ICRASH Reference Guide

Document Number 108-0167-001

Contributors

Written by Greg Russell
Illustrated by Greg Russell
Edited by Cindi Leiser
Production by Cindy Stief
Engineering contributions by Tom Morano

Contents

Examples

Introduction to the ICRASH User's Guide

This document contains information about ICRASH, a crash analysis program that assists the support provider in debugging potential system problems.

The ICRASH manual is organized to guide the reader through a logical progression of topics, with the intention that the reader ultimately be able to effectively resolve system failures by employing the crash analysis support that ICRASH offers.

The reader audience includes engineers, analysts, support personnel, and field service personnel.

Document Contents

The list of topics covered in this document includes:

- An overview of ICRASH

  This Chapter contains a brief overview of the ICRASH utility with an emphasis on its major features and uses. It provides a basic framework for the material that follows. Some questions answered here are: Why was ICRASH designed? When is ICRASH used? What are the various modes of ICRASH operation? What are some of the differences among the various ICRASH releases? This Chapter also defines the basic terminology used within the document.

- An introduction to core dump analysis

  This chapter presents the basic concepts of the analysis process and discusses kernel internals, including threads and stacks. The discussion in this chapter is general in nature, but will provide anyone who is not familiar with core dump analysis with a complete picture of the analysis process, ranging from information gathering to the actual bug fix.

  This Chapter contains sufficient introductory information to prepare the reader for most of the subsequent material in this document. It does not present a comprehensive discourse of IRIX™ kernel internals. Instead, this Chapter focuses on the internal elements of the kernel that relate to system crashes and core dump analysis.

- An guide to ICRASH usage

  This Chapter outlines the various ICRASH modes of operation and reporting. It also presents some ICRASH commands, especially the trace commands.

- Advanced topics

  This Chapter covers data extraction from the stack frame, including procedure parameters, local variables, and saved registers. This material in this Chapter is in development.

- A guide to core dump analysis

  This chapter describes a methodology for analyzing IRIX system core dumps and describes each of the five stages of core dump analysis.

- A step-by-step examination of an actual core dump

  Examples follow the core dump analysis process through the five stage process.

- Appendix

  Appendix A contains tables of commands used in several versions of ICRASH. Each table lists the commands, indicates the man page where each command is described in detail, and briefly describes the function of each command.

Typographic Conventions

The following typographic conventions are used throughout this document:

Convention	Meaning
`TYPEWRITER FONT`	Denotes literal items such as command names, file names, routines, directory names, path names, signals, messages, and programming language structures.
italic font	Denotes variable entries and words or concepts being defined.
`bold typewriter font`	In screen drawings of interactive sessions, denotes literal items entered by the user. Output is shown in nonbold typewriter font.
[]	Indicates an optional item.
<>	Indicates a required variable within an optional item.
Enter	Enter means either to type a command or to select a menu command and then press the Enter key on your keyboard.

Within this document, reference is made to the online man pages available under IRIX through the `man` command. A *man page* is a discussion of a particular element of the IRIX operating system or a compatible product.

Each man page includes a general description of one or more commands, routines, or other topics and provides details of their usage (command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry will appear in the printed manual alphabetized only under its major name. You can access a man page named `ls` on-line by typing **`man ls`**.

Man pages are grouped into sections numbered from 1 to 8. Each section contains entries of a particular type. Types of entries include user commands (1), administrator commands (8), system calls (2), library routines (3), file formats (5), and device descriptions (4).

Section numbers appear in parentheses after man page names. Man pages are referenced in text by entry name and section number.

Chapter 1

Overview of ICRASH

This Chapter provides an overview of the IRIX™ Crash (ICRASH) core dump analysis tool. ICRASH is specifically designed to help kernel development engineers and support providers isolate the cause of a system failure.

1.1 What is ICRASH

ICRASH is a system crash analysis tool that can greatly enhance an engineer's ability to analyze IRIX system core dumps. It contains many features for displaying information in a clear, easy-to-read manner about the events that precede a system crash . ICRASH has two basic modes of operation:

- It can be used to generate an IRIX CORE FILE REPORT for any system core dump. This report contains selected pieces of information from the kernel that are considered most useful when you are trying to identifying the cause of the crash.
- It can be used to interactively (on a system core dump or on a live system) examine specific pieces of data in the kernel. ICRASH contains a wide range of commands that display kernel stack traces, kernel structures, and memory dumps, to name a few examples. The output from each ICRASH command can also be piped to IRIX commands such as `more`, `grep`, and `pg`. Online help provides information about each ICRASH command.

