernel and user page tables, the user structure, the proc structure, the per process kernel stack, and the secondary storage used to back the process. Finally, fork() ensures a return to both the child and parent process since at the end of the fork() system call, two processes exist [LMKQ89].

An exec() system call usually follows a fork() system call because the exec() system call enables the kernel to overlay the virtual address space of the child process with the contents of a new executable image found in an executable file [LMKQ89]. This newly created process consists of regions called the text, data, and stack. The text is the program instructions, and the data is the initialized and uninitialized data of a program. The kernel automatically creates
the stack and changes its size as the process continues to run. As there are two modes of operation in UNIX, the kernel creates both a user stack and a kernel stack corresponding to the user mode and kernel mode of operation. A process in the UNIX system executes normally in user mode and in kernel mode upon invocation of system calls. Both stacks have the same structure, but the user stack contains arguments, local variables, and other data for functions executing in user mode, while the kernel stack contains the stack frames for functions executing in kernel mode. Figure 2.7 reveals that frames, which are pushed on

![Figure 2.7 User and Kernel Stack](image)

the stack when calling a function and popped off the stack when returning, are the building blocks of both user and kernel stacks; a stack pointer indicates the current position on the stack [Bac86]. Each of the frames contain the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the program counter and stack pointer at the
time of the function call. Instruction sequences in the program code manage
stack growth while the kernel allocates space for each stack.

2.4.2 Process Data Structures

There are five main data structures which exist for each process created in
the UNIX operating system. As Figure 2.8 shows, there is a u area, a process
table, a per process region table, a region table, and main memory [Bac86]. The u
area contains fields that need to be accessible only to the running process. The
kernel allocates space for the u area only when creating a process. The process
table contains fields that must always be accessible to the kernel. The process

![Data Structures for Processes](image)

Figure 2.8 Data Structures for Processes

table entry and the u area describe the state of a process. A process has several
states which characterize it. The per process region table is an extra level of
indirection which allows independent processes to share regions. Finally, the
region table entries describe the attributes of the region, which is a contiguous
area of a process' address space. The region table entries maintain information
such as whether the region contains text or data, whether it is shared or private,
and where the data of the region is located in memory.

Understanding the functions of a process completely necessitates
understanding what the context of a process is. As described by Bach:
The context of a process is its state, as defined by its text, the values of its global user variables and data structures, the values of machine registers it uses, the values stored in its process table slot and u area, and the contents of its user and kernel stacks. The text of the operating system and its global data structures are shared by all processes but do not constitute part of the context of a process [Bac86].

The system executes in the context of a process, and the kernel can switch each process in order to execute in the context of another process. When the kernel does execute a context switch, it saves enough information so that it can later switch back to the first process.

![Process State Transition Diagram](image)

**Figure 2.9 Process State Transition Diagram**

Figure 2.9 illustrates the different states in which a process can operate, and how the process can make transitions between these different states [Bac86]. The states "preempted" and "ready to run in memory" are the same state; nevertheless, Figure 2.9 separates them in order to illustrate that the kernel can
preempt a process executing in kernel mode only when it is about to return to user mode.

For an example of a process going through some of the state transitions, suppose the system swaps out a process to make room for another process that is in the state “ready to run, swapped”. The kernel (scheduler) chooses which process to place in main memory and switches the process from “ready to run, swapped” to “ready to run in memory”. Eventually the scheduler will run the process by switching it to “kernel running”. This is a typical example of how a process transition takes place.

Figure 2.10 shows the context of a process which consists of the contents of its user address space and the contents of hardware registers and kernel data structures that relate to the process. There is a user-level context, a register context, and a system-level context as shown in Figure 2.10. In order to complete a context switch, the kernel pushes the context layer of the old process and pops the context layer of the new process. In short, the kernel must always save the context of a process when it swaps the process out of memory in order to bring that process back at a later time.

![Diagram](image-url)

Figure 2.10 Components of the Context of a Process
The previous examples and explanations represent a quick overview of how the UNIX operating system performs. UNIX is a very complex system, and the following chapters will explain more about processes and memory management.
CHAPTER THREE

3.0 Existing Approaches to Process/Task Recovery

Many approaches exist in which stateful task recovery can be achieved. The primary motivation for each method has stemmed from the fact that the restarted task must be "stateful" or have the same state consistency after recovery. Two different models for fault-tolerant computing in distributed systems include transaction based recovery and checkpoint-rollback based recovery.

3.1 Transaction-Based Recovery

In transaction-based recovery, units of work called transactions divide up the processing. The transactions are simply any transaction that may take place on computer systems in banks or in stock markets. Transaction boundaries always define consistent system states from which a computation can recover. Transactions encapsulate services for checkpointing, recovery, and replication. A transaction is atomic which means that it is both serializable and recoverable. Serializable means that the overall effect of executing multiple concurrent transactions is as if they had been executed in some sequential order. Recoverable means that the execution of a transaction must be all or nothing; data is either consistently changed by the action or no changes occur [Bab90], [LDHJLSW87]. The transaction protocols of either committing or aborting updates to stable storage can be built into an operating system and can be made transparent to applications. However, not all systems can employ transaction recovery, since it is based on serializability. Furthermore, transaction-based recovery assumes a low probability that different transactions contend for the same data, and it assumes that transactions involve enough computation to amortize the I/O delays of synchronous commits at each transaction boundary. Additionally, transaction based recovery is overly restrictive for general distributed computations because its designers specifically created it for database applications.
3.1.1 Tandem “NonStop”

A widely known transaction processing computer system is the Tandem 16 computer system. The creators built this system to be a “nonstop” operating system; an incredibly reliable system for on-line applications. The architecture consists of processors which each have their own power supply, memory, and I/O channels. Processors are able to communicate with each other over redundant interprocessor buses. The entire system provides error detection and correction through its communication paths and memory accordingly.

The designers had several design goals for the Tandem operating system. The goals consist of the following:

- The system should be tolerant of any single module or bus fault.
- Any failed module or bus must be able to be restored and reinserted into the system while other errorless processes continue to run.
- Hardware and software must work synergistically so that one does not corrupt the other.
- The operating system must be very general so that it is easily portable to any size hardware configuration and/or system configuration.

To meet these design goals in a system called Guardian, Tandem employs two low-level abstractions, messages and processes. The sending of messages enables process to process interaction. Messages are only implemented when processes on different processors must communicate. Processes on the same processor communicate through memory. The only time a failure could occur is if the sender or receiver process fails or the entire processor fails. Thus, even though multiple computers make up the system, it acts as one system which contains many processes that communicate through messages.

Tandem “NonStop” also uses process pairs for reliability. For example, two processes on separate processors would handle an I/O device such as a printer. One of the processes is the primary process while the other is a backup process which the system will implement only if the primary process fails or an I/O channel error occurs. When the system opens or closes a file, the primary process sends information to the backup process through the message system [SS82].
3.1.2 Argus

Another system which utilizes the transaction model is Argus, a system built in order to support the construction and execution of distributed programs. Argus implements procedural abstraction, control abstraction, and data abstraction. It provides a module, called a guardian, which encapsulates and controls access to one or more resources. A guardian is a kind of data abstraction which resides at a single physical location on a processor called a node. If a node fails, the guardian survives and recovers the crash by reconstructing its volatile objects from its stable objects. From this, it is recognized that the persistent state of an application should be kept in stable objects. This method of transaction based fault tolerance is not transparent to applications since programs must announce the beginning and end of transactions within programs at opportune points [LDHJLSW87].

3.1.3 Transaction Processing Kernel (TPK)

The Department of Computer Science at Princeton University designed another type of transaction-based fault tolerant processing system. The department implemented the prototype, called the transaction processing kernel (TPK), on DEC Firefly workstations. The goal was to evaluate the systems performance through the combination of multiple processors and massive memories. The department determined that the system could process simple transactions at a high rate on stock multiprocessors with sufficient memory [LN88].

3.1.4 Transaction-Based Fault Tolerance Implemented on MachTP

In 1992 a more advanced transaction paradigm was implemented on the Camelot distributed transaction facility running on the Mach distributed system [NF92]. This system, the MachTP, combines process-pairs and transactions to obtain fault tolerance. Process-pairs ensure that each server process that provides services to applications has a backup process that will assume responsibility in the event of a failure. The transaction module, which is atomic, guarantees a consistent state after a recovery from failure. MachTP implemented another concept of replicated transactions which effectively could leave the
system in a consistent state that could continue to provide services to future client requests even after a lost transaction.

3.2 Checkpoint-Rollback Based Recovery

The other class of failure recovery, checkpoint-rollback, includes two disparate strategies to guarantee global state consistency: optimistic and pessimistic recovery. Both methods are more application independent than transaction-based recovery. Instead of fault tolerant computing through the use of transactions, checkpoint-rollback enables a system to restart a failed task from a saved past state. In Ozalp Babaoglu’s article on Fault-Tolerant Computing Based on Mach, he states:

All that is required is a mechanism whereby computations can be restarted from some past state in response to failures. To prevent having to restart computations always from the very beginning and thus guarantee forward progress, the state of the failure-free execution is periodically saved to stable storage. The saved past states are called checkpoints. The act of restoring a computation to a past state is called rolling back and the interval during which recovery is taking place rolling forward [Bab90].

Because rollback could possibly result in an inconsistent state, the two unique strategies of optimistic recovery and pessimistic recovery guarantee that the restart will be a stateful procedure.

3.2.1 Optimistic Recovery

Optimistic recovery is designed for failure-free execution of a system by gambling that failures will not occur; that is, it guesses that the set of state intervals named in the dependency vector of an input message will be made recoverable before the next failure [SY85]. Strom and Yemini designed a system which requires no synchronization upon computation, communication, checkpointing, and committing to proceed. However, they assume that the computations are deterministic, meaning that an initial state and a sequence of messages to be received uniquely establish the final computation state and the sequence of messages it sends [SY85]. Koo and Toueg considered the problem of nondeterministic computations or “processes” in order to construct more
accurate algorithms. Their application requires the storage of two checkpoints per computation in order to allow the toleration of failures during checkpointing and rollback [KT87], [Bab90].

