May 1996

UILU-ENG-96-2213
CRHC-96-08

Center for Reliable and High Performance Computing

UNIX-Based Operating Systems Robustness Evaluation

Yu-Ming Chang

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-96-2213 (CRHC-96-08)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA
6c. ADDRESS (City, State, and ZIP Code) 1308 W. Main St. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Ames Research Ctr., Moffett Field, CA
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 7a. 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
UNIX-BASED OPERATING SYSTEMS ROBUSTNESS EVALUATION

12. PERSONAL AUTHOR(S)
Yu-Ming Chang

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	May 1996	49

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
robustness evaluation, operating system, UNIX, exception handling, crash, resource management, workload, stress testing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Robust operating systems are required for reliable computing. Techniques for robustness evaluation of operating systems not only enhance the understanding of the reliability of computer systems, but also provide valuable feedback to system designers. This thesis presents results from robustness evaluation experiments on five UNIX-based operating systems, which include Digital Equipment's OSF/1, Hewlett Packard's HP-UX, Sun Microsystems' Solaris and SunOS, and Silicon Graphics' IRIX. Three sets of experiments were performed. The methodology for evaluation tested (1) the exception handling mechanism, (2) system resource management, and (3) system capacity under high workload stress.

An exception generator was used to evaluate the exception handling mechanism of the operating systems. Results included exit status of the exception generator and the system state. Resource management techniques used by individual operating systems were tested using programs designed to usurp system resources such as physical memory and process slots. Finally, the workload stress testing evaluated the effect of the workload on system performance by running a synthetic workload and recording the response time of local and remote user requests. Moderate to severe performance degradations were observed on the systems under stress.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

UNIX-BASED OPERATING SYSTEMS ROBUSTNESS EVALUATION

BY

YU-MING CHANG

B.S., National Taiwan University, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

ABSTRACT

Robust operating systems are required for reliable computing. Techniques for robustness evaluation of operating systems not only enhance the understanding of the reliability of computer systems, but also provide valuable feedback to system designers. This thesis presents results from robustness evaluation experiments on five UNIX-based operating systems, which include Digital Equipment's OSF/1, Hewlett Packard's HP-UX, Sun Microsystems' Solaris and SunOS, and Silicon Graphics' IRIX. Three sets of experiments were performed. The methodology for evaluation tested (1) the exception handling mechanism, (2) system resource management, and (3) system capacity under high workload stress.

An exception generator was used to evaluate the exception handling mechanism of the operating systems. Results included exit status of the exception generator and the system state. Resource management techniques used by individual operating systems were tested using programs designed to usurp system resources such as physical memory and process slots. Finally, the workload stress testing evaluated the effect of the workload on system performance by running a synthetic workload and recording the response time

of local and remote user requests. Moderate to severe performance degradations were observed on the systems under stress.

ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis advisor, Professor Ravi K. Iyer, for his guidance and to Dr. Mei-Chen Hsueh for her ideas and discussions on this subject. I would also like to thank Digital Equipment Corporation for their support, especially to Mike Pallone for his very valuable suggestions. In addition, I would like to thank our group member Tom Kraljevic for his contributions to this work and all of my friends at CRHC for answering my many questions. Finally, I would like to thank my parents for their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. INTRODUCTION

Robust operating systems are required for reliable computing. Techniques for robustness evaluation of operating systems not only enhance the understanding of the reliability of computer systems, but also provide valuable feedback to system designers.

UNIX operating systems are widely used in industry as well as academia. This thesis presents results from robustness evaluation experiments on five UNIX-based operating systems, which include Digital Equipment's OSF/1,[1] Hewlett Packard's HP-UX, Sun Microsystems' Solaris and SunOS, and Silicon Graphics' IRIX. The system platforms under evaluation are shown in Table 1.1.

Three sets of experiments were performed. The methodology for evaluation tested (1) the exception handling mechanism, (2) system resource management, and (3) system capacity under high workload stress.

[1]Digital changed the name of its UNIX operating system from DEC OSF/1 to Digital UNIX on March 14, 1995.

Table 1.1: System platforms under evaluation

System model	DEC 3000	SPARC 20	SPARC 2	HP 735	HP 715	SGI Indy
Vendor	Digital Equipment	SUN	SUN	Hewlett Packard	Hewlett Packard	Silicon Graphics
OS type	OSF/1	Solaris	SunOS	HP-UX	HP-UX	IRIX
OS version	2.1/3.0	2.3/2.4	4.1.3	9.05/10.0	9.05	5.3

First of all, an exception generator was used to evaluate the exception handling mechanism of the operating systems. During the experiment, the operating system had to keep the system in a safe state by properly handling all kinds of exceptions generated by illegal instructions, bad operands, etc. Otherwise, the machine state might be corrupted and a system crash would occur. Results included exit status of the exception generator and the system state. We were able to crash the HP 715 running HP-UX 9.05 and the SGI Indy running IRIX 5.3 within 10 minutes after the testing started.

Secondly, resource management techniques used by individual operating systems were tested using programs designed to usurp system resources such as physical memory and process slots. In OSF/1, Solaris, SunOS, HP-UX, and SGI IRIX, a single user could monopolize the system swap space. We also observed that process slots could be monopolized in Solaris, SunOS, and IRIX. In all these cases, no more processes could be started in the system.