1.2 Who Uses ICRASH

Operating system developers and any support provider may use ICRASH.

1.3 When to Use ICRASH

There are a variety of scenarios where ICRASH is the tool of choice. ICRASH can be used:

- To see what type of problem caused a system to crash (by reviewing the ICRASH CORE FILE REPORT)
- To perform an in-depth analysis of a system core dump

- To examine kernel data on a live system
- To determine the hardware configuration of a system (valid only for Origin200™ and Origin2000™ systems at this time)

1.4 IRIX Releases Supported by ICRASH

Each version of ICRASH is specific to the operating system (OS) release of which it is a part and is not able to analyze system core dumps from other OS releases. There are versions of ICRASH available for IRIX 5.3, 6.1, 6.2, 6.3, and 6.4.

Chapter 2

Fundamentals of IRIX System Core Dumps

This chapter contains a brief overview of IRIX system core dumps. It describes what an IRIX system core dump is and how and why the core dump is generated. Throughout this discussion, a number of fundamental terms and expressions, used commonly throughout the remainder of this document, are introduced. Other terms that have more limited usage are addressed later in the document in the specific context of their usage. Following is a listing of the terms and expressions introduced in this chapter:

- IRIX kernel (kernel)
- program
- user mode
- kernel mode
- system crash (crash)
- application core dump
- system core dump
- dumping core
- system panic (panic)
- panic string
- NMI
- system hang
- hardware error
- software error
- thrashing

2.1 Overview of System Core Dumps

2.1.1 IRIX Kernel

In many ways, the IRIX kernel is just like any application, command, or utility program running as a process on the system. It is composed of machine instructions and performs specific tasks.

Programs running on the system usually perform tasks on behalf of the users. For example, the `ls` command displays a listing of disk files. The kernel, however, performs tasks on behalf of the system on which it is running. The kernel controls the basic system resources and maintains all the data structures that allow multiple processes (*programs*) to run.

2.1.2 Application Core Dumps

If a fatal error occurs while a process is executing application code or manipulating application data (running in user mode), the process dies and copies its memory image to a file on disk.

2.1.3 System Core Dumps

When a fatal system error occurs, the kernel stops executing (*panics*), sends a message (*a panic string*) to the system console, copies the system memory to the dump device (*system core dump*), and reboots the system.

2.1.4 Nonmaskable Interrupt (NMI) Core Dumps

It is possible to manually generate a system core dump without the benefit of a system panic. All high-end (CHALLENGE® L or CHALLENGE® XL servers, Onyx® workstations, Origin200 and Origin2000 systems) systems contain a special feature that enables system administrators to initiate system core dumps. They accomplish this by issuing a Nonmaskable Interrupt (NMI) request. Depending on the system, selecting a system controller menu option or pressing a special button on the system controller will initiate an NMI. A system administrator (often at the request of SGI support) normally induces an NMI core dump when users of a system complain that the system is partially or completely hung. The resulting system core dump may provide a clue about the cause of the problem.

2.1.5 Errors that Cause System Panics

There are a variety of errors that can cause a system panic. They all, however, fall into one of three basic categories:

- Hardware errors

  If a fatal error is detected in any of the basic hardware components of the system, the system panics with a hardware error message. If the error was not fatal, a warning message is sent to the system console. Generally, hardware-caused errors are clearly indicated in the panic string.

  Depending on the type of device and the nature of the failure, a hardware device driver may or may not force a panic if it detects a fault in the device it controls. For example, a read error in a disk partition may not justify bringing down the entire system unless the block contains kernel data. Instead, a warning message is displayed on the system console to notify the system administrator of the bad disk block.

- Software errors

  It is a common misconception that system crashes are usually caused by faulty hardware. That is, all that is required to correct a problem is to determine the failing component, replace it, and the problem will be solved. Unfortunately, the majority of system panics are attributed to software problems in the kernel or in a device driver.

  Software panics are usually caused by bugs in the kernel. For example, if an array in kernel memory is overwritten, a number of undesirable results may occur, depending on the location of the array in kernel virtual memory space and the contents of memory beyond the bounds of the array.