3.2.2 Pessimistic Recovery

In the rare case of failure, a method which synchronizes communication and computation with checkpointing is pessimistic recovery. With this methodology, the recovery of a failed process only involves the computations affected by the failure. To avoid any inconsistencies in case of failure, a pessimistic system delays each message until it checkpoints both the state of the sender and the state of the receiver [SY85], [Bab90]. In order to avoid the substantial delays associated with checkpointing to a stable storage location, pessimistic recovery systems employ a backup process, which resides on another processor, to hold checkpoints. TARGON/32 is a notable system which utilizes pessimistic recovery. This system uses a process-pair scheme, which means that every primary process has a backup process which will take over execution if the primary process fails [BBOG90]. Atomic three-way message transmission supports the recovery of actions between checkpoints. Three-way message transmission means that every message which is sent from one backed up primary process to another is actually sent to the target process, the backup of

![Diagram](image)

Figure 3.1 Atomic Three-way Message Transmission
the target process, and the backup of the sending process. Three-way atomic
broadcasting, shown in Figure 3.1, ensures all messages are sent to both the
primary process and its backup. In addition, three-way atomic broadcasting also
ensures that its backup process counts all messages sent by the primary process
as writes-since-sync. Writes-since-sync means that a primary process and its
backup are periodically synchronized to avoid recomputation upon failure and
to achieve recovery from a more recent point. Synchronization occurs either after
the primary process reads a defined number of messages or when a defined
amount of time has elapsed since the last synchronization. Critical operating
system functions have been moved out of the kernel into recoverable server
processes allowing the system to recover from one or more than one hardware
failure and many nondeterministic software problems [BBGHO89]. The
processors in this system must be available for execution and never exist solely as
backups. TARGON/32 embodies complete transparency for recovery while
concentrating on efficiency of recovery, not on immediate recovery, since the
creators of the system designed it for a transaction processing environment. In
order for the pessimistic-based recovery scheme to work correctly, a crashed
process' state must be available, all of the messages that would have been
available to the primary in that state or since that state was reached must be
available in the correct order, and the process must behave deterministically.

The architecture of TARGON/32 consists of a local area network (LAN) of
2 to 16 machines connected via a fast dual bus. The machines consist of three
shared-memory processors; one processor handles incoming and outgoing
messages and the creation, maintenance, and recovery of backup processes while
the other two processors execute UNIX-style processes and much of the system-
call-related kernel code [BBGHO89].

Optimistic recovery has several advantages over pessimistic recovery. Since
optimistic recovery requires no synchronization upon communication,
logging delays generally do not slow down computation. Furthermore,
optimistic recovery can salvage some failures not immediately detected by the
system, it can checkpoint without requiring a backup processor, and it can
recover tasks even after the temporary loss of all processors. However,
optimistic recovery recovers somewhat more slowly when failures occur, and it
is not as fault-proof as pessimistic recovery; nevertheless, if failures in a system
are infrequent or never occur, optimistic recovery can perform significantly
better than other recovery techniques [SY85].
4.0 Advantages of Stateful Task Recovery

This research proposes a method of stateful task recovery for a more general solution to the problem of recovering from software failures. For instance, the method of pessimistic recovery employed in TARGON/32 is a very constraining method. This method uses atomic three-way message transmission which demands considerable extra software. This may be fine for on-line transaction processing applications but not for other applications, such as real time applications, for which people use fault-tolerant UNIX systems. Additionally, the backup processes in TARGON/32 are doing as much work as the primary processes. The idea in pessimistic recovery is to intercept all shared memory writes and place them into backup; however, this breaks down when there are other types of communications such as signals and semaphores. In addition to the weaknesses previously mentioned, pessimistic recovery involves ample computation.

The core of the TARGON/32 design is the fail-stop model, which means that the system detects all failures immediately, and then the system initiates recovery action. Nevertheless, if the data is somehow corrupted without failing the task, the process will continue running, and the fault will propagate, through messages, to the backup. On the other hand, the method of stateful task recovery will not allow a single process to corrupt the system because the faulty process will get voted out. In short, the FTPP will work irrespective of structures of tasks. The goal is to take a logical fault-free process whose state we can copy and start another logical fault-free process in synchrony with the other three processes running on the FTPP.

4.1 Proposed Design for Stateful Task Recovery

When one of the processes on the FTPP fails, the system must be capable of determining that a process has failed and exactly what process on which processor has failed. The Fault Detection, Isolation, and Recovery (FDIR) program will enable the system to determine what process has failed or is faulty, and where the fault has occurred. This relates to the first three words of FDIR,
Fault Detection, Isolation. This research deals specifically with the recovery part of FDIR. Thus, we will use recovery and stateful task recovery interchangeably throughout the next chapters.

An FDIR program resides on each processing element of the FTPP. When processes vote their data over the network elements, the message sent back through the device driver has a message syndrome which has information about whether or not the process is faulty. The FDIR program then looks at the syndrome message. Figure 4.1 illustrates this procedure. If a fault exists, FDIR

![Diagram](image)

Figure 4.1 FDIR Message Transmission and Receiving

...can send a message to all processors informing them of the faulty process.

The following nine figures illustrate the implementation of processes on the FTPP. Figure 4.2 shows the normal allocation of processes on the FTPP, and the figure also shows how FDIR resides on all four processors. FDIR can detect not only software faults or process failures, but also any hardware failures.
Therefore, if a fault were to occur, FDIR will detect and isolate the fault. Figure 4.3 illustrates FDIR's fault detection and isolation. Afterwards, FDIR must halt the correct application processes on the other three processors at a natural synchronization point where all processes must be in the same state as shown in Figure 4.4. Following this, FDIR must create a new application process in order
to replace the failed process; this is the beginning of the recovery phase of FDIR. In order to complete this task, FDIR must fork a new process as depicted in Figure 4.5. The new application process, however, is an exact replica of the FDIR process which created it; therefore, FDIR must invoke the exec() system call in order to replace the address space of the newly forked process with the desired address space of the application process which failed. Figure 4.6 represents this "exec() like" system call, my_exec(), an altered version of the exec() system call which enables process recovery. After invoking my_exec(), recovery must save
the state of the correct application processes in files in order to copy this saved state into the newly created application process on processor 1. FDIR needs to copy the state because the exec() system call enables the application process to start from the beginning of the application program's execution; however, the saved state will enable the advancement of the process to the same instruction at which FDIR halted the other three application processes. In order to achieve this progression to the desired instruction, stateful task recovery saves the states of the three correct application processes in files, the FTPP votes the files over the fault tolerant network, and finally it writes the voted saved state into processor 1.
as illustrated in Figures 4.7 and 4.8. We will ensure that the processes are at the same state during voting. Finally, recovery writes the saved state file on processor 1 into the newly created application process, Figure 4.9, and the recovery process simultaneously starts the four identical processes, Figure 4.10.
The network interface allows the FDIR programs, resident on each processor, to act as one because they can communicate over the fault tolerant network.

4.2 Initial System Configuration

The initial system utilized for this research is a 66MHz i486 processor with 20Mbyte RAM and 500Mbyte disk. The operating system is BSD/386 (Berkeley Software Distributions). This system provides a much simpler configuration than the FTPP for development; nevertheless, the methods used will allow us to host the process recovery configuration within the FTPP. Furthermore, the generality of process recovery will enable any user to port it to other systems with ease.

We have broken the task of process recovery into two phases, Phase 1 and Phase 2. Phase 1 entails determining what the critical data structures of a process are, finding and copying those data structures, and then saving them to a file. Phase 2 entails creating a new application process, initializing that process, refreshing the process state with the previously saved state, and finally executing the recovered process.
4.3 Process Setup on 486

Two distinct processes are running on the 486, an application process, representing the correct version of a process, and the recovery process (FDIR) which will read the state of the application process, write that state to a disk file, create a new application process, and write the saved state from the original application process into the newly created application process. The following figures show a relation between process recovery on the FTPP, the picture on the left side of each figure, and process recovery on the 486 processor, the picture on the right side of each figure. In actuality, there is only one application process on the 486 processor; nevertheless, for this setup, we assumed that an application process has faulted, Figures 4.11 and 4.12. Thus, Figure 4.13 is where work began.

Figure 4.11 Normal Application Processes

Figure 4.12 Fault in Application Process #2
on the 486. Figures 4.14 and 4.15 illustrate the creation and initialization of the new application process. Figure 4.16 represents the creation of the saved state file. This simpler model does not do any sort of voting which the FTPP must do as shown in Figure 4.17. Figure 4.18 shows FDIR writing the saved state into the
new application process, and finally, Figure 4.19 represents the two processes as the recovery process simultaneously starts them.
At the time recovery begins, the application process will have suspended itself transparently via a sleep() system call. Additionally, in order to test the recovery process, the application process computes a reasonable function iteratively, thus making checks possible and very easy. To be succinct, the recovered process should continue the series computation where the previous instantiation terminated.

We imposed the following restrictions on the system in order to facilitate the initial design:

- Nothing would be written out to files, thus, eliminating the additional burden of saving and restoring file and inode tables, of maintaining a buffer cache, of capturing processor cache state, and of refreshing disk blocks.

- The application process should not attempt communication with another process such as through shared memory or via signals.

We proposed these restrictions only for this research on stateful task recovery; future work can remove these restrictions.

### 4.3.1 The Defining Attributes of a Process

Each process has an area of memory reserved for process-specific information used by the kernel when the process enters kernel mode. This information includes a small stack for use by the kernel while executing system calls for the process. Below this stack, in terms of addresses, is the user area which, in actuality, encompasses the per-process kernel stack. The user area resides in user address space and is swappable; however, the operating system
configures the user area such that when a process is executing, the user area is directly addressable at a specific virtual address within the kernel. Figure 4.20 illustrates the location of the user area [BJ94].

![Figure 4.20 User Area Addressable in Kernel](image)

The user area contains long-term information about the state of the process. It contains a per-process kernel stack, as shown in Figures 4.20 and 4.21, it includes accounting information about the process, resource controls, the descriptor table of the process, the state related to system calls, and the user and kernel mode execution states [LMKQ89]. The process control block, or PCB, is the structure which saves the user and kernel mode execution states.

In addition to the information maintained in the user area, a process also requires the use of some global system resources. The kernel maintains a process table that contains an entry for each process created in the system. Among other things, the process table records information on scheduling and on virtual-memory allocation. Because the entire process address space, including the user area, may be swapped out of main memory, the process table must record enough information to be able to locate the user area and to bring the latter back into memory. The process table maintains the scheduling information, rather
than the user area, in order to avoid swapping in the process only to decide that it is not at a high-enough priority to be run.

The location of the kernel stack in the user area simplifies context switching by localizing all a process' kernel-mode state in a single structure. This design, however, restricts the stack to a fixed size and requires programmers to be careful when writing code that executes in the kernel. That is, kernel code must avoid using large local variables and deeply nested subroutine calls to avoid overflowing the kernel stack.

As stated previously, the process control block (PCB) encapsulates the current execution state of a process. Each specific machine defines its own PCB. In general, however, it includes the general-purpose registers, stack pointers, program counter, processor status longword, and segment base and length registers [LMKQ89].

After studying the structures of a process, we determined that the four structures which completely define a process' state are its text, data, user stack, and user area. Figure 4.21, similar to Figure 4.20, shows the critical data structures of a UNIX process [LMKQ89]. In addition, it exhibits how the user area contains substructures which are extremely important to all processes.

![Critical Data Structures](image)

**Figure 4.21**  Critical Data Structures
4.3.2 Phase 1 of Stateful Task Recovery

After determining the critical data structures which needed copying, we then scrutinized the UNIX kernel organization and operation. For instance, since UNIX operates in two modes, user and kernel modes, we had to be extremely careful about copying structures from one mode to another. User processes run in user mode and can access the kernel through system calls. Kernel processes execute code which the system compiles into the kernel's load image, and kernel processes operate with the kernel's privileged execution mode. In order to copy the critical data structures needed for process recovery, we created system calls which enabled entrance into the privileged kernel execution mode.