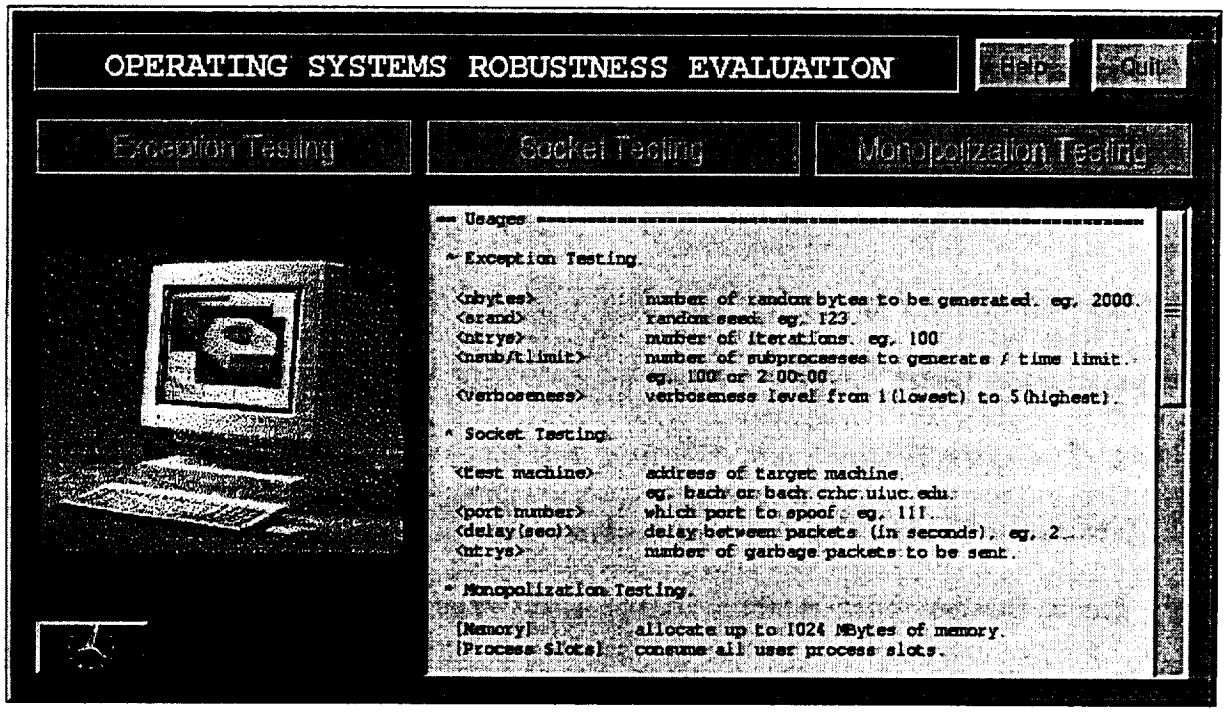

Figure 1.1: Graphical user interface of robustness testing tools.

A graphical user interface was implemented to perform the above two experiments. It allows users to input testing parameters, activate the test and monitor the results. Figure 1.1 shows this interactive interface.[2]

Finally, the workload stress testing evaluated the effect of the workload on system performance by running a synthetic workload and recording the response time of local and remote user requests. We stressed OSF/1 V3.0 and Solaris 2.4 with various disk I/O, CPU, memory, and network workload mixes. Moderate to severe performance degradations were observed on the systems under stress.

To achieve maximum repeatability, all these experiments were driven by shell scripts. Moreover, we avoided using root privileges. Instead, all experiments were conducted

[2]The socket testing shown in the figure is not included in this thesis.

in regular user mode. Therefore, all users are potentially affected by the vulnerability exposed in this study.

Throughout the course of this thesis, a *crash* is said to have occurred if a system has no interactive user response and also fails to service NFS requests. The NFS condition is particularly useful, because NFS often still works when the system appears to have hung. Two types of crashes are described in this study. One is a system hang with no NFS response. In the other case, the system panics and dumps a memory image.

The remainder of the thesis is organized as follows. Chapter 2 summarizes the related research. Chapter 3 describes system platforms, including hardware and operating systems under evaluation. Chapter 4 details the exception handling analysis and the results including system crashes. Chapter 5 concentrates on the resource management experiment and its results. Chapter 6 presents the workload stress testing, and the performance degradation is evaluated. Finally, Chapter 7 summarizes the major results in this study and suggests future work.

2. RELATED WORK

Testability and reliability issues of software have been investigated extensively. The study in [1] overviewed the fundamental issues in reliability. The concept of testability and its use in reliability assessment was presented in [2].

A large number of testing models and reliability models have been proposed. A software usage model was developed in [3] to characterize the population of intended uses of the software. Based on the software usage model, statistical testing is able to find the failures that will occur most frequently in operational use early in the testing cycle. In [4], three models for the behavior of software failures were proposed to predict reliability growth by predicting failure rates and mean times to failure. The research in [5] reviewed a number of reliability models and predicted the faults in the microcode for the IBM 4381 and the IBM 9370 families of computer systems. In [6], two mathematical models based on structural computer systems were investigated. Several cost related reliability measures were also studied on operating environments including DOS and UNIX.

A lot of software testing techniques have been developed. The empirical evidence in [7] showed that the testing method does affect the reliability estimates. In [8], it was argued that more experimental work in software testing was necessary in order to classify testing techniques in such a way that is useful to the software engineer. Automatic test case generation algorithms were introduced in [9] to perform load testing for telecommunications software systems. The reliability as a function of the degree of system degradation experienced was also presented. The research in [10] described the on-system data logging process and analysis methodology to measure system, product and operating system reliability. The automated data collection process, which collects on-system data logging information from customer sites, was developed by Digital Equipment Corporation. In [11], the failures in Tandem's NonStop-UX operating system were investigated and categorized. Both software failures from the field and failures reported by Tandem's test center were covered in this analysis.

Fault injection has been applied in software testing. The research in [12] presented a fault injection and monitoring environment (FINE) as a tool to study fault propagation in the UNIX kernel. A fault propagation study for Sun Microsystems' SunOS 4.1.2 was described in [13]. It was shown that memory faults and software faults usually have a long latency while bus faults and CPU faults tend to crash the system immediately.

Since the advent of 64-bit architectures, UNIX vendors have worked on defining a set of interfaces and a 64-bit C programming model for data representation. Companies such as Silicon Graphics, Digital, and HAL Computer already have 64-bit versions of

UNIX [14]. The study in [15] proposed both a short and long term plan for the evolution of the UNIX operating system to 64-bit architectures.

This thesis shows important results. It presents three sets of experiments to evaluate the robustness of UNIX-based operating systems. It covers issues such as exception handling ability, resource management, and performance degradation under high workload stress. In addition, it provides a reasonable comparison among the operating systems under evaluation. Two 64-bit system platforms, Digital and Silicon Graphics systems, and their operating systems are also included. Finally, it exposes the vulnerable aspects of the systems under evaluation and offers valuable feedback to the system designers.

3. SYSTEM PLATFORMS UNDER EVALUATION

3.1 Test Configuration

In this study, we evaluated the robustness of five UNIX-based operating systems from four computer vendors. The system platforms under evaluation included:

- DEC 3000 workstation running OSF/1 V2.1 and then OSF/1 V3.0.[1]
- Sun SPARCstation 20 running Solaris 2.3 and then Solaris 2.4.
- Sun SPARCstation 2 running SunOS 4.1.3.
- HP 735/125 workstation running HP-UX 9.05 and then HP-UX 10.0.
- HP 715/64 workstation running HP-UX 9.05.
- SGI Indy running IRIX 5.3.

[1]Since the testing of DEC OSF/1 V3.0, Digital has made significant changes to many aspects of their UNIX operating system now known as Digital UNIX. In February of 1995 they released V3.2 and as of May 1996 are shipping a major new release of Digital UNIX V4.0. A future comparison of Digital UNIX V4.0 against the other operating systems would be desirable to have a similar comparison of available releases.