  If the array is located near the end of a mapped virtual page, such as a K2 memory page and the next virtual page is not mapped, a `KERNEL TRAP - TLBMISS` panic will occur. If the memory beyond the array contains valid kernel data, the data might be modified in such a way that it causes a panic the next time it is referenced. If the memory is allocated by the kernel memory allocator, the block header for the next block is likely to be corrupted, which causes a panic the next time an attempt is made to allocate that block. These examples are only a few of the possibilities.

  Note: Depending on the nature of the bug, it is quite possible for a single bug to cause the system to crash in a variety of ways.

- Resource Errors

  A system panic may occur when the system runs out of certain critical kernel resources, when certain resources cannot be allocated, or when certain daemon processes (for example, `init`) are killed.

  Note: A shortage of some kernel resources is also likely to result in a system hang.

2.1.6 System Hangs

A system hang is not a panic. There is a significant difference between the two events. During a panic event, the kernel or a CPU detects an error condition, dumps core, and shuts down the system. When a system hang occurs, the system does not panic; it continues to run. Response to input from the users, however, usually halts.

2.1.7 System Thrashing

System *thrashing* occurs when the kernel is performing such heavy repetitive activity (for example, swapping memory pages to disk) that no other processes are able to run and the system cannot accomplish any practical work. The system may appear to hang when it is actually thrashing.

2.2 System Crash and Recovery

ICRASH plays an integral part in the system crash and recovery process. It generates an ICRASH CORE FILE REPORT after each system core dump. An overview of the various kernel facilities, IRIX utilities, and scripts involved with a system crash and recovery follows.

2.2.1 System Core Dump

After a system panic or NMI occurs, the kernel systematically copies the contents of memory to the dump device (normally the primary swap partition). The memory pages are stored in a compressed form in the order of their importance to the core dump debugging process. Memory pages used by the kernel are saved first, followed by all of the memory pages allocated to user processes, and finally by the memory pages that are not in use.

2.2.2 `savecore` Startup Script

Every time a system starts up, the `savecore` startup script determines if there is a valid system core dump on the dump device. If a valid dump is found, several things occur:

- If the `savecore chkconfig` option is set to *on*, the `vmcore` image is copied from the dump device to the `/var/adm/crash` directory. A copy of the IRIX kernel that was running at the time of the panic (`/unix`) is also placed in the `/var/adm/crash` directory.
- The `savecore` script calls ICRASH to generate an ICRASH CORE FILE REPORT. If the savecore chkconfig option is set to *on*, then ICRASH generates the report from the files saved in `/var/adm/crash`. If this option is not on, ICRASH generates the report using `/unix` and the vmcore image on the dump device.
- On systems running IRIX 5.3, 6.1 and 6.2, the `savecore` script issues the `fru` command to generate a Field Replaceable Unit (FRU) Analysis report.

  Note: The `fru` command is relevant only to system core dumps created on CHALLENGE L and CHALLENGE XL servers, and on Onyx workstations, whose FRU Analyzer package is not part of the kernel. Origin systems running IRIX 6.4 have a FRU Analyzer that resides entirely within the kernel.

One or more of the following files are likely to be created as a result of actions taken by the `savecore` script:

- /var/adm/crash/analysis.*n*

  The core dump analysis contains such items as the `putbuf` dump, `fru` information, and stack traces. This analysis is a verbose description of what happened when the system crashed, and it is intended to provide a preliminary analysis of the system before you make any hardware or software changes.

- /var/adm/crash/summary.*n*

  The `summary` report contains the panic string, the crash time, and the FRU Analyzer information in one file. Availmon, a tool for gathering and reporting availability data and diagnostic data, reads this report file. For further information regarding the use of Availmon, refer to the *Availmon Reference Guide*, publication number 108-0169-001.

- /var.adm/crash/fru.*n*

  The output from the FRU Analyzer is placed in this file. It will be created only if a FRU analysis can be performed.

- /var/adm/crash/unix.*n*

  The IRIX kernel that was running at the time the system core dump was created.

- /var/adm/crash/vmcore.*n*.comp

  A copy of the `vmcore` image from the dump device.

The numeric extension, *n*, associated with these files increments with each system core dump. The contents of the `bounds` file, located in the `/var/adm/crash` directory, determines the next value to use.