Since we are dealing with several disjoint virtual memory spaces such as the target process' virtual memory, the forked() process' virtual memory, and execed() process' virtual memory, the system calls had to carefully access each virtual memory space by appropriate mechanisms and at appropriate times. For example, the virtual memory space of the forked() process exists only until my_exec() is called which destroys the forked() process' address space.

A system activity or entry into the kernel can occur through four initiating events; system calls, hardware interrupts, hardware traps, and software-initiated traps or interrupts. The kernel can then be thought of as two separate parts; the top executes in a privileged execution mode in which it has access both to kernel data structures and to the context of the user-level process that it supports. The top half of the kernel handles system calls and traps both of which are unexpected yet are related to the current executing process [LMKQ89].

The bottom part of the kernel contains routines which handle hardware interrupts and traps. Both of these system activities are asynchronous and rarely relate to the context of the process that is currently running. An example of an interrupt would be I/O devices which need attention. Activities occurring in the bottom half of the kernel, which runs on a system wide interrupt stack, should never alter the run-time stack used by the top half of the kernel.

When handling a system call, nothing can preempt the kernel. If there is an interrupt, the bottom half of the kernel handles the interrupt, and there is a careful sharing of the system resources between the top and bottom of the kernel so as to maintain system consistency [LMKQ89]. Figure 4.22 illustrates the two halves of the kernel.
In this research, the system call, read_state(), allows the recovery process to enter into kernel mode from user mode. When this entry occurs, the operating system saves the machine state so that the kernel can not alter values in the currently executing program. A general procedure for a system call is that the hardware switches from user to kernel mode so that the system now operates with the kernel’s privileged instructions. When this switch occurs, the system saves the user stack pointer and then uses the kernel stack pointer. The system then places the program counter and the processor status longword onto the per-process kernel stack as well as the system call number which enabled the system entry into the kernel. Finally, an assembly-language routine saves the general-purpose registers. By saving this state, the kernel can act like any normal C routine without destroying other process states.

When finished with the system call, an assembly-language routine then restores the register state and reverses the process of entering the kernel as described above. Execution resumes at the next instruction, as defined by the saved program counter, in the user’s space [LMKQ89]. Understanding the switching and saving which occurs during the system call read_state() gives clues in working with the kernel source code.

An important structure called the proc structure contains many pointers which allow access to critical process data. First of all, we employ the function
pfnd() to find the target process from which we want to copy the critical data structures. After finding the target process, we create a pointer which points to that process' proc structure. This was the critical observation which allowed us to copy any and all critical structures needed for stateful task recovery. Therefore, we call read_state() in order to find the target process and copy the critical structures, and then we pass the copied structures back into test.c and save them into files. Test.c is a program on the 486 processor which acts like the recovery part of FDIP on the FTPP. Figure 4.23 illustrates the process state and how the proc structure is the key to finding all other critical data structures of a process [LMKQ89].

![Figure 4.23 Process State](image)

### 4.3.3 Phase 2 of Stateful Task Recovery

As explained previously, the recovery process uses the system call fork() to create the new application process. By doing this, fork() builds the process table entry, assigns the process ID, and creates images of the parent's user area, text, data, and stack segments. Furthermore, the kernel stack enables the child process to restore the child context, and both child and recovery process return from the fork().

The child process, which is the new application process, will then make a system call, write_state(), which mimics the UNIX exec() system call yet also takes the saved state information and writes it into the new application process.
An exec() system call replaces the address space of a process with the contents of a new program. To be more specific, the "exec-like" call, write_state(), copies the executable load module of the original application program from disk into the text and data segment memory space of the child process. It initially acquires the inode associated with the application executable file, ensures that the file is executable, and reads its header information. As my_exec() reads data from the file, it allocates map entries as well as copying the data into memory pages. Then, instead of initializing register context with the program counter of the beginning of the application program, my_exec() copies the saved register context from the copied application program. We can only restore the saved state information within the context of the write_state() system call. The following figure, Figure 4.24, represents the architecture of programs which constitute stateful task recovery.

![Diagram of Correct Process and Newly Created Process]

**Figure 4.24** Recovery Architecture Using Kernel
CHAPTER FIVE

5.0 Detailed Design of Stateful Task Recovery

The previous chapter provides a broad overview of stateful task recovery. While reading the following sections, note that the terms "correct application process" and "halted application process" refer to the same process from which recovery copies the critical state structures. The terms "newly created application process" and "recovered process" refer to the process which my_exec() creates.

For this research, we created two C programs, test.c and draper_code.c, and an "h" file, draper_code.h, in order to achieve stateful task recovery. Test.c is the main program which invokes all other system calls in order to obtain the critical state structures from a correct application process and in order to write these structures into a newly created application process. Draper_code.c is a program where the system calls read_state(), write_state(), and my_exec() reside. Read_state() and write_state() are system calls we created for stateful task recovery. On the other hand, my_exec() is an original 4.3BSD UNIX operating system call, exec(), modified for the purpose of recovery. Draper_code.h contains variables for specific cases used in test.c and draper_code.c. It also contains two structures, my_args and args, which define common variables used by the program test.c and the kernel system calls read_state, write_state, and my_exec. The following paragraph explains the two parts of test.c.

5.1 Test.c / Code Which Invokes All System Calls for Recovery

The code, test.c, located in Appendix B, represents the job that the recovery part of FDIR will perform on the AFTA (Fault Tolerant Parallel Processor). Corresponding to the two phases of stateful task recovery as explained in the previous chapter, we divided test.c into two stages of execution. The first part of the program emulates an FDIR process on a processor where a correct, halted application process resides. This first segment of test.c enters the kernel by calling read_state() in order to determine the size of the critical state structures and copy the state structures. Read_state() locates the halted process' proc structure.
The proc.h file defines the proc structure. It is this structure which allowed us to find all the critical state structures of the halted application process. The proc structure must always reside in main memory, and it contains information about memory management, scheduling, process identification, signals pending, process CPU time, and many more important process attributes. The proc structure is the road map to recovery because it references substructures which contain descriptions of all important characteristics of a process. From this structure, read_state() can acquire any and all information about a process. For example, the proc structure assigns variables which point to the location of both the size of the data and the location of the data in virtual memory. The " .h " file " vm.h " defines a process' virtual address space in the structure " vm-space ". The proc structure makes reference to " vm-space " by defining a pointer " *p_vm-space " which points to the " vm-space " structure.

After proper coding, read_state() utilizes " *p_vm-space " to locate the data.

After read_state() locates the data, it copies it from the virtual address space of the correct application process, to the virtual address space of test.c utilizing the kernel function vm_map_copy(). In order to enable this copy, test.c uses the system function " malloc() " which allocates an area of virtual memory. A pointer named data_ptr, which is defined in the structure my_args located in draper_code.h, is set to point at the address location of the allocated memory space. Read_state() writes the copied data into the allocated memory space in test.c beginning at the address which data_ptr points to. Read_state() then returns to test.c which in turn saves the data into a file called " data.file ". This was an important accomplishment since we are using files to emulate message passing, which is the only way to pass information to each processor in the FTPP.

![Diagram](image)

Figure 5.1 Phase 1 of Test.c: Read_state()
Test.c and read_state() perform the same procedures for copying the text, user stack, and all registers. Appendix B contains the code for test.c minus any superfluous printf() statements. Figure 5.1 pictorially represents the copy operation completed by the first part of test.c and by read_state(). A.OUT depicts the halted application process.

Even though proc.h declares pointers for referencing vital parts of a process, it does not clearly define where the user stack and kernel stack begin and end. For instance, on different computers or different hardware architectures, the user stack may grow up in memory or down in memory. The kernel stack then begins from the end or the beginning of the user stack and grows in the opposite direction, as described in several UNIX system books referenced for this research [Bac86], [LMKQ89], [Sch94], [Ste92]. The information supplied by UNIX operating system books, however, was very circumlocutory as to what the authors implied by saying that the user stack “grows up” or “grows down” in memory. In order to precisely determine the direction of user stack and kernel stack growth, we scrutinized kernel code for the virtual memory system. In the kernel source code, the directory “kern/i386/include” contains a file called “vmlayout.h” which explains the 4.3BSD virtual memory layout. In our system, the user stack always starts at virtual address 0xefbfe000 and grows to a lower value address. The user text starts at virtual address 0x00000000 and

![Virtual Address Space Arrangement](image-url)
ends where the user data begins at address 0x00007000. Above the user stack is a space which is used for double-mapping the user structure, which is 2 pages long, 8192 bytes, and contains the kernel stack. Above the kernel stack is the page-table map and the kernel address space.

From the information supplied by the “vmlayout.h” file, we were successful in programming read_state() to locate the user stack, kernel stack, and process control block in virtual memory and copy them to test.c’s virtual memory space. Figure 5.2 depicts the virtual address space arrangement of a process.

The final task of Phase 1 was locating and copying the saved registers. The proc structure defines a variable called “*p_regs” as the location of the saved registers during a system call or trap. Since the application process is halted by a sleep() system call, the registers are located in the p_regs structure. Through searching the kernel source code, we discovered that the kernel locates the registers by defining a register integer that points to the p_regs structure. For instance, the code is:

- register int *regs = p->p_regs;

Here, the variable “p” is a pointer to the proc structure, and again one can see how the proc structure enables referencing of process attributes. After this, the kernel simply accesses the registers by the command:

- regs[register name];

The file “reg.h” contains a list of all of the registers and their offsets when referenced during a trap or system call.

The second half of the test.c algorithm emulates FDIR on a processor where an application process has crashed, and FDIR must create a new application process. We encountered two initial difficulties in the second half of recovery, namely determining where to place the fork() system call and where to invoke the newly created my_exec() system call. At first, we invoked fork() in write_state; however, the operating system would not allow the calling of the fork() system call inside the kernel since fork() was specifically designed for invocation in user mode. Thus, we moved fork() into test.c. Secondly, we placed my_exec() in write_state() so that it could use the kernel’s privileged mode to
complete its task of overwriting the image of one program over another. Section 5.2.1 will further explain the purpose of my_exec().

Just as the first part of test.c copies the critical state structures from one process into its own virtual memory space and finally into files, the opposite procedure occurs in order to write the structures into the newly created application code. Figure 5.3 illustrates the task of both parts of test.c. This figure is identical to Figure 4.24 and is shown below for convenience. As Figure 5.3 illustrates, the second part of test.c opens the saved files and then reads them

**Figure 5.3** Recovery Architecture Using Kernel

into its virtual memory space so that write_state() can gain access to the information contained in the saved critical state structures. Test.c again allocates memory by calling malloc(); however, it uses variables from the second structure in draper_code.h, my_other_uap, to point to the allocated memory. In short, the first structure defined in draper_code.h, my_uap, holds the variables which the first phase of recovery uses, and the second structure, my_other_uap, holds the variables the second phase of recovery uses. The following algorithm illustrates the steps of test.c.

### 5.1.1 FDIR/ test.c Algorithm

*On processor with non-faulty application processes,*
Set up needed variables.

Invoke system call \texttt{read\_state}:
  Obtain critical structure sizes.
Exit \texttt{read\_state}.