Table 3.1: System information of testing platforms

System model	DEC 3000	SPARC 20	SPARC 2	HP 735	HP 715	SGI Indy
Vendor	Digital Equipment	SUN	SUN	Hewlett Packard	Hewlett Packard	Silicon Graphics
CPU model	RISC 21064	Super SPARC	SPARC	PA-RISC 7150	PA-RISC 7100	MIPS 4600
Clock speed	175 MHz	60 MHz	40 MHz	125 MHz	64 MHz	133 MHz
CPU word size	64 bits	32 bits	32 bits	32 bits	32 bits	64 bits
OS type	OSF/1	Solaris	SunOS	HP-UX	HP-UX	IRIX
OS version	2.1/3.0	2.3/2.4	4.1.3	9.05/10.0	9.05	5.3
Primary cache	8kd/8ki	36k	16k	256kd/256ki	256k	16kd/16ki
Secondary cache	2MB	1MB	N/A	N/A	N/A	0.5MB
Main memory	64 MB	32 MB	32 MB	256 MB	32 MB	32 MB

Table 3.1 shows detailed system information of each platform. These RISC-based systems (with Alpha, SPARC, SuperSPARC, PA-RISC 7150, PA-RISC 7100 and MIPS processors) are widely used both in industry and academia. They are either entry level or mid range workstations, with processor speeds ranging from 40 to 175 MHz. Note that the DEC 3000 system and the SGI Indy have 64-bit architectures while the others are 32-bit systems.

The UNIX-based operating systems under evaluation include Digital Equipment's OSF/1, Hewlett Packard's HP-UX, Sun Microsystems' Solaris, SunOS, and Silicon Graphics' IRIX. In the progress of this study, we upgraded some operating systems with their latest versions available at the time of testing.

3.2 Operating Systems Under Evaluation

All the operating systems we evaluated, except SunOS 4.1.3, are based on System V Release 4,[2] which has gained broad industry acceptance as the standard UNIX environment. Each of the operating systems is briefly described in the following subsections.

3.2.1 Digital Equipment's OSF/1

The DEC OSF/1 Operating System V3.0 is a 64-bit kernel architecture based on Carnegie Mellon University's Mach V2.5 kernel design with components from Berkeley Software Distribution (BSD) 4.3 and 4.4, UNIX System V, and other sources. OSF/1 V3.0 supports for *symmetrical multiprocessing* (SMP), which allows multiple threads, from the same or different tasks, to run concurrently on different processors. OSF/1 V3.0 is qualified on 12 CPUs with no architecture limits. Processor affinity, the ability to tie a process to a processor is also supported.[3]

---

[2]UNIX System V Release 4 (SVR4) is a UNIX standard which combines the best features of System V, BSD, XENIX, and SunOS.

[3]Digital has added the following SMP functionality to their Digital UNIX V3.2 release:

- Multiple threads from the same task or different tasks can be run concurrently on different processors.
- Unattended Reboot - On a hard failure of a non-boot processor, the OS will tag the failing CPU and automatically reboot the system, without enabling the defective CPU.
- Start/Stop CPU - Ability to stop/start a specified non-boot processor.
- Processor Sets - Ability to dedicate a process, or set of processes, to a specific processor or set of processors. Processes sets can also be used to partition the available processors among a set of users.

3.2.2 Sun Microsystems' Solaris

While SunOS 4 is derived from Berkeley's UNIX (BSD), Solaris 2.x uses a kernel based on UNIX System V Release 4.0 (SVR4). Solaris 2.x is designed to support *multiprocessing* (MP) and *multithreaded* (MT) applications, affording users the advantages of MP/MT performance gains on desktop and server systems. Multithreading and multiprocessing boosts performance levels for compute-intensive and I/O-intensive applications such as multimedia, graphics, and file service.

3.2.3 Hewlett Packard's HP-UX

Hewlett Packards UNIX Operating System or HP-UX is based on both System V and BSD. HP-UX also supports symmetric multiprocessing, providing scaling of application performance across multiple processors using a single version of the operating system.

Core system configuration is conducted with *System Administration Manager* (SAM). SAM allows the administrator to perform all major administrative functions using an intuitive graphical user interface that leads the administrator through the choices in a given task.

HP-UX 10.0 has additional reliability features. It protects data integrity with a journaled file system, VxFS (the Veritas file system) [16]. Compared to the BSD 4.2 HFS and NFS, VxFS has superior data integrity, recovery, and performance. It also provides resilience to memory faults. The diagnostic system and the operating system can mark bad pages and then avoid using them, therefore preventing system panics [17].

3.2.4 Silicon Graphics' IRIX

The operating system included with SGI workstations is their version of UNIX, IRIX. IRIX is a mix of AT&T System V, Release 4, and BSD. IRIX 5.3 is upwardly compatible, providing binary compatibility with applications developed under IRIX 4 and 5. IRIX 5.3 provides new features and enhanced performance, including parallelized TCP/IP, better virtual memory performance, and Caching File System (CFS) support, which uses local disk to cache remote data, reducing network traffic, speeding up application response time, and allowing use of smaller local disks.

4. EXCEPTION HANDLING ANALYSIS

This chapter describes a set of experiments which were performed based on the exception generator *Crashme* [18]. The purpose was to evaluate the exception handling mechanism of each testing platform. Specifically, we invoked single or multiple calls to Crashme in each system and let Crashme run for hours. During the experiment, the operating system had to keep the system in a safe state by properly handling all kinds of exceptions generated by illegal instructions, bad operands, etc. Otherwise, the machine state might be corrupted and a crash could occur. Figure 4.1 shows our interactive graphic interface, which allows users to input Crashme arguments and displays output after the experiment completes.