The word *savecore* can be associated with several software elements in the system. Ambiguous use of this name can create confusion. The term `savecore` can refer to the:

- `savecore` utility
- `savecore` startup script
- `chkconfig` command option for `savecore`

2.2.3 FRU Analyzer

The FRU Analyzer provides the SSE with a starting point in the resolution of a system malfunction. When the operating system determines that a fatal hardware error has occurred, the hardware state is collected and passed to the FRU Analyzer for interpretation. This process does not require user intervention.

The FRU Analyzer applies a set of rules to the hardware state and from those rules attempts to determine the cause of the fatal error. The conclusion may be that a particular piece of hardware (for example, a memory bank) has failed, that software has caused the crash, or that the hardware state provides no conclusive evidence. Generally, FRU Analyzer reports that a FRU (field replaceable unit) has failed, and in certain circumstances the error can be traced to a more specific part of a FRU. In those cases, both the FRU and the detailed conclusion are reported in an analysis summary report.

For more information regarding the use of FRU Analyzer, refer to the *FRU Analyzer User's Guide*, publication number 108-0166-001.

2.2.4 Availmon

The Availability Monitor (Availmon) utility is available as a patch for IRIX 5.3, and is included in base IRIX release for 6.1 and beyond. Availmon, together with ICRASH and the FRU Analyzer, provides a technology platform for system availability and diagnostic data gathering and distribution.

Availmon is embedded in the system boot and shutdown processes. It differentiates controlled shutdowns, system panics, system hangs, power cycles, and power failures. On high-end systems such as the IP19, IP21, IP25, and IP27, it can further differentiate NMIs, power cycles, and power failures. A light-weight daemon can be used to track uptime, and send status reports. Diagnostic information is collected from ICRASH, the FRU Analyzer, SYSLOG, `hinv`, `versions`, and `gfxinfo`.

All availability data and diagnostic data for participating systems are archived in a central SGI database. Access to that database can provide overall reliability data and specific problem history for individual systems.

All aspects of Availmon operation are fully configurable.

After installation, you must register your system with the SGI Availmon Database by executing `/usr/etc/amregister -r` (for IP19, IP21, IP22, IP25, and IP27 systems; other systems require a serial number entry, see the man page `amregister(1M)`).

For more information regarding the use of Availmon, please refer to the *Availmon Reference Guide*, publication number 108-0169-001. Availmon help is also available on the World Wide Web at:

http://infohub.engr.sgi.com/availmon

2.3 Kernel Internals

2.3.1 Kernel Architecture

This manual does not attempt to provide a complete IRIX kernel tutorial. This subsection however, does attempt to provide a general understanding of the IRIX kernel structure as it relates to ICRASH.

[This subsection is still under development.]

2.3.2 Kernel Data Structures

A number of structures in the kernel are of particular interest during system core dump analysis. Many of the ICRASH interactive commands dump the contents of various structures, which provides an insight to the activity of the kernel when a panic occurred.

2.3.2.1 Processes and Kernel Threads

In IRIX revision 6.1 and earlier versions of IRIX, the process was the central mechanism for distributing processor resources over a collection of independent and cooperating tasks in the operating system. IRIX 6.2 introduced the migration toward the use of kernel threads (*kthreads*) as a central mechanism.

Kernel threads resemble the execution model of a UNIX process and consist of a code stream, private stack, and private register space. Unlike UNIX processes, kthreads are inexpensive to create and destroy, and can be quickly scheduled. A kthread has an associated user process context only if it is running on behalf of a system call or page fault, otherwise it has no logical connection to a user process.

With IRIX 6.2 and 6.3, only a partial conversion was made. The construct of a kthread was introduced, however it was still closely associated with entries in the proc table. In fact, a proc table entry was allocated for each active kthread in the system (even those that had no process context). Not until IRIX 6.4 was the conversion more or less complete (the evolution will continue with future revisions of IRIX). The kthread has now become the fundamental execution entity in the system. There are three types of kthreads:

- User process
- Interrupt thread (*ithread*)
- Service thread (*sthread*)

As you will see in subsequent sections of this manual, there are a number of ICRASH commands that focus on processes/kthreads.

2.3.2.2 CPU Private Data Area

A Private Data Area (*pda*) is an area in kernel memory, which is set aside for each CPU installed in the system. The pda contains information that is crucial for the proper operation of a CPU. The pda_s structure contains useful information such as the current state of the CPU, a pointer to the currently running process/kthread, pointers to the interrupt and boot stacks, and so forth.