Allocate memory space for text pages.
Invoke system call \texttt{read\_state}:
  Copy text pages from application process.
  Save text pages into allocated memory area in test.c.
Exit \texttt{read\_state}.
Create a file called "text.file" and write copied text into file.
  \textit{(would vote on FTPP . . .)}

Allocate memory space for data pages.
Invoke system call \texttt{read\_state}:
  Copy data pages from application process.
  Save data pages into allocated memory area in test.c.
Exit \texttt{read\_state}.
Create a file called "data.file" and write copied data into file.
  \textit{(would vote on FTPP . . .)}

Allocate memory space for user stack.
Invoke system call \texttt{read\_state}:
  Copy user stack pages from application process.
  Save user stack pages into allocated memory area in test.c.
Exit \texttt{read\_state}.
Create a file called "stack.file" and write copied user stack into file.
  \textit{(would vote on FTPP . . .)}

Allocate memory space for kernel stack.
Invoke system call \texttt{read\_state}:
  Copy kernel stack pages from application process.
  Save kernel stack pages into allocated memory area in test.c.
Exit \texttt{read\_state}. 

72
Create a file called “kstack.file” and write copied kernel stack into file.

(\textit{would vote on FTPP . . .})

Allocate memory space for process control block.
Invoke system call \texttt{read\_state}:
  
  Copy PCB from application process.
  
  Save PCB into allocated memory area in test.c.

Exit \texttt{read\_state}.
Create a file called “pcb.file” and write copied PCB into file.

(\textit{would vote on FTPP . . .})

Allocate memory space for registers.
Invoke system call \texttt{read\_state}:
  
  Copy registers from application process.
  
  Save registers into allocated memory area in test.c.

Exit \texttt{read\_state}.
Create files for all registers and write copied registers into files.

(\textit{would vote on FTPP . . .})

\textbf{On processor with newly created application process,}

Open created files:
  
  “text.file”
  
  “data.file”
  
  “stack.file”
  
  “kstack.file”
  
  “pcb.file”
  
  “register files”

Allocate memory space using respective structure sizes.
Read opened files into respective memory spaces.
Invoke system call \texttt{fork}.

\{

  Set up entry into \texttt{my\_exec}.

  Invoke system call \texttt{write\_state}.

\}
{ 
    Invoke system call my_exec().

    Exit my_exec().
}
Exit write_state().

End of test.c.

5.2 Newly Created System Calls, read_state() and write_state()

The first step in creating new system calls was discovering how to build a system call into the UNIX kernel, a name given by computer programmers as rebuilding the kernel. A file called “syscalls.master” exists in the directory “/usr/src/sys/kern”. The file “syscalls.master” contains all of the system calls for the 4.3BSD UNIX operating system. Thus, we placed the read_state() and write_state() system calls inside of this file with an appropriate system call number, the identifying mark for a system call. For instance, when the operating system makes a system call, it takes the system call name and matches it with its identifying number. When the operating system enters the kernel due to a system call, it places the system call number on the per process kernel stack as described previously in Chapter Four, section 4.3.2.

After placing read_state() and write_state() in the “syscalls.master” file, we compiled and linked the files through initiating a special “Makefile” located in the same directory. After this, we altered the kernel “Makefile”, found in “/usr/src/sys/kern/LOCAL”, so that the operating system would link read_state() and write_state() into the operating system code. Consequently, each time we altered the code in either system call, we had to reboot the entire system so that the kernel compiled and linked the two new system calls into the system kernel source code. Thus, altering code was a tedious and slow process.

After we set up read_state() and write_state() in the system, we had to create the actual code so that the two system calls could function just like any other operating system call such as fork() or exec(). We placed the two system calls in a file called draper_code.c located in Appendix C.
Finding the proper way to initiate or invoke the system call read_state() was perplexing. Finally, we determined that the call needed a system call number and a pointer to the structure my_args defined in draper_code.h, found in Appendix D. The method for calling the newly created system call read_state() was different from any other original 4.3BSD operating system call. Original system calls do not take arguments, such as fork(), yet the newly created system calls need to pass arguments for successful execution because they are not automatically set up inside the kernel. As shown in Appendix C, the system call read_state() requires a pointer of type “struct proc” which points to the currently running process’ proc structure, a handle which points to the my_args structure defined in draper_code.h, and a return value (integer). The proc structure pointer, of course, enables read_state() to find and copy the halted process’ vital structures as described in section 5.1. The handle, which is what the C programming language calls a pointer to a pointer, enables the kernel the ability to pass data between kernel and user modes. The return value simply returns a zero for an errorless read_state() execution or a one if an error occurs anywhere in the read_state() system call.

Once inside read_state(), the system call must locate the correct application process. In order to do this, we utilize the system function “pfnd()” to acquire the application process through searching for its known process ID number (PID). By doing this, we obtain a pointer to the halted application process’ proc structure. From this structure, read_state() can then locate and copy the critical structures of the application process such as its text, data, user stack, kernel stack, process control block, and registers. Read_state() incorporates both vm_map_copy() and bcopy() system functions for copying the structures from kernel virtual memory to user virtual memory.

The structure of the virtual memory map system is actually bi-level. The top level of the map structure refers to regions of sharing maps which then refers to the second level of the map structure, the actual virtual memory objects. Thus, vm_map_copy() uses virtual memory maps to copy virtual memory object references from one virtual map to another. A virtual memory map provides mapping, protection, and sharing of virtual memory objects. Additionally, the virtual memory maps allow efficient and speedy virtual copies of one memory map to another. Bcopy(), on the other hand, simply copies data within a single virtual memory space with no concern about virtual memory maps.
In order to verify that `vm_map_copy()` copied the correct structures, we tested the code in three ways. The first way was having `read_state()` simply print out the halted application process' text or data or user stack while in the system call. A simple "for loop" was constructed in order to complete this task. Then, we could look at the file which test.c creates and compare it with the printout from `read_state()`. The file and printouts matched. Secondly, we defined buffers in the application process and filled them with strings such as "hello123456789" or "iminbuffer987654321" so that when we looked at the files saved by test.c, we could easily see that these strings were located in the data file or stack file. Thirdly, we coded return values for all `vm_map_copy()` statements and all return values printed a zero, implying no error. In fact, if an error occurred, the return values would actually return numbers from an error file called "errno.h." From this file, we could then look-up the error numbers and find what caused the errors.

In order to verify `bcopy()`, we simply matched a printout of the registers from inside the `read_state()` system call with a listing of the register files created by test.c. Again, the files matched the printouts.

After test.c copies the critical state structures into files, the second part of test.c completes Phase 2 of stateful task recovery. The second part of test.c invokes the `fork()` system call in order to create a new application process. This new application process which `fork()` creates is a replica of the process which called the `fork()` system call. In other words, the new application process is a replica of test.c. However, the new application process needs to be identical to the application process from which test.c copied the critical state structures. Thus, test.c calls the system call `write_state()` which emulates the `exec()` system call.

Once test.c enters the kernel through `write_state()`, it finds the path name and process arguments of the correct application process through the implementation of `pfind()` as done in the first part of test.c. It is inside of the `write_state()` system call where the code must initiate `my_exec()`.

5.2.1 Altered 4.3BSD System Call: my_exec()

`write_state()` calls `my_exec()`, and it is this system call which writes the saved critical data structures into the newly created application process.
My_exec() basically follows the normal exec() code except for copying the saved state into the new application process' virtual memory.

In order to keep this alteration "clean", we tried to avoid altering existent code by simply adding new code to the exec() system call. Nevertheless, we had to alter the calling sequence code of the exec() system call.

The original exec() system call, when invoked, automatically sets up a path, recognizes its argument variable(s) (argv), and sets up its environment (envp). However, in order to enable my_exec() to work, we hard-wired the path to point to the halted application process' executable file, a.out, we set the argument variable to point to a.out, and we zeroed the environment variable pointer. The following illustrates the differences between the my_exec() system call and the exec() system call:

The exec() system call,

```c
int
execve(p, uap, retval)
    struct proc *p;
    struct args {
        char  *path;  /* Argument names from POSIX.1 3.1.2.1 */
        char  **argv;
        char  **envp;
    }    *uap;
    int *retval;
```

The my_exec() system call,

```c
int
my_exec(p, uap, retval)
    struct proc *p;
    struct args **uap;
    int * retval;
```

This is the only code which was changed for process recovery, and it is quite obvious that the two calls only differ by the variable uap. As mentioned previously, the handle, **uap, enables the kernel to pass data between user and kernel modes. The "struct args" is a structure of type "struct my_other_uap" which draper_code.h contains and which relates variables used for the second
phase of recovery. Appendix D shows the construction of draper_code.h and how the two structures, my_uap and my_other_uap, are set up.

We only added code to the rest of the exec() system call for recovery. The very beginning of my_exec() and the very end of my_exec() contain this new code. The following section depicts the steps of my_exec() illustrating only the newly appended code. Appendix C contains the actual code for my_exec() minus all superfluous printf() statements.

5.2.2 My_exec() Algorithm

Located in write_state(),
{
    Call my_exec( p, uap, retval).
    {
        Set up new variables needed by my_exec().

        Use pfind() to find process test.c.
        Obtain sizes of critical structures from test.c.
        Obtain virtual addresses of saved critical structures
        from test.c.

        Normal exec() system call programming . . .

        Copy the saved register states.

        Set up the destination and source map addresses.
        Set up addresses for vm_map_copy() system function.

        Copy the text using vm_map_copy() system function.
        Print the return value to check for any error.

        Copy the data using vm_map_copy() system function.
        Print the return value to check for any error.

        Copy the user stack using vm_map_copy() system function.
        Print the return value to check for any error.
Copy the kernel stack using \texttt{vm\_map\_copy}().
Print the return value to check for any error.

Copy the PCB using \texttt{vm\_map\_copy}().
Print the return value to check for any error.

\}
\}
return(0)
\}

\subsection*{5.2.3 Data Passing From the Beginning of \texttt{My\_exec}() to the End}

For clarity, let process number 1 be the correct application process, process number 2 be the \texttt{forked}() process, and process number 3 be the newly created application process. The reason for this is that each of the previous processes has its own process ID number. In short, the \texttt{forked}() process, number 2, and the newly created process, number 3, have different process IDs as represented by the numbers 2 and 3. In actuality, \texttt{my\_exec}() destroys process number 2 when creating number 3. \texttt{Write\_state}() obtains the addresses and sizes of the critical state structures from \texttt{test.c} at the top of \texttt{my\_exec}(). The beginning of \texttt{my\_exec}(), after the newly appended code, begins to set up the operating system for a new process. For instance, \texttt{my\_exec}() allocates new memory, gathers arguments for a new process, and changes the name from number 2 to number 3. After this, \texttt{my\_exec}() destroys the virtual memory of number 2 by a call to \texttt{vm\_map\_remove}(). Finally, the code sets up the initial register state for executing in the new image. Therefore, anything which needs to make reference to number 2, the \texttt{forked}() process, needs to make the reference before the call to \texttt{vm\_map\_remove}(). After this call, number 2's virtual memory space no longer exists. We discovered this fact quickly after crashing the system several times while trying to access number 2's virtual memory space after the call to \texttt{vm\_map\_remove}() because UNIX is not robust for users who try to access nonexistent virtual memory spaces or nonexistent pages in an existing virtual memory space. Thus, all addresses which \texttt{my\_exec}() must pass from the top of the program to the bottom must be saved in integer variables. For example, if an address in process 2's virtual memory space is saved at the beginning of \texttt{my\_exec}() using a "char *", the bottom of \texttt{my\_exec}() can not reference what the
“char *" pointer points to because after vm_map_remove(), the virtual memory for the “char *" reference no longer exists.