According to our observations, Crashme could monopolize certain system resources and even crash the system. Table 4.1 summarized the major results of this experiment. In particular, Crashme crashed the HP 715 running HP-UX 9.05 and the SGI Indy running IRIX 5.3 numerous times within 10 minutes after testing started. A system crash also occurred on the DEC 3000 running OSF/1 V3.0 in deferred swapping mode. On the

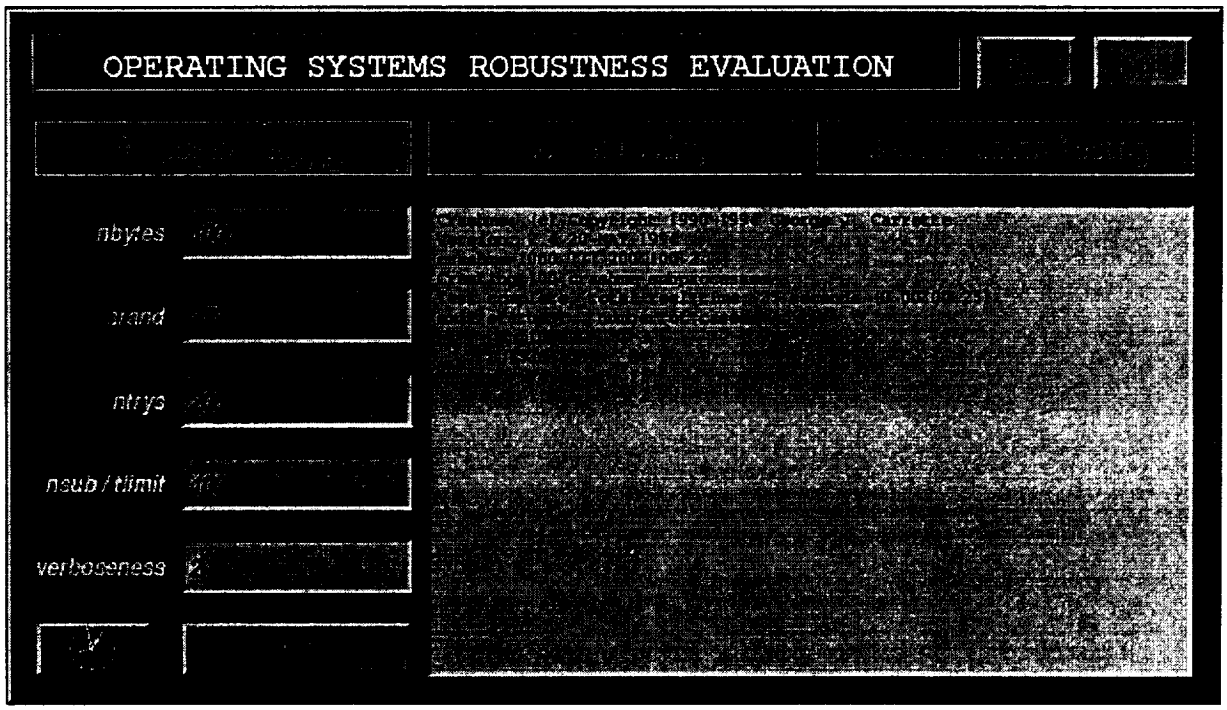

Figure 4.1: Graphical user interface of our Crashme experiment.

other hand, we did not experience any crashes in the HP 735 running HP-UX 10.0, nor in the SPARC 20 running Solaris 2.4. In addition to crashes, Crashme also succeeded in monopolizing certain system resources in the OSF/1 and Solaris systems.

4.1 Sequential Crashme Experiment

We invoked Crashme subprocesses sequentially on the DEC 3000 running OSF/1 V3.0, the HP 735 running HP-UX 10.0 and the SPARC 20 running Solaris 2.4 for 24 hours with the following arguments:

```
crashme +2000 111 200 24:00:00 2
```

Figure 4.2 shows how the arguments function during the sequential Crashme run, whose subprocesses were forked and executed in sequence. In particular, the above

Table 4.1: Possible Crashme-induced conditions on testing platforms

Observation	OSF/1 3.0 deferred	OSF/1 3.0 immediate	Solaris 2.3/2.4	HP-UX 9.05	HP-UX 10.0	SGI IRIX 5.3
Swap space was monopolized	√	√	√	crash occurred first		crash occurred first
Process slots were monopolized			swap space limit reached first	crash occurred first		crash occurred first
NFS was disrupted	√			√		√
System crashed (no NFS, no I/O)	√			√		√

Crashme instance generates 2,000 bytes of pseudo-random data and executes them as a sequence of instructions. A signal handler is set up to catch most of the machine exceptions generated by the illegal instructions, bad operands, etc. After this sequence of instructions is executed 200 times, the random seed is incremented to try another round of randomness. Eventually a random instruction may corrupt the program or the machine state so that the program must halt. System crash may occur in the middle of the experiment. Otherwise, the Crashme subprocesses will be terminated after 24 hours of running. Using a verbose level 2 will print out brief summary information.

4.1.1 Crashme-induced resource monopolization

For both Solaris and OSF/1 systems, Crashme monopolized the system swap space. Even though no additional processes could be started on the Solaris system, NFS continued to respond. According to our definition, Solaris system did not crash. As for

```
Usage : crashme <nbytes> <srand> <ntrys> [time] [verboseness]

        <nbytes> :  number of random bytes to be generated.
        <srand>  :  random seed.
        <ntrys>  :  number of iterations.
        [time]   :  duration for the experiment.
        [verboseness] :  verboseness level.

while ( experiment time < [time] ){
    generate <nbytes> of pseudo-random data using random seed <srand>;
    for i = 1 to <ntrys> {
        Execute the data sequence as an instruction stream and
                trigger exceptions continuously;
    }
    <srand> = <srand> + 1;
}
```

Figure 4.2: Sequential Crashme experiment.

the OSF/1 system, both immediate (guaranteed) and deferred (overcommit) swapping modes were tested. In the immediate swapping mode, Crashme consumed the entire swap space, and no new processes could be started. In the deferred swapping case, Crashme actually crashed the OSF/1 system. The details are described respectively as follows.

Sun Solaris Having Crashme run for more than 10 hours, the entire system swap space was consumed by the subprocesses spawned by Crashme. The system refused to take any more processes. No other users, including root, could function, and a reboot was necessary. However, NFS requests were still serviced promptly. Hence we did not consider a crash to have occurred.

When the specified time limit was reached, the background Crashme subprocesses were not properly cleaned up and continued to monopolize the memory. The fact the main process failed to kill its subprocesses might be due to lack of swap space.

The swap space monopolization by Crashme was very similar to the result of the memory monopolization experiment, which is described in Section 5.1. Vendors are reluctant to impose memory or swap space limits on individual processes and thus limit the flexibility of their systems. Rather, a vendor's response to such memory or swap space contention would probably be the suggestion to simply buy more memory or disk.

DEC OSF/1 Crashme experiment was performed twice on the OSF/1 system, once in the immediate swapping mode and the other in the deferred mode. In OSF/1, the two swapping modes operate as follows: if the immediate mode is used, swap space is allocated when modifiable virtual address space is created. If the deferred mode is used, swap space is not allocated until the system needs to write a modified virtual page to swap space.

For the immediate swapping case, Crashme was able to consume all the swap space after about 4 hours. However, NFS service was never disturbed. For the deferred swapping case, Crashme did interrupt NFS and crashed the system. We also observed the same problem of Crashme improperly terminating child processes on the OSF/1 system as on the Solaris system.