2.3.3 Stacks

2.3.3.1 Application Stacks

Each application running on the system allocates a block of memory for use as a *stack*. The stack contains information about where a procedure was called from and is the place where procedure arguments, local variables, and the contents of saved general and floating-point

registers are stored. The stack is made up of a series of activation levels or *frames*, which are created each time a nonleaf procedure call is made. (Nonleaf procedures are procedures that make calls to other procedures.) Activation levels can also consist of *blocks* that define local variables within a procedure. In most cases, the stack is located at the top of the application's virtual address space and grows down, frame by frame, as each new procedure call is made or program block is entered. The last activation level for the application is always its main program block.

2.3.3.2 The kernelstack

When an application switches from user mode to kernel mode (that is, when a system call is made or when there is a page fault), the kernelstack is used to store the kernel function parameters, save registers, and so forth. A separate stack with a kernel virtual address is required when operating in kernel mode. When the kernel completes executing, control switches back to the application (and the application stack).

The kernelstack contains information about what the kernel is doing on behalf of a particular process. This information can help to determine the cause of a system panic or hang. The kernelstack virtual address is the same for all processes running on the system. For example, on an IP27 system, the address of the kernelstack is `0xfffffffffffff800`. The kernelstack address is platform specific and is determined when the kernel is built. Information is contained in the proc struct about the mapping of the kernelstack address to a particular physical memory page.

2.3.3.3 Kthread stack

The concept of a kthread was introduced with IRIX 6.2 (see section 2.3.2.1 above). The kthread structure contains a pointer to the stack used by the kernel when it is active. If the kthread is a user process, then the address of the kernelstack is the same as the kthread stack. If the kthread is an ithread or an sthread (IRIX 6.4 or later), then the stack address will be a kernel address. The size of the stack is also contained in the kthread.

2.3.3.4 CPU Stacks

Each CPU has an interrupt and bootstack (idlestack) permanently allocated to it. Pointers to these stacks are located in the `pda_s` structure of each CPU. These stacks are used by the kernel when no kthreads are active.

2.3.3.5 Active Stack

Each CPU must always have an active stack where it can store function parameters, temporary data, and save registers, whether a kthread/process is running on the CPU or when the CPU is sitting idle. The active stack can be one of the following:

- USER (application stack)
- kthread (kernelstack with the IRIX 5.3 and 6.1 versions of ICRASH)
- interrupt
- idle/boot

2.3.4 Stack Frames

From a kernel debugging point of view, the stack frame contains some of the most useful information. It's where function parameters are stored (hopefully), register values are saved, and temporary data storage is allocated from. Using ICRASH in conjunction with the kernel source code, it is possible to reconstruct exactly how a system panic occurred. In general, each stack frame consists of storage space for:

- Procedure call arguments

  If the current routine calls any procedure that requires parameters, space to store the parameters (if needed) will be reserved in the stack frame. The way that parameters are stored with IRIX 6.4 is different than with previous OS releases. Previously, it was the responsibility of the calling routine to provide space in its stack frame for all parameters passed. With 6.4, it is the responsibility of the called routine to provide space for any parameter that needs to be saved. The calling routine needs to reserve space only for those argument bytes that exceed the first eight argument words passed.

- Local variables

  Space for all local variables is allocated in the stack frame.

- Saved general and floating-point registers

  The values contained in certain general and floating-point registers must be maintained across procedure calls. If any of these registers are used in a procedure, their original value must be saved in the stack frame before they can be used. This saved value must then be restored to the register before a jump to the return address. For non-leaf procedures, the return address (ra or $31) must also be saved.

2.3.4.1 Stack Frame Organization

The following illustration of stack frame organization is exactly the opposite of stack frame organization in the MIPSPro Assembly Language Programmer's guide. In Figure 2-1, high memory is placed at the BOTTOM of the frame and low memory is placed at the TOP of the frame. Representing the stack frame upside down more closely conforms to the way in which stacks and stack frames are displayed in ICRASH (and `dbx`).