5.2.4 Verifying vm_map_copy() of the Text, Data, and User Stack

Once again, we had to test the code in order to verify that vm_map_copy() copied the correct structures. This time, however, we had to check if the kernel copied the critical state structures into the recovered process. For an example, we coded my_exec() to allocate memory in the kernel by using the MALLOC() kernel system call which is different from the malloc() user system called utilized by test.c. My_exec() allocates this memory after it copies the critical structures into the new application process' virtual memory. Next, my_exec() copies the text, user stack, and data from the newly created application process' virtual memory, which has just been written over with the saved critical states, into this allocated memory space which the code then prints out to the computer screen. By printing this out onto the screen, we were then able to compare this printout with the files of the critical structures which test.c has saved, in this case text.file, data.file, and stack.file respectively. This test demonstrated their equivalence.

Additionally, the correct application process has a buffer instantiated in its data area. When the program runs, it prints the buffer’s contents out to screen before the buffer is filled with any data. The program then fills in the buffer with data and enters an infinite while loop which contains a counter with a printf() statement. The following algorithm illustrates this program:

Define databuffer
Initialize main().
{
    Print out contents of databuffer.
    Fill in data buffer with string.

    Infinite while(1) loop.
    {
        Print incremented counter.
        Sleep process for 1 second.
    }
}
Thus, when the application process is first started, it prints out zeros for the databuffer’s contents since there is nothing located in the buffer when the print statement is executed. Then the program begins to print numbers on the screen starting with 1 and sleeping one second between each number printed. However, the recovered process prints out the contents of the databuffer when the print statement is executed. Furthermore, the counter begins counting from the last number which the correct application process printed before recovery was initiated. This proves that recovery has successfully copied the text, data, and stack of the halted application process. We performed this test before we copied the kernel stack, PCB, and saved registers.

Finally, the return values for the `vm_map_copy()` functions returned values of zero implying error-free copies of the text, data, and user stacks. However, before successfully copying the state structures into the new process’ virtual memory, the return statements gave errors such as “Cannot Allocate Memory” and “Permission Denied.” We then discovered that before we could copy the state structures into the recovered process, we had to delete the virtual memory maps of the recovered process’ text, data, and user stack employing the kernel function `vm_map_delete()`. The registers, however, could be simply overwritten.

### 5.2.5 Copying the Kernel Stack and Process Control Blocks

Copying the kernel stack and process control block into the new application process caused the kernel to crash. The kernel would not allow a `vm_map_delete()` of the old kernel stack or process control block, and it would not allow a simple overwrite of the two saved structures over the old kernel stack and process control block. The function we studied in order to copy the kernel stack and the process control block is called `cpu_fork()` located in “/usr/src/sys/i386/i386/vm_machdep.c”. The call is part of the `fork()` system call which copies and updates the kernel stack and PCB. Chapter Six further explains the difficulties in copying the kernel stack and PCB.

### 5.2.6 Copying the Saved Registers

`My_exec()` makes a call to `exec_set_state()` before it copies the registers or the text, data, and user stack. `Exec_set_state()` sets up the initial register state for
executing in the new image. The frame pointer and stack pointer are set to point to the same address at the top of the user stack. The program counter is set to point to the entry of the text segment while all other registers are initialized with zeros. Thus, after this call, we coded my_exec() to redefine the initialized frame pointer, stack pointer, and program counter to the copied ones from the halted application process. However, the process would do a coredump and enter zombie state.
CHAPTER SIX

6.0 Conclusions

6.1 Success of Stateful Task Recovery

The recovery process successfully copies the text, data, and user stack and writes them into the new application process. However, the recovery process is deficient in that it cannot copy any of the registers such as the program counter, frame pointer, and stack pointer; and therefore, it does not begin execution at the desired instruction.

The entire process recovery takes the CPU an average total time, user mode plus system mode, of .133 seconds for recovery to reach its present working state. The first part of recovery from the instantiation of main() in test.c to the end of Phase 1 requires .100 seconds to complete. The second half of recovery, Phase 2, requires .033 seconds to complete before the process successfully begins to run. The time was measured using a system function called clock() which measures the amount of processor time used since the invocation of the calling process.

As a comparison of the job which stateful task recovery achieves, memory realignment requires 4.71 seconds to reach an operational repaired state on C1. Of course an entire processor’s memory must be copied by memory realignment, and it might take stateful task recovery the same time to achieve memory alignment; however, stateful task recovery can recover a failed process without the need of copying an entire processor’s memory.

6.2 Kernel Stack Difficulties

We determined that the kernel stack which read_state() copied from the correct application process could not be overwritten onto the kernel stack of the newly created process. The reason is that the kernel stack contains information for a process when it is in a system call. This information enables a process to return to user mode, from the kernel, to the next user instruction the CPU should execute. Consequently, if a sleep() system call suspends the correct application process, its per-process kernel stack has the information that it is in a sleep() system call. However, when copied over into the newly created application
process, which is in a my_exec() system call, the newly created process does not know where to return to because its kernel stack is destroyed and written over with another which is not compatible with its current call sequence. The following paragraph further explains the situation by discussing how the operating system handles signals.

### 6.2.1 Kernel Handling of Signals

Signals are designed to be software equivalents of hardware interrupts or traps, and they can be delivered asynchronously to a process through application-specified signal handlers, or may result in default actions, such as process termination, carried out by the system [LMKQ89]. Some examples of signals are SIGSTOP (stop process), SIGKILL (terminate process), and SIGQUIT (create core image). The important feature of signals is that when the kernel takes action on them, nothing is on the per-process kernel stack. Signals must be handled on return from kernel mode to user mode when the kernel stack for the running process is empty. Similarly, since stateful task recovery suspends the correct application processes in sleep() system calls, and the newly created application process is in a my_exec() system call, the system cannot alter the kernel stack of the recovered process. Furthermore, the kernel stack cannot be copied on return from a system call either because there is no kernel stack to copy. The only process which can do a kernel stack copy is the fork() system call when it is creating a new process. The parent process actually makes a context switch to itself so that the kernel can access the current and new user structures simultaneously in the kernel’s virtual address space [LMKQ89]. This ensures that the kernel stack is at the same address in each process. With this ability, the current per-process execution stack for the kernel can be copied from the parent to the new process.

### 6.2.2 Ramifications of the Inability to Copy the Kernel Stack

The program counter in the kernel stack would have given recovery the ability to execute the next user instruction which the halted application process would execute when brought out of its sleep() system call. Since recovery could not copy the kernel stack, we had to obtain the program counter in some other manner.
Furthermore, we realized that there is no need to copy the kernel stack because the newly created application process does not need to “know” what system call the correct application process is in. The new application process only needs the user program counter of the halted process in order to know what next user instruction it should perform on returning from the my_exec() system call to user mode, and the p_regs structure contains this information.

6.3 Process Control Block Difficulties

Because the main goal of recovery is to copy the state of an application process, it only seems reasonable that the recovery process should copy the process control block since it is the structure which saves the user and kernel mode execution states. Nevertheless, the process control block returned a stack pointer, frame pointer, and program counter which were all located in the kernel’s virtual memory space.

The process control blocks of two programs were printed out on the screen. One program ran continuously in a while(1) loop, and the second program was test.c. Both process control blocks yielded addresses for the stack pointer, frame pointer, and program counter which were located above the user stack. The while(1) program should have given a stack pointer and frame pointer located somewhere in the user stack segment and a program counter located somewhere in the user text segment. The process control block for test.c should have given a value for the kernel stack pointer since it is in a system call, yet it contained a zero for the kernel stack pointer. Thus, we determined that the process control block did not supply the recovery process with the correct information.

6.4 Saved Registers During System Calls

The kernel saves a process' state in special registers when making system calls. The kernel references these registers through using the p_regs variable defined in the proc structure. The most important registers for a stateful recovery were the stack pointer, frame pointer, and program counter because we wanted the recovered process to begin execution at the exact location where the halted application process would begin execution after returning from the sleep() system call. However, when the recovery process copied the saved stack pointer,
frame pointer, and program counter into the new application process, the new process entered zombie state and created a core dump of the process. As long as the program counter starts at the beginning of the user text segment and the stack pointer and frame pointer start at the top of the user stack segment, the recovery process executes correctly. However, the execution does not start from an initial state but at the state where the application process was halted as described in section 5.2.3. Even though the program counter enters at the beginning of text segment and the stack and frame pointers start at the top of the user stack segment, the recovered process does not reinitialize any of the data.

6.5 Recommendations/Future Work

Determining how to copy registers such as the stack pointer, frame pointer, and program counter should be the first priority of any future work on stateful task recovery. Once recovery can copy these registers, the recovered process will then begin execution at the same instruction as the halted application process after both processes enter user mode. As a final test, recovery could then be tried on the FTPP; the file I/O would be replaced by voting.

Another possible area of work could be concentrating on copying the kernel stack and process control block of the halted process. One possible beginning place would be studying the file “vm_machdep.c” where cpu_fork() is located. It is here where the kernel copies the kernel stack, process control block, and the p_regs structures when creating a child process.

The next topic which needs investigating is how to get rid of the initial constraints which this research placed on recovery. For instance, we assumed no open files, no process to process communication, and no shared memory. The clue for determining how to eliminate the constraints is looking at the proc structure which actually contains variables concerning open file structures, pending signals, and accounting statistics.

Furthermore, a better method for setting up a process’ path, arguments, and environment needs to be worked on. Currently we hardwire the path and argument variables with a.out, and we zero the environment variable. Additionally, the recovery process now works only by the user typing in the PID of the application process which needs to be recovered. The code must be set up
so that FDIR will automatically do a pfnd() on the correct application process with no user interface; in short, we want stateful task recovery to be transparent. Furthermore, test.c needs a way to determine the size of the saved files when opening them before calling write_state(). With these improvements, stateful task recovery could be used in recovering much more difficult processes.
Appendix A: Glossary of Terms and Acronyms

This glossary contains a list of the notation used throughout this thesis.