4.1.2 Crashme-induced system crashes

Crashme crashed the HP 715 running HP-UX 9.05 and the SGI Indy running IRIX 5.3 a number of times within 10 minutes with any of the following arbitrarily picked Crashme arguments.

```
crashme +2000 111 200 02:00:00 3

crashme +1000 777 200 02:00:00 3

crashme +1462 654 123 02:00:00 3
```

We also succeeded in crashing the HP 735 running HP-UX 9.05 within an hour. After the operating system on HP 735 was upgraded to HP-UX 10.0, however, Crashme could no longer recreate a system crash. The detailed results are described as follows.

SGI IRIX 5.3 Each of the above Crashme instances were run several times on the SGI Indy running IRIX 5.3. In less than 5 minutes, the Indy did not respond to interactive commands and stopped servicing NFS requests. Accordingly a crash was considered to have occurred. The system also failed to respond to "ping". A reboot was required. However, the system did not panic, hence no memory image was dumped.

The log file showed that the final Crashme subprocess arguments were `2000 511 200`, and we verified numerous times that invoking the following could crash the IRIX 5.3 system in one second.

```
crashme +2000 511 200 1 2
```

HP-UX 9.05 The same Crashme runs also crashed the HP 715 running HP-UX 9.05. In less than 10 minutes of running , the system panicked, dumped the memory image, and then self-rebooted. The HP 735 running the same version of HP-UX was also crashed in the same manner within an hour of running.

HP-UX 10.0 After the operating system on the HP 735 was upgraded to HP-UX 10.0, the experiment was repeated. No crashes occurred. The above test runs could not crash the system nor monopolize any system resources. However, many Crashme subprocesses were killed by the operating system due to a *stack growth failure.* Therefore, Crashme spent more time forking subprocesses, computing the random bytes and less time triggering exceptions. This explains why Crashme was less effective when running on HP-UX 10.0.

In order to better understand this particular phenomenon, another Crashme experiment was performed to generate all the subprocesses at once and let them run concurrently. Abnormal terminations of these subprocesses were also analyzed. The experiment and analysis are described in the following subsection.

4.2 Concurrent Crashme Experiment

As previously noted, HP-UX 10.0 killed many subprocesses in the sequential Crashme experiments. It was our concern that killing user processes to avoid system-wide impact might not be good enough for certain applications such as banking. In addition to crashes, it is also important to understand how each of the operating systems handled

Table 4.2: System configurations for concurrent Crashme experiment

	HP-UX 10.0	Solaris 2.4
`MAXUPROC` (max. num of user proc)	200	200
`MAXSSIZE` (max. user stack size)	8MB	8MB

the exceptions. Since SGI IRIX 5.3 and HP-UX 9.05 have already been shown highly vulnerable to Crashme, the following experiment was performed only on HP-UX 10.0 and Solaris 2.4. We did not show the results of this experiment on OSF/1 because Digital has added the ability to set user resource limits to Digital UNIX V4.0 (unavailable at our time of testing) which provides the ability to set a limit on the number of processes and the amount of memory that can be consumed by a single user.

4.2.1 Experiment description

As shown in Table 4.2, the systems configurations were modified such that both system platforms are comparable. In particular, the maximum number of processes per user, `MAXUPROC`, was raised to 200 on each of the operating systems. The maximum stack size, `MAXSSIZE`, was configured to be 8 MBytes on both systems.

Firstly, 100 concurrent Crashme subprocesses were generated by using the following arguments:

```
crashme +1000.4 777 200 100 2
```

As shown in Figure 4.3, the above Crashme instance generated all the subprocesses at once and let them run concurrently in the background. Crashme subprocesses terminated

```
Usage : crashme <nbytes> <srand> <ntrys> [nsub] [verboseness]

        <nbytes> :  number of random bytes to be generated.
        <srand>  :  random seed.
        <ntrys>  :  number of iterations.
        [nsub]   :  number of subprocesses to be forked.
        [verboseness] :  verboseness level.

for i = 1 to [nsub] {   /* Generate [nsub] concurrent subprocesses */
    fork a subprocess;
}

Each subprocess in parallel do {
    generate <nbytes> of pseudo-random data using random seed <srand>;
    for j = 1 to <ntrys> {
        Execute the data sequence as an instruction stream and
                trigger exceptions continuously;
    }
    <srand> = <srand> + 1;
}
```

Figure 4.3: Concurrent Crashme experiment.

either voluntarily through an `exit` system call or involuntarily as a result of a signal. In either case, an exit status was returned to the parent Crashme process through the `wait` system call [19]. By logging and analyzing the error messages and the exit status of subprocesses, we could measure how many subprocesses were killed by the operating system.

Table 4.3: Numbers of killed Crashme subprocesses out of 100

	HP-UX 10.0	Solaris 2.4
Number of killed subprocesses	3	0
Percentage	3%	0%

Table 4.4: Numbers of killed Crashme subprocesses out of 150

	HP-UX 10.0	Solaris 2.4
Number of killed subprocesses	5	0
Percentage	3.3%	0%

4.2.2 Results

Table 4.3 summarizes the major results in this experiment. In specific, 3 out of 100 Crashme subprocesses were killed by HP-UX 10.0 because of a stack growth failure. On the other hand, Solaris 2.4 did not kill any Crashme subprocesses in this experiment.

A similar experiment was performed on each platform to generate 150 subprocesses. The purpose was to see if the systems behaved differently with an increased number of subprocesses. Table 4.4 shows the result of the 150-subprocess experiment. The

Table 4.5: Exit status of 100 Crashme subprocesses

HP-UX 10.0		Solaris 2.4	
Exit status	Num of subproc	Exit status	Num of subproc
1536	8	1792	17
10	20	139	83
11	32		
4	40		

Table 4.6: Exit status of 150 Crashme subprocesses

HP-UX 10.0		Solaris 2.4	
Exit status	Num of subproc	Exit status	Num of subproc
1536	12	1792	27
10	29	139	123
11	53		
4	56		

Table 4.7: Causes of termination for Crashme subprocesses

HP-UX 10.0		Solaris 2.4	
Exit status	Cause	Exit status	Cause
1536	normal exit	1792	normal exit
10	bus error SIGBUS	139	seg. violation SIGSEGV
11	seg. violation SIGSEGV		
4	illegal instruction SIGILL		

percentages of killed subprocess were very close to the numbers shown in Table 4.3. No significant difference was observed between these two experiments in terms of the operating system's exceptions handling.

In addition, Table 4.5 and Table 4.6 summarize the exit status values of each of the Crashme subprocesses. The status value can be used to differentiate between normally exited processes and terminated processes. This is accomplished using the macros defined in `sys/wait.h` with the status value as an argument.