ACRONYMS:

AFTA  Army Fault Tolerant Architecture
AGN&C  Apollo Guidance, Navigation, and Control
BSD  Berkeley Software Distributions
CPU  Central Processing Unit
DFBW  Digital Fly-by-Wire
DMERT  Duplex Multi-Environment Real-Time
DPS  Data Processing System
FCR  Fault Containment Region
FDIR  Fault Detection, Isolation, and Recovery
FMEA  Failure Modes and Effects Analysis
FTDB  Fault Tolerant Data Bus
FTPP  Fault Tolerant Parallel Processor
IMP  Integrated Micro Products
I/O  Input/Output
IOCs  Input/Output Controllers
LAN  Local Area Network
LRU  Line Replaceable Unit
MMU  Memory Management Unit
NDI  Non-Developmental Item
NE  Network Element
OLTP  On Line Transaction Processing
PC  Power Conditioners
PCB  Process Control Block
PE  Processing Element
PID  Process Identifier
RAM  Random Access Memory
RISC  Reduced Instruction Set Computer
ROM  Read Only Memory
SBB  System Building Blocks
VG  Virtual Group
VLSI  Very Large Scale Integration

TERMS:

a.out  halted applications process' executable file
bus  electrical and mechanical interconnection for the components of a computer
bss  "block started by symbol", uninitialized data that is initialized to zero when the program is loaded into memory
child process the direct descendant of a process which invokes the fork()
system call
core dump a memory image dump (core) of the execution of a process
code which contains read_state(), write_state(), and
my_exec()
draper_code.c draper_code.h code which contains cases and structures for recovery
eval() system call which overlays the virtual address space of a
process with the contents of an executable image
fork() system call which creates new processes
frame pointer bonds together a chain of activation records into a single
activation record called a frame
my_exec() stateful task recovery version of exec() system call; writes
saved structures into the newly created process
Phase 1 first part of stateful task recovery
Phase 2 second part of stateful task recovery
pointer a variable that contains the address of a variable
p_regs holds saved registers during a system call
process a task or thread of execution
proc structure a structure which contains the information needed to
manage a process
program counter contains the virtual address of the instruction the CPU will
execute next
read_state() system call created for recovery which attains critical
structure sizes and copies the critical structures so test.c can
save them to files
stack pointer contains the virtual memory address of the uppermost
element of the stack
test.c code which invokes read_state() and write_state(); acts like
the recovery part of FDIR
u area structure which contains the kernel stack, user structure, and
process control block
write_state() system call created for recovery which calls my_exec()
Appendix B: test.c

#include <stdio.h>
#include <stdlib.h>
#include </sys/sys/types.h>
#include </sys/sys/stat.h>
#include </sys/sys/times.h>
#include <fcntl.h>
#include </sys/sys/wait.h>
#include <errno.h>
#include <unistd.h>
#include </sys/syscall.h>
#include "draper_code.h"
#include </sys/i386/include/vmparam.h>
#include </sys/i386/include/vmlayout.h>

struct args my_other_uap;

main()
{
    int retval0, retval, retval1, retval2, i;
    int fd_text, fd_data, fd_stack;
    int fd_kstack, fd_pcb;
    int fd_pc, fd_fp, fd_sp, fd_reg_seflags;
    int fd_reg_eax, fd_reg_edx, fd_reg_ecx;
    int fd_reg edi, fd_reg esi, fd_reg ebx;
    struct my_args my_uap;
    pid_t pid;
    int open_text_file, open_data_file, open_stack_file;
    int open_kstack_file, open_pcb_file;
    int open_pc_file, open_fp_file, open_sp_file;
    int open_reg_seflags_file, open_reg_eax_file, open_reg_edx_file;
    int open_reg_ecx_file, open_reg edi_file;
    int open_reg esi_file, open_reg ebx_file;
    float time_phase1, time_phase2, total_time;

    char buf_path[] = "/usr/usr/recovery/source/a.out";
    char buf_argv[] = "a.out";
    char buf_envp[] = "0";
    char *buf_path_ptr, *buf_argv_ptr, *buf_envp_ptr;

    printf("Enter pid number\n");
    scanf("%d", &i);
    my_uap.desired_pid = i;

    my_uap.command = GET_TARG_SIZES;
retval0 = syscall(151,&my_uap);

my_uap.text_ptr = malloc((my_uap.text_size)*NBPG);
if (((int)my_uap.text_ptr & (int)0xffff) != 0)
| perror("error in text malloc");
| exit(0);
|
my_uap.command = COPY_TEXT_PAGES;
retval = syscall(151,&my_uap);

if ((fd_text = creat("text.file",MYMODE)) < 0)
{ |
| perror("create");
|
if ((write(fd_text,my_uap.text_ptr,(my_uap.text_size)*NBPG))
|!=(my_uap.text_size)*NBPG)
{ |
| perror("write");
|
my_uap.data_ptr = malloc((my_uap.data_size)*NBPG);
if (((int)my_uap.data_ptr & (int)0xffff) != 0)
| perror("error in data malloc");
| exit(0);
|
my_uap.command = COPY_DATA_PAGES;
retval = syscall(151,&my_uap);

if ((fd_data = creat("data.file",MYMODE)) < 0)
{ |
| perror("create");
|
if ((write(fd_data,my_uap.data_ptr,(my_uap.data_size)*NBPG))
|!=(my_uap.data_size)*NBPG)
{ |
| perror("write");
|
my_uap.stack_ptr = malloc((my_uap.stack_size)*NBPG);
if (((int)my_uap.stack_ptr & (int)0xffff) != 0)
| perror("error in stack malloc");
| exit(0);
my_uap.command = COPY_STACK_PAGES;
retval1 = syscall(151,&my_uap);

if ((fd_stack = creat("stack.file",MYMODE)) < 0)
{
    perror("create");
}

if ((write(fd_stack,my_uap.stack_ptr,(my_uap.stack_size)*NBPG))
!=(my_uap.stack_size)*NBPG)
{
    perror("write");
}

my_uap.kstack_ptr = malloc(my_uap.kstack_size);

my_uap.command = COPY_KSTACK_PAGES;
retval1 = syscall(151,&my_uap);

if ((fd_kstack = creat("kstack.file",MYMODE)) < 0)
{
    perror("create");
}

if ((write(fd_kstack, my_uap.kstack_ptr, my_uap.kstack_size))
!=(my_uap.kstack_size))
{
    perror("write");
}

my_uap pcb_ptr = malloc(my_uap.pcb_size);

my_uap.command = COPY_PCB_PAGES;
retval1 = syscall(151,&my_uap);

if ((fd pcb = creat("pcb.file",MYMODE)) < 0)
{
    perror("create");
}

if ((write(fd pcb, my_uap.pcb_ptr, my_uap.pcb_size)) !=(my_uap.pcb_size))
{
    perror("write");
}
my_uap.pc_ptr = malloc(my_uap.pc_size);
if (((int)my_uap.pc_ptr & (int)0xffff) != 0)
{ perror("error in pc malloc");
  exit(0);
}

my_uap.command = COPY_PROG_COUNT;
retval = syscall(151,&my_uap);

if ((fd_pc = creat("pc.file",MYMODE)) < 0)
{
  perror("create");
}

if ((write(fd_pc,my_uap.pc_ptr, my_uap.pc_size)) !=(my_uap.pc_size))
{
  perror("write");
}

my_uap.fp_ptr = malloc(my_uap.fp_size);

my_uap.command = COPY_FRAME_POINT;
retval = syscall(151,&my_uap);

if ((fd_fp = creat("fp.file",MYMODE)) < 0)
{
  perror("create");
}

if ((write(fd_fp,my_uap.fp_ptr, my_uap.fp_size)) !=(my_uap.fp_size))
{
  perror("write");
}

my_uap.sp_ptr = malloc(my_uap.sp_size);

my_uap.command = COPY_STACK_POINT;
retval = syscall(151,&my_uap);

if ((fd_sp = creat("sp.file",MYMODE)) < 0)
{
  perror("create");
}

if ((write(fd_sp,my_uap.sp_ptr, my_uap.sp_size)) !=(my_uap.sp_size))
{
  perror("write");
}
{
    perror("write");
}

my_uap.reg_seflags_ptr = malloc((my_uap.reg_seflags_size));

my_uap.command = COPY_REGS_SEFLAGS;
reval1 = syscall(151,&my_uap);

if ((fd_reg_seflags = creat("reg_seflags.file",MYMODE)) < 0)
{
    perror("create");
}

if ((write(fd_reg_seflags,my_uap.reg_seflags_ptr,(my_uap.reg_seflags_size))
!=(my_uap.reg_seflags_size))
{
    perror("write");
}

my_uap.reg_eax_ptr = malloc((my_uap.reg_eax_size));

my_uap.command = COPY_REGS_SEAX;
reval1 = syscall(151,&my_uap);

if ((fd_reg_eax = creat("reg_eax.file",MYMODE)) < 0)
{
    perror("create");
}

if ((write(fd_reg_eax,my_uap.reg_eax_ptr,(my_uap.reg_eax_size))
!=(my_uap.reg_eax_size))
{
    perror("write");
}

my_uap.reg_edx_ptr = malloc((my_uap.reg_edx_size));

my_uap.command = COPY_REGS_SEDX;
reval1 = syscall(151,&my_uap);

if ((fd_reg_edx = creat("reg_edx.file",MYMODE)) < 0)
{
    perror("create");
}
if ((write(fd_reg_edx,my_uap.reg_edx_ptr,(my_uap.reg_edx_size)))
  !=(my_uap.reg_edx_size))
{
  perror("write");
}

my_uap.reg_ecx_ptr = malloc((my_uap.reg_ecx_size));

my_uap.command = COPY_REGS_sECX;
retval1 = syscall(151,&my_uap);

if ((fd_reg_ecx = creat("reg_ecx.file",MYMODE)) < 0)
{
  perror("create");
}

if ((write(fd_reg_ecx,my_uap.reg_ecx_ptr,(my_uap.reg_ecx_size)))
  !=(my_uap.reg_ecx_size))
{
  perror("write");
}

my_uap.reg_edi_ptr = malloc((my_uap.reg_edi_size));

my_uap.command = COPY_REGS_sEDI;
retval1 = syscall(151,&my_uap);

if ((fd_reg_edi = creat("reg_edi.file",MYMODE)) < 0)
{
  perror("create");
}

if ((write(fd_reg_edi,my_uap.reg_edi_ptr,(my_uap.reg_edi_size)))
  !=(my_uap.reg_edi_size))
{
  perror("write");
}

my_uap.reg_esi_ptr = malloc((my_uap.reg_esi_size));

my_uap.command = COPY_REGS_sESI;
retval1 = syscall(151,&my_uap);

if ((fd_reg_esi = creat("reg_esi.file",MYMODE)) < 0)
{
  perror("create");
}
if (((write(fd_reg esi, my_uap.reg_esi_ptr, (my_uap.reg_esi_size)))
!=(my_uap.reg_esi_size))
{
    perror("write");
}

my_uap.reg_ebx_ptr = malloc((my_uap.reg_ebx_size));

my_uap.command = COPY_REGS_sEBX;
retvall = syscall(151, &my_uap);

if (((fd_reg_ebx = creat("reg_ebx.file", MYMODE)) < 0)
{
    perror("create");
}

if (((write(fd_reg_ebx, my_uap.reg_ebx_ptr, (my_uap.reg_ebx_size)))
!=(my_uap.reg_ebx_size))
{
    perror("write");
}

time_phase1 = clock();
printf("time_phase1 is %.4f seconds\n", time_phase1/60);

*****************************************************************************
          Now this is the second phase of Stateful Task Recovery.
*****************************************************************************

open_text_file = open("text.file", O_RDWR, 0);
open_data_file = open("data.file", O_RDWR, 0);
open_stack_file = open("stack.file", O_RDWR, 0);
open_kstack_file = open("kstack.file", O_RDWR, 0);
open_pcb_file = open("pcb.file", O_RDWR, 0);
open_pc_file = open("pc.file", O_RDWR, 0);
open_fp_file = open("fp.file", O_RDWR, 0);
open_sp_file = open("sp.file", O_RDWR, 0);
own_reg_seflags_file = open("reg_seflags.file", O_RDWR, 0);
own_reg_eax_file = open("reg_eax.file", O_RDWR, 0);
own_reg_edx_file = open("reg_edx.file", O_RDWR, 0);
own_reg_ecx_file = open("reg_ecx.file", O_RDWR, 0);
own_reg_edi_file = open("reg_edi.file", O_RDWR, 0);
own_reg_esi_file = open("reg_esi.file", O_RDWR, 0);
own_reg_ebx_file = open("reg_ebx.file", O_RDWR, 0);
my_other_uap.test1_pid = getpid();

my_other_uap.new_text = malloc((my_uap.text_size)*NBPG);
if (read(open_text_file, my_other_uap.new_text, (my_uap.text_size)*NBPG) != (my_uap.text_size)*NBPG)
    printf("read error for new_text
");

my_other_uap.new_data = malloc((my_uap.data_size)*NBPG);
if (read(open_data_file, my_other_uap.new_data, (my_uap.data_size)*NBPG) != (my_uap.data_size)*NBPG)
    printf("read error for new_data
");

my_other_uap.new_stack = malloc((my_uap.stack_size)*NBPG);
if (read(open_stack_file, my_other_uap.new_stack, (my_uap.stack_size)*NBPG) != (my_uap.stack_size)*NBPG)
    printf("read error for new_stack
");

my_other_uap.new_kstack = malloc(my_uap.kstack_size);
if (read(open_kstack_file, my_other_uap.new_kstack, my_uap.stack_size) != (my_uap.stack_size))
    printf("read error for new_kstack
");

my_other_uap.new_pcb = malloc(my_uap.pcb_size);
if (read(open_pcb_file, my_other_uap.new_pcb, my_uap.stack_size) != (my_uap.stack_size))
    printf("read error for new_pcb
");

my_other_uap.new_pc = malloc((my_uap.pc_size)*NBPG);
if (read(open_pc_file, my_other_uap.new_pc, (my_uap.pc_size)) != (my_uap.pc_size))
    printf("read error for new_pc
");

my_other_uap.new_fp = malloc((my_uap.fp_size)*NBPG);
if (read(open_fp_file, my_other_uap.new_fp, (my_uap.fp_size)) != (my_uap.fp_size))
    printf("read error for new_fp
");

my_other_uap.new_sp = malloc((my_uap.sp_size)*NBPG);
if (read(open_sp_file, my_other_uap.new_sp, (my_uap.sp_size)) != (my_uap.sp_size))
    printf("read error for new_sp
");

my_other_uap.new_reg_seflags = malloc((my_uap.reg_seflags_size));
if (read(open_reg_seflags_file, my_other_uap.new_reg_seflags, (my_uap.reg_seflags_size)) != (my_uap.reg_seflags_size))

98
printf("read error for new_reg_seflags \n");

my_other_uap.new_reg_eax = malloc((my_uap.reg_eax_size));
if (read(open_reg_eax_file, my_other_uap.new_reg_eax,
(my_uap.reg_eax_size)) != (my_uap.reg_eax_size))
    printf("read error for new_reg_eax \n");

my_other_uap.new_reg_edx = malloc((my_uap.reg_edx_size));
if (read(open_reg_edx_file, my_other_uap.new_reg_edx,
(my_uap.reg_edx_size)) != (my_uap.reg_edx_size))
    printf("read error for new_reg_edx \n");

my_other_uap.new_reg_ecx = malloc((my_uap.reg_ecx_size));
if (read(open_reg_ecx_file, my_other_uap.new_reg_ecx, (my_uap.reg_ecx_size))
!= (my_uap.reg_ecx_size))
    printf("read error for new_reg_ecxs \n");

my_other_uap.new_reg_edi = malloc((my_uap.reg_edi_size));
if (read(open_reg_edi_file, my_other_uap.new_reg_edi, (my_uap.reg_edi_size))
!= (my_uap.reg_edi_size))
    printf("read error for new_reg_edi \n");

my_other_uap.new_reg_esi = malloc((my_uap.reg_esi_size));
if (read(open_reg_esi_file, my_other_uap.new_reg_esi, (my_uap.reg_esi_size))
!= (my_uap.reg_esi_size))
    printf("read error for new_reg_esi \n");

my_other_uap.new_reg_ebx = malloc((my_uap.reg_ebx_size));
if (read(open_reg_ebx_file, my_other_uap.new_reg_ebx,
(my_uap.reg_ebx_size)) != (my_uap.reg_ebx_size))
    printf("read error for new_reg_ebx \n");

my_other_uap.text_size = (my_uap.text_size)*NBPG;
my_other_uap.data_size = (my_uap.data_size)*NBPG;
my_other_uap.stack_size = (my_uap.stack_size)*NBPG;
my_other_uap.kstack_size = my_uap.kstack_size;
my_other_uap.pcb_size = my_uap.pcb_size;
my_other_uap.pc_size = my_uap.pc_size;
my_other_uap.fp_size = my_uap.fp_size;
my_other_uap.sp_size = my_uap.sp_size;
my_other_uap.reg_seflags_size = my_uap.reg_seflags_size;
my_other_uap.reg_eax_size = my_uap.reg_eax_size;
my_other_uap.reg_edx_size = my_uap.reg_edx_size;
my_other_uap.reg_ecx_size = my_uap.reg_ecx_size;
my_other_uap.reg_edi_size = my_uap.reg_edi_size;
my_other_uap.reg_esi_size = my_uap.reg_esi_size;
my_other_uap.reg_ebx_size = my_uap.reg_ebx_size;

if ((pid = fork()) < 0)
    printf("fork error");
else if (pid == 0)
{
    buf_argv_ptr = &buf_argv[0];
    buf_envp_ptr = (char *)NULL;

    my_other_uap.path = buf_path;
    my_other_uap.argv = buf_argv_ptr;
    my_other_uap.envp = buf_envp_ptr;

    syscall(152, &my_other_uap);
}

    total_time= clock();
    time_phase2 = total_time - time_phase1;
    printf("time_phase2 is %4f seconds\n",time_phase2/60);
    printf("total time is %4f seconds\n",total_time/60);

    sleep(10000);
}
Appendix C: draper_code.c

```
#include "/sys/sys/cdefs.h"
#include "/sys/sys/types.h"
#include "/sys/sys/systm.h"
#include "draper_code.h"
#include </sys/vm/vm.h>
#include </sys/i386/isa/isa.h>
#include </sys/i386/include/param.h>
#include </sys/sys/errno.h>
#include </sys/i386/include/vmparam.h>
#include </sys/i386/include/vmlayout.h>
#include </sys/sys/user.h>
#include </sys/sys/resource.h>
#include </sys/sys/malloc.h>
#include </sys/i386/include/reg.h>
#include </sys/sys/exec.h>

int
  read_state(p, my_uap, retval)
struct proc *p;
struct my_args **my_uap;
int *retval;
{
  struct proc *targ;
  char *cp_in_text, *cp_out_text;
  char *cp_in_data, *cp_out_data;
  char *cp_in_stack, *cp_out_stack;
  vm_map_t dstmap, srcmap;
  int retval_text, retval_data, retval_stack;
  extern char kstack[];
  extern int mvesp();
  int offset;
  register int *regs;

  targ = pfind((*my_uap)->desired_pid);
  *regs = (int)targ->p_regs;

  switch ((*my_uap)->command)
  {
  case GET_TARG_SIZES:
    (*my_uap)->text_size = targ->p_vmspace->vm_tsize; /* in pages */
    (*my_uap)->data_size = targ->p_vmspace->vm_dsize; /* in pages */
  ```
(*my_uap)->stack_size = targ->p_vmspace->vm_ssize; /* in pages */

(*my_uap)->kstack_size = UPAGES * NBPG;

(*my_uap)->pcb_size = sizeof(targ->p_addr->u_pcb);

(*my_uap)->pc_size = sizeof(regs[sEIP]);

(*my_uap)->fp_size = sizeof(regs[sEBP]);

(*my_uap)->sp_size = sizeof(regs[sESP]);

(*my_uap)->reg_seflags_size = sizeof(regs[sEFLAGS]);

(*my_uap)->reg_eax_size = sizeof(regs[sEAX]);

(*my_uap)->reg_edx_size = sizeof(regs[sEDX]);

(*my_uap)->reg_ecx_size = sizeof(regs[sECX]);

(*my_uap)->reg_edi_size = sizeof(regs[sEDI]);

(*my_uap)->reg_esi_size = sizeof(regs[sESI]);

(*my_uap)->reg_ebx_size = sizeof(regs[sEBX]);

break;

case COPY_TEXT_PAGES:

dstmap = &(curproc->p_vmspace->vm_map);
srcmap = &(targ->p_vmspace->vm_map);

cp_in_text = (*my_uap)->text_ptr;
cp_out_text = targ->p_vmspace->vm_taddr; /* user virtual address of text */

retval_text = vm_map_copy(dstmap, srcmap,
  cp_in_text,
  ((*my_uap)->text_size)*NBPG,
  cp_out_text,
  FALSE,FALSE);

break;

102
case COPY_DATA_PAGES:

dstmap = &(curproc->p_vmspace->vm_map);
srcmap = &(targ->p_vmspace->vm_map);

cp_in_data = (*my_uap)->data_ptr;
cp_out_data = targ->p_vmspace->vm_daddr; /* data address at pg boundary */

retval_data = vm_map_copy(dstmap, srcmap,
    cp_in_data,
    ((*my_uap)->data_size)*NBPG,
    cp_out_data,
    FALSE, FALSE);

break;

case COPY_STACK_PAGES:

dstmap = &(curproc->p_vmspace->vm_map);
srcmap = &(targ->p_vmspace->vm_map);

cp_in_stack = (*my_uap)->stack_ptr;
cp_out_stack = (caddr_t)USRSTACK - ((targ->p_vmspace->vm_ssize) * NBPG);

retval_stack = vm_map_copy(dstmap, srcmap,
    cp_in_stack,
    (targ->p_vmspace->vm_ssize * NBPG),
    cp_out_stack,
    FALSE, FALSE);

break;

case COPY_KSTACK_PAGES:

offset = mvesp() - (int)kstack;
bcopy((caddr_t)kstack + offset, (caddr_t)(*my_uap)->kstack_ptr, (unsigned)ctob(UPAGES) - offset);

break;

case COPY_PCB_PAGES:

bcopy(&targ->p_addr->upcb, (*my_uap)->pcb_ptr, sizeof(targ->p_addr->upcb));
break;

case COPY_PROG_COUNT:
    bcopy(&regs[sEIP], (*my_uap)->pc_ptr, sizeof(regs[sEIP]));
    break;

case COPY_FRAME_POINT:
    bcopy(&regs[sEBP], (*my_uap)->fp_ptr, sizeof(regs[sEBP]));
    break;

case COPY_STACK_POINT:
    bcopy(&regs[sESP], (*my_uap)->sp_ptr, sizeof(regs[sESP]));
    break;

case COPY_REGS_sEFLAGS:
    bcopy(&regs[sEFLAGS], (*my_uap)->reg_seflags_ptr, sizeof(regs[sEFLAGS]));
    break;

case COPY_REGS_sEAX:
    bcopy(&regs[sEAX], (*my_uap)->reg_eax_ptr, sizeof(regs[sEAX]));
    break;

case COPY_REGS_sEDX:
    bcopy(&regs[sEDX], (*my_uap)->reg_edx_ptr, sizeof(regs[sEDX]));
    break;

case COPY_REGS_sECX:
bcopy(&regs[sECX], (*my_uap)->reg_ecx_ptr, sizeof(regs[sECX]));
break;

case COPY_REGS_sEDI:
bcopy(&regs[sEDI], (*my_uap)->reg edi_ptr, sizeof(regs[sEDI]));
break;

case COPY_REGS_sESI:
bcopy(&regs[sESI], (*my_uap)->reg esi_ptr, sizeof(regs[sESI]));
break;

case COPY_REGS_sEBX:
bcopy(&regs[sEBX], (*my_uap)->reg ebx_ptr, sizeof(regs[sEBX]));
}