Table 4.7 summarizes the causes of termination of Crashme subprocesses. In particular, Solaris 2.4 had a normal process exit 17 times in the 100-subprocess experiment while HP-UX 10.0 had 8 times. In addition to normal exit, Crashme subprocesses might be terminated due to the receipt of signal which was not caught. For instance, 32 out of 100 subprocesses were terminated by signal `SIGSEGV` (segmentation violation) in HP-UX. Similar cases happened 83 times in Solaris.

5. SYSTEM RESOURCE MONOPOLIZATION

This chapter describes our attempts to have a single user monopolize the system resources including swap space, virtual memory, and process slots. The systems under test include the DEC 3000 running OSF/1 V3.0, the Sun SPARC 20 running Solaris 2.4, the Sun SPARC 2 running SunOS 4.1.3, the HP 715/64 running HP-UX 9.05, and the SGI Indy running IRIX 5.3 systems.

Figure 5.1 shows our interactive graphic interface, which allows users to either consume all available memory or available process slots. Each monopolization experiment is described in the following sections.

5.1 Swap Space Monopolization

Allocating all the available memory may leave the system with no swap space. The simple program given in Figure 5.2 is capable of taking up to 1024 MBytes of the system's virtual memory.

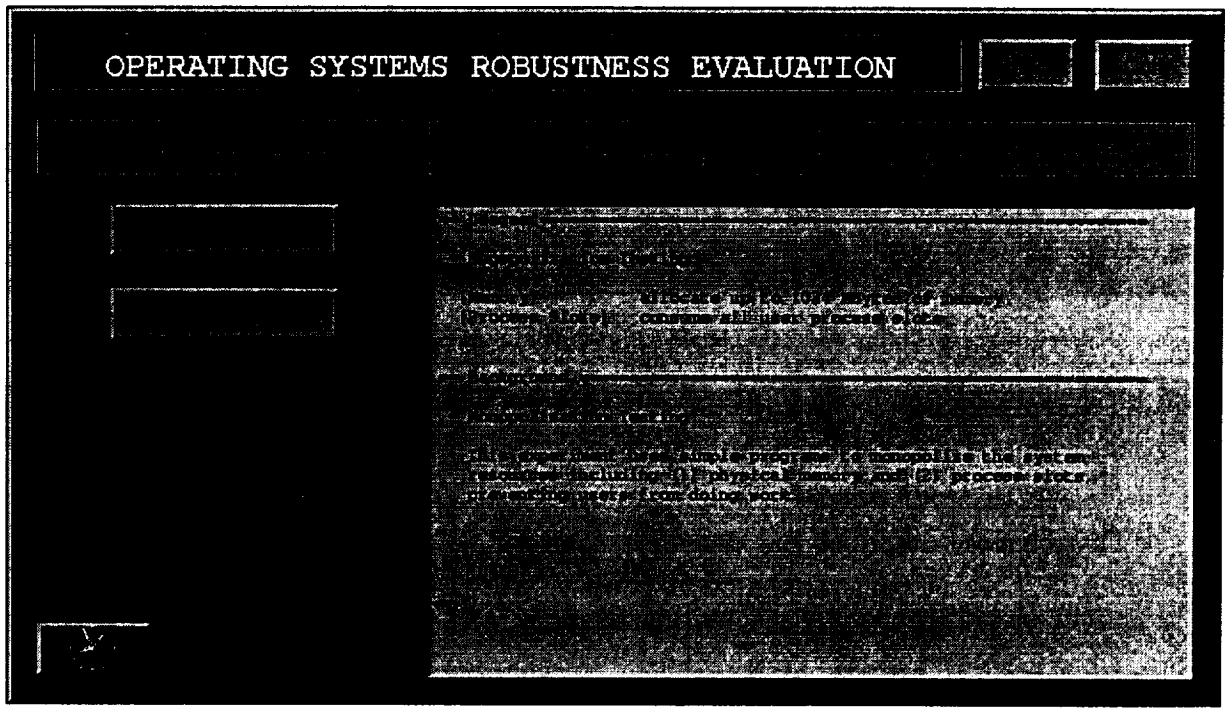

Figure 5.1: Graphical user interface of our monopolization experiment.

```
memSize  = 512*1024*1024;
do {
    buffer = (char *) malloc(memSize);
    memSize /=2;
} while (memSize>=1);
while (1)
  {       /* hold the memory and keep idle */
  }
```

Figure 5.2: Memory monopolizing program.

In OSF/1 V3.0, HP-UX 9.05, Solaris 2.3/2.4 ,SunOS 4.1.3 and IRIX 5.3, our program could allocate all the memory. After that, no new processes could be started in the system due to lack of swap space. A malicious use of this program can idle the whole system. Even a system administrator cannot kill this program because a "ps" to get the program's `pid` will also be blocked. However, vendors have generally chosen not to place a bound on the memory usage of a process in order to retain maximum flexibility, as discussed in Section 4.1.1.

5.2 Process Slots Monopolization

In addition to system swap space, a user can also monopolize process slots on some platforms. The following script was used to monopolize all the available process slots left in each of the system platforms.

```
#! /bin/csh
runme
runme
```

The shell script `runme` calls itself until the number of processes reaches the limit. In OSF/1, one user can run by default at most 64 processes concurrently. A monopolization of the process slots is not possible. In Solaris, however, one can run up to about 465 processes – the total number of process slots in the system. After that, the OS does not allow any user to start new processes. Like the memory monopolizing program, this simple script can potentially idle the whole system. The system recovered after we

user-interrupted (Ctrl-C) the program. Similar monopolizations were also observed in the SunOS and SGI IRIX system.

In the SunOS 4.1.3 source code file `conf.common/param.c`, the maximum number of processes per user, **MAXUPRC**, is defined as maximum number of processes allowed in the system minus five (**NPROC - 5**). This parameter is used to control forking. Since the system background jobs and daemons use more than 5 process slots, it is possible for a user to use the rest of process slots. In Solaris, a similar definition is likely used. For OSF/1, however, the default value of **MAXUPRC** is 64, and such monopolization is not possible without changing this parameter.

In HP-UX, a super user can run SAM (System Administration Manager) to modify **MAXUPROC** (maximum number of processes per user) in the kernel configuration. Otherwise a user cannot monopolize all the process slots in the normal situation.

5.3 Memory Swapping Experiment

In this experiment, we managed to create page faults and forced high memory swapping activity. A 100 MB array was allocated and randomly accessed. Each access of the array element not resident in current memory pages caused a page fault. Continuous accessing the array at random locations would force paging activity and lots of cache flushes.

In the deferred swapping mode of OSF/1, more than one such memory-swapping process could be started. Running more than two such processes concurrently could

cause a crash (no interactive response and no NFS response). In the immediate swapping mode of OSF/1 and other platforms, the experiment showed significant performance degradation, but no crash occurred.

Table 5.1: Resource monopolizing conditions observed

Observation	OSF/1 deferred	OSF/1 immediate	Solaris	SunOS	HP-UX	SGI IRIX
Memory monopolization experiment						
memory was monopolized	√	√	√	√	√	√
NFS was disrupted						
Process slot monopolization experiment						
process slots were monopolized			√	√		√
NFS was disrupted						
Memory swapping experiment						
NFS was disrupted	√					
System crashed	√					

5.4 Results

Table 5.1 summarizes the results of the resource limitation experiments and memory-swapping experiment under all operating systems. On OSF/1, both immediate and deferred modes for swap space allocation were tested. One single user can monopolize the memory in either mode. Although there was a warning in immediate mode when

the free swap space was below 10 percent, we were still able to allocate all the memory, blocking any new processes.[1]

[1]Digital has added the ability to set user resource limits to Digital UNIX V4.0 (unavailable at our time of testing) which provides the ability to set a limit on the number of processes and the amount of memory that can be consumed by a single user.

6. WORKLOAD STRESS TESTING

This chapter presents our evaluation of the system capability for the DEC 3000 running OSF/1 V2.1 and the Sun SPARC 20 running Solaris 2.3. We conducted a set of tests which stressed each system with a high workload. The tests used a synthesized workload which was composed of disk I/O, CPU, memory and network workload.

We refer to the disk I/O, CPU and memory portions collectively as the *local workload.* In this experiment, the response times to typical interactive commands were measured under seven different local workload and five different network workload. Moderate to severe performance degradations were observed on each system.

6.1 Local Workload

The local workload was produced by a *synthetic workload generator* [20], which allows its user to specify the desired workload. The workload was generated by calling one of three work functions: an I/O-intensive function, a memory-intensive function, or a CPU-intensive function. The sequence of workload functions was randomly chosen.

Table 6.1: Local workload description

LOAD	P[CPU]	P[MEM]	P[I/O]
1	0	0	1
2	0	1	0
3	1	0	0
4	0.33	0.33	0.33
5	0.2	0.2	0.6
6	0.2	0.6	0.2
7	0.6	0.2	0.2

P[FN] is the probability that function FN will be the next function chosen.

The frequency of each function type was specified before the workload started. Table 6.1 summarizes each of the local background workload probability distributions in the experiment.

6.2 Network Workload

The remote workload consisted of client machines each running a remote-net-stress script. To generate network activity, the script copied data from an NFS-mounted file system on the stressed machine. In this experiment, the remote workload was varied by stepping through the number of clients (0-4) requesting data.

6.3 Tests Performed

We repeatedly timed two typical interactive commands to obtain a performance measure for the target system. The locally timed command was a "grep" of 200 files (for

Table 6.2: Local response times (in seconds) on OSF/1

LOCAL WORKLOAD	NET0 0 clients	NET1 1 client	NET2 2 clients	NET3 3 clients	NET4 4 clients
1	34.72	38.11	38.79	41.79	49.31
2	14.27	18.26	23.15	29.49	39.79
3	13.46	16.95	21.22	28.87	32.00
4	23.54	22.65	25.53	32.29	40.91
5	31.59	31.80	30.28	35.50	42.85
6	20.38	22.42	25.72	32.84	42.50
7	18.98	20.38	24.57	32.43	40.27

See Table 6.1 for load explanation.

a total of 20 MB of data). The remotely timed command was an "ls" of a user's home directory mounted on a network file server's disk.

6.4 Results

The performance information collected is summarized in Tables 6.2 – 6.5. In particular, Tables 6.2 and 6.3 provide local and remote execution times for the DEC 3000 system. Tables 6.4 and 6.5 provide equivalent information for the Solaris SPARC 20 system. The enormous performance difference between the two systems is evident from the execution times in the tables.

As expected, the data shows that increasing the network workload generally raises execution time. There are some singularities, however, particularly in Table 6.5 (Load 1/NET 2, for example). The values that stand out could be the result of external users (we did not have an isolated system) using the network.

Table 6.3: Remote response times (in seconds) on OSF/1

LOCAL WORKLOAD	NET0 0 clients	NET1 1 client	NET2 2 clients	NET3 3 clients	NET4 4 clients
1	4.61	4.85	5.09	6.00	8.35
2	2.31	2.34	2.52	3.70	6.12
3	2.15	2.35	2.51	3.69	4.61
4	3.57	3.36	3.45	4.33	6.39
5	4.54	4.70	4.00	4.84	6.40
6	3.32	3.65	3.38	4.69	6.35
7	3.47	3.90	3.91	4.76	6.19

See Table 6.1 for load explanation.

Table 6.4: Local response times (in seconds) on Solaris

LOCAL WORKLOAD	NET0 0 clients	NET1 1 client	NET2 2 clients	NET3 3 clients	NET4 4 clients
1	71.52	67.86	74.02	79.14	79.86
2	134.03	154.40	160.45	168.45	167.60
3	134.10	149.50	151.03	155.25	158.55
4	133.67	142.10	150.17	158.80	162.30
5	133.83	122.60	142.03	149.40	161.85
6	134.27	150.20	155.07	159.60	171.10
7	132.70	141.53	148.10	150.85	162.75

See Table 6.1 for load explanation.

Table 6.5: Remote response times (in seconds) on Solaris

LOCAL WORKLOAD	NET0 0 clients	NET1 1 client	NET2 2 clients	NET3 3 clients	NET4 4 clients
1	4.90	11.01	9.36	11.42	22.21
2	7.57	8.69	9.16	10.64	11.49
3	6.96	7.96	7.94	9.09	9.85
4	7.31	8.93	9.01	10.04	11.04
5	7.73	9.70	10.08	9.33	10.29
6	7.72	8.82	9.19	9.98	10.92
7	7.53	9.06	8.24	12.95	10.43

See Table 6.1 for load explanation.

We observed that in the DEC 3000 system local as well as remote response times increased when the local workload was I/O bound. (14.27 seconds versus 34.72 seconds in Table 6.2, Load 2 versus Load 1, with zero clients). One possible reason for a CPU or memory workload's lowering of the response time was the DEC's available computation bandwidth. Grep might not utilize all of the available CPU cycles, and a CPU background workload could use the otherwise wasted CPU cycles. An I/O background workload, on the other hand, would compete with grep directly.