*retval = 0;
return(0);
}

/********************************************/

int
  write_state(p, uap, retval)
struct proc *p;
struct args **uap;
int *retval;
{
  if (my_exec(p, uap, retval) < 0)
    printf("my_exec error");

  *retval = 0;
  return(0);
}
/*
 * my_exec system call.
 */
int
my_exec(p, uap, rval)
struct proc *p;
struct args **uap;
int *retval;
{
  struct text *xp;
  struct exec_load *lp;
  struct ucred *cr = p->p_ucred;
  struct syscframe *sfp = (struct syscframe *) p->p_regs;
  vm_map_t map;
  int error;
  struct proc *test1;
  int offset;
  extern char kstack[];
  extern int mvesp();
  int retval3, retval4, retval5, retval6;
  int retval7, retval8, retval9, retval10;
  int retval11, retval12;
  int sizeoftext, sizeofdata, sizeofstack;
  int sizeofkstack, sizeofpcb;
  int sizeofpc, sizeoffp, sizeofsp;
  int sizeofreg_seflags, sizeofreg_eax, sizeofreg_edx;
  int sizeofreg_ecx, sizeofreg_edi, sizeofreg_esi, sizeofreg_ebx;
  char *cp_in_text, *cp_out_text;
  char *cp_in_data, *cp_out_data;
  char *cp_in_stack, *cp_out_stack;
  char *cp_in_kstack, *cp_out_kstack;
  char *cp_in_tcb, *cp_out_tcb;
  int *cp_out_pc, *cp_out_fp, *cp_out_sp;
  int *cp_out_reg_seflags, *cp_out_reg_eax, *cp_out_reg_edx;
  int *cp_out_reg_ecx, *cp_out_reg_edi, *cp_out_reg_esi, *cp_out_reg_ebx;
  int temp_pc, temp_sp, temp_fp;
  int temp_reg_seflags, temp_reg_eax, temp_reg_edx;
  int temp_reg_ecx, temp_reg_edi, temp_reg_esi, temp_reg_ebx;
  int end_text, end_data, end_stack, end_kstack, end_tcb;
  vm_map_t dst_map, src_map;
**** BSD/OS CODE REMOVED TO AVOID COPYRIGHT PROBLEMS ****
test1 = pfind((uap)->test1_pid);

cp_out_data = (uap)->new_data;
cp_out_stack = (uap)->new_stack;
cp_out_text = (uap)->new_text;
cp_out_kstack = (uap)->new_kstack;
cp_out_pcb = (uap)->new_pcb;
cp_out_pc = (int*)(uap)->new_pc;
cp_out_fp = (int*)(uap)->new_fp;
cp_out_sp = (int*)(uap)->new_sp;
cp_out_reg_seflags = (int*)(uap)->new_reg_seflags;
cp_out_reg_eax = (int*)(uap)->new_reg_eax;
cp_out_reg_edx = (int*)(uap)->new_reg_edx;
cp_out_reg_ecx = (int*)(uap)->new_reg_ecx;
cp_out_reg_edi = (int*)(uap)->new_reg_edi;
cp_out_reg esi = (int*)(uap)->new_reg_esi;
cp_out_reg_ebx = (int*)(uap)->new_reg_ebx;

temp_pc = (int*)cp_out_pc;
temp_sp = (int*)cp_out_sp;
temp_fp = (int*)cp_out_fp;
temp_reg_seflags = (int*)cp_out_reg_seflags;
temp_reg_eax = (int*)cp_out_reg_eax;
temp_reg_edx = (int*)cp_out_reg_edx;
temp_reg_ecx = (int*)cp_out_reg_ecx;
temp_reg_edi = (int*)cp_out_reg_edi;
temp_reg_ei = (int*)cp_out_reg_esi;
temp_reg_ebx = (int*)cp_out_reg_ebx;

sizeoftext = (uap)->text_size;
sizeofdata = (uap)->data_size;
sizeofstack = (uap)->stack_size;
sizeofkstack = (uap)->kstack_size;
sizeofpcb = (uap)->pcb_size;
sizeofpc = (uap)->pc_size;
sizeoffp = (uap)->fp_size;
sizeofsp = (uap)->sp_size;
sizeofreg_seflags = (uap)->reg_seflags_size;
sizeofreg_eax = (uap)->reg_eax_size;
sizeofreg_edx = (uap)->reg_edx_size;
sizeofreg_ecx = (uap)->reg_ecx_size;
sizeofreg_edi = (uap)->reg_edi_size;
sizeofreg_esi = (*uap)->reg_esi_size;
sizeofreg_ebx = (*uap)->reg_ebx_size;

**** BSD/OS CODE REMOVED TO AVOID COPYRIGHT PROBLEMS ****

/**************************************************************************/
/* Copy in saved state registers */
/**************************************************************************/

/
  sfp->sf_eip = temp_pc;
  sfp->sf_ebp = temp_fp;
  sfp->sf_esp = temp_sp;

  sfp->sf_eflags = temp_reg_seflags;
  sfp->sf edi = temp_reg_edi;
  sfp->sf esi = temp_reg_esi;
  sfp->sf ebx = temp_reg_ebx;
  sfp->sf edx = temp_reg_edx;
  sfp->sf ecx = temp_regxECX;
  sfp->sf eax = temp_reg_eax;
/

delete_text(xp);

/**************************************************************************/
/* Copy in data, stack and text from saved files */
/**************************************************************************/

dst_map = &(curproc->p_vmspace->vm_map);
src_map = &(test1->p_vmspace->vm_map);

cp_in_text = curproc->p_vmspace->vm_taddr;
cp_in_data = curproc->p_vmspace->vm_daddr;
cp_in_stack = (caddr_t)USRSTACK - INITSSIZ;
cp_in_kstack = (caddr_t)kstack + offset;
cp_in_pcb = (char*)&curproc->p_addr->u_pcb;

end_text = (int)cp_in_text + sizeoftext;
retval3 = vm_map_delete(dst_map, cp_in_text, end_text);
retval4 = vm_map_copy(dst_map,src_map,
  cp_in_text,
  sizeoftext,
  cp_out_text,
  TRUE,FALSE);
end_data = (int)cp_in_data + sizeof(data);
retval5 = vm_map_delete(dst_map, cp_in_data, end_data);
retval6 = vm_map_copy(dst_map, src_map,
                     cp_in_data,
                     sizeof(data),
                     cp_out_data,
                     TRUE,FALSE);

end_stack = (int)USRSTACK;
retval7 = vm_map_delete(dst_map, cp_in_stack, end_stack);
retval8 = vm_map_copy(dst_map, src_map,
                     cp_in_stack,
                     sizeof(stack),
                     cp_out_stack,
                     TRUE,FALSE);

/*
end_kstack = (int)cp_in_kstack + sizeof(kstack);
retval9 = vm_map_delete(dst_map, cp_in_kstack, end_kstack);
retval10 = vm_map_copy(dst_map, src_map,
                     cp_in_kstack,
                     sizeof(kstack),
                     cp_out_kstack,
                     TRUE,FALSE);

end_pcb = (int)cp_in_pcb + sizeof(pcb);
retval11 = vm_map_delete(dst_map, cp_in_pcb, end_pcb);
retval12 = vm_map_copy(dst_map, src_map,
                     cp_in_pcb,
                     sizeof(pcb),
                     cp_out_pcb,
                     TRUE,FALSE);
*/

**** BSD/OS CODE REMOVED TO AVOID COPYRIGHT PROBLEMS ****

return (0);
}
Appendix D: draper_code.h

#include </sys/sys/time.h>
#include </sys/sys/param.h>
#include </sys/sys/proc.h>

#define MYMODE 0777

#define GET_TARG_SIZES 1
#define COPY_TEXT_PAGES 2
#define COPY_DATA_PAGES 3
#define COPY_STACK_PAGES 4
#define COPY_KSTACK_PAGES 5
#define COPY_PCB_PAGES 6
#define COPY_PROG_COUNT 7
#define COPY_FRAME_POINT 8
#define COPY_STACK_POINT 9
#define COPY_REGS_sEFLAGS 10
#define COPY_REGS_sEAX 11
#define COPY_REGS_sEDX 12
#define COPY_REGS_sECX 13
#define COPY_REGS_sEDI 14
#define COPY_REGS_sESI 15
#define COPY_REGS_sEBX 16

struct my_args
{
    int desired_pid;
    int command;
    int text_size, data_size, stack_size;
    int kstack_size, pcb_size;
    int pc_size, fp_size, sp_size;
    int reg_seflags_size, reg_eax_size, reg_edx_size;
    int reg_ecx_size, reg_edi_size, reg_esi_size, reg_ebx_size;
    char *text_ptr, *data_ptr, *stack_ptr;
    char *kstack_ptr, *pcb_ptr;
    char *pc_ptr, *fp_ptr, *sp_ptr;
    char *reg_seflags_ptr, *reg_eax_ptr, *reg_edx_ptr;
    char *reg_ecx_ptr, *reg_edi_ptr, *reg_esi_ptr, *reg_ebx_ptr;
};

struct args {
    char     *path, *argv, *envp;
    char     *new_text, *new_data, *new_stack;
    char     *new_kstack, *new_pcb;
char *new_pc, *new_fp, *new_sp;
char *new_reg_seflags, *new_reg_eax, *new_reg_edx;
int test1_pid;
int text_size, data_size, stack_size;
int kstack_size, pcb_size;
int pc_size, fp_size, sp_size;
int reg_seflags_size, reg_eax_size, reg_edx_size;
int reg_ecx_size, reg_edi_size, reg_esi_size, reg_ebx_size;
}
References