In contrast with the DEC 3000, the SPARC 20 tests showed response times *decreased* for an I/O bound local workload. An reason for this behavior discrepancy might be that grep on the SPARC system actually needed most of the CPU cycles. Competing CPU-intensive processes might delay grep more than competing I/O processes if grep did not get enough CPU cycles. Overall, we observed a much larger performance difference

between the SPARC and DEC systems for CPU-intensive background workload cases than for I/O-intensive background workload cases.

7. CONCLUSIONS

7.1 Summary

UNIX has been implemented on a wider range of machines than any other operating system. With the enhanced capabilities and complexity of today's UNIX systems, there is a need to clearly understand their reliability. In this study, we conducted three sets of experiments to evaluate the robustness of five UNIX-based operating systems, which included DEC OSF/1, HP-UX, Sun Solaris, SunOS, and SGI IRIX. These experiments included exception handling analysis, resource monopolization experiment, and workload stress testing.

An extensive set of Crashme runs have been performed on each of the system platforms under evaluation. We were able to crash the HP 715 running HP-UX 9.05 and the SGI Indy running IRIX 5.3 numerous times within 10 minutes after the testing started. We also experienced a crash of the DEC 3000 running OSF/1 V3.0 in the deferred swapping mode. No crashes occurred in the HP 735 running HP-UX 10.0, nor in the SPARC 20

running Solaris 2.4. In addition to crashes, we also observed that Crashme succeeded in monopolizing certain system resources in the DEC 3000 running OSF/1 V3.0 and the SPARC 20 running Solaris 2.4. It was also shown that Solaris 2.4 could handle a large number of concurrent Crashme subprocesses, while HP-UX 10.0 killed many subprocesses to keep the system alive.

Our resource monopolization experiments used programs designed to usurp system resources, preventing users from doing work. In OSF/1 V3.0, Solaris 2.4, SunOS 4.1.3, HP-UX 9.05, and IRIX 5.3, a single user could monopolize the swap space. We also observed that process slots could be monopolized in Solaris, SunOS, and IRIX. In all these cases, no more processes could be started in the system.

Finally, the workload stress testing in DEC OSF/1 V3.0 and Solaris 2.4 showed the execution times of typical interactive commands in OSF/1 V3.0 and Solaris 2.4 under various disk I/O, CPU, memory, and network workload mixes. Moderate to severe performance degradations were observed.

7.2 Future Work

Our results show that Crashme could crash an HP 715 running HP-UX 9.05, an SGI Indy running IRIX 5.3, and a DEC 3000 running OSF/1 V3.0 in deferred swapping mode. However, it is unclear what kind of exceptions actually crashed the systems. In order to identify the cause of a crash, we need to develop a diagnostic tool to examine the crash

panic message and the system's memory image. Some UNIX systems provide tools, such as *icrash* on SGI IRIX, to read the memory image file.

Usually an operating system handles exceptions as user processes execute in the system. By combining our exception handling experiment and workload stress testing, we are able to stress the system more. To implement this testing environment, we can generate disk I/O, CPU and memory workload activity in the background using a synthetic workload generator, and then apply exception handling test by running Crashme program. While the tested operating system is handling the synthetic workload and the exceptions triggered by Crashme, we measure the system's response time to local and remote interactive commands and compute the performance degradation. If the system crashes in the experiment, the panic message and memory image are then examined to identify the cause.

REFERENCES

[1] N. D. Singpurwalla, "The failure rate of software - does it exist," *IEEE Transactions on Reliability*, vol. 44, Sep. 1995.

[2] A. Bertolino and L. Stringini, "On the use of testability measures for dependability assessment," *IEEE Transactions on Software Engineering*, Feb. 1996.

[3] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical testing of software based on a usage model," *Software-Practice-&-Experience*, vol. 25, pp. 97–108, Jan. 1995.

[4] J. Ferdous, M. B. Uddin, and M. Pandey, "Reliability estimation with Weibull inter failure times," *Reliability-Engineering-&-System-Safety*, vol. 50, no. 3, pp. 285–296, 1995.

[5] G. Triantafyllos and S. Vassiliadis, "Software reliability models for computer implementations - an empirical study," *Software-Practice-&-Experience*, vol. 26, pp. 135–164, Feb. 1996.

[6] V. S. Rana, "Reliability modelling for some computer systems," *Microelectronics-and-Reliability*, vol. 34, pp. 93–106, Jan. 1994.

[7] M. H. Chen, A. P. Mathur, and V. J. Rego, "Effect of testing techniques on software reliability estimates obtained using a time-domain model," *IEEE Transactions on Reliability*, vol. 44, pp. 97–103, Mar. 1995.

[8] J. Miller, M. Roper, M. Wood, and A. Brooks, "Towards a benchmark for the evaluation of software testing techniques," *Information-and-Software-Technology*, vol. 37, pp. 5–13, Jan. 1995.

[9] A. Avritzer and E. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software," *IEEE Transactions on Software Engineering*, vol. 21, pp. 705–716, Sep. 1995.

[10] B. Murphy and T. Gent, "Measuring system and software reliability using an automated data collection process," *Quality-and-Reliability-Engineering-International*, vol. 11, pp. 341–353, Sep. 1995.

[11] A. Thakur, R. K. Iyer, L. Young, and I. Lee, "Analysis of failures in the Tandem NonStop-UX operating system," *IEEE Transactions on Reliability*, vol. 44, pp. 97–103, Mar. 1995.

[12] W. L. Kao, R. K. Iyer, and D. Tang, "FINE: A fault injection and monitoring evbironment for tracing the UNIX system behavior under faults," *IEEE Transactions on Software Engineering*, Nov. 1993.

[13] W. L. Kao, D. Tang, and R. K. Iyer, "Study of fault propagation using fault injection in the UNIX system," *IEEE Transactions on Software Engineering*, 1993.

[14] D. Andrews, "Vendors rally for 64-bit UNIX," *BYTE*, Nov. 1995.

[15] J. Forys, C. Rosa, and H. Ikeda, "UNIX on 64-bit architectures," *NEC Research & Development*, vol. 36, pp. 312–324, Apr. 1995.

[16] T. Yager, "The beat little file system," *BYTE*, Feb. 1995.

[17] J. Sontag, "HP-UX 10.0," *BYTE*, Apr. 1995.

[18] G. J. Carrette, "Crashme 2.4," 1994.

[19] S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The design and implementation of the 4.3BSD UNIX operating system.* Addison-Wesley publishing company, 1989.

[20] T. K. Tsai and R. K. Iyer, "An approach towards benchmarking of fault-tolerant commercial systems," to appear in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, 1996.