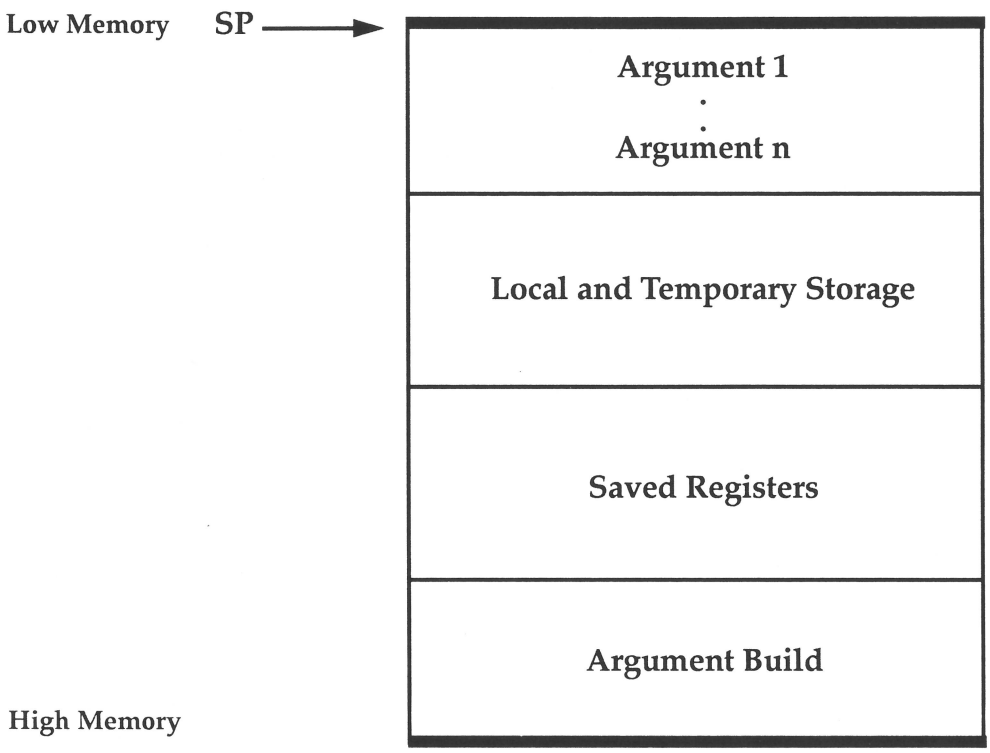

Figure 2-1 Organization of Stack Frames

2.3.4.2 Locating Useful Information in the Stack Frame

As explained in the previous subsection, the stack frame is divided into sections that hold procedure call arguments, local variables, and saved general and floating-point registers. This information can be very useful when you are trying to identify the cause of a system panic or hang. The procedures for extracting useful information from the stack frame are documented in Chapter 4, "Advanced Topics."

Chapter 3

Using ICRASH

This Chapter outlines the various ICRASH modes of operation and reporting. It also presents some ICRASH commands, especially the trace commands.

3.1 ICRASH Modes of Operation

3.1.1 Noninteractive Core Dump Analysis

3.1.1.1 The ICRASH CORE FILE REPORT

One of the primary features of ICRASH is its ability to generate an ICRASH CORE FILE REPORT. This report is automatically generated after every system core dump. It contains enough detailed information about a system crash so that a kernel-knowledgeable engineer can usually determine what went wrong and perhaps why. The ICRASH CORE FILE REPORT can also be generated manually by using the `-r` command line option or by issuing the interactive report command (available only with the IRIX 6.4 version of ICRASH).

The `savecore` startup script invokes ICRASH each time the system reboots after a system core dump. It uses the `unix` and `vmcore` image files, which are created by `savecore`, to generate an ICRASH CORE FILE REPORT. If the `savecore` option has been disabled by `chkconfig`, the configuration state checker (preventing `savecore` from being invoked at startup), ICRASH uses the running kernel (`/unix`) and the `vmcore` image stored on the dump device to generate the report. If `savecore` receives an error while creating the `unix` and `vmcore` image files, no report will be generated. The ICRASH CORE FILE REPORT is written to a file on disk in the same directory as the `unix` and `vmcore` images from which the report was generated. By default, the full pathname of the ICRASH report file is `/var/adm/crash/analysis.`*n*, where *n* is the same numeric file extension given to `unix` and `vmcore` image files by `savecore`.

The ICRASH CORE FILE REPORT provides the following information:

- General system information
- Type of crash
- Dump of the putbuf

- CPU summary
- Stack trace leading up to the panic if the dump was not initiated by an NMI
- Disassembly of instructions before and after the instruction that caused the crash if the dump was not initiated by an NMI
- The name of the active process running on each CPU (if there were any) and a stack trace for each CPU
- List of sleeping processes with wait channels if the dump was initiated by an NMI

3.1.1.2 Custom Reports Using the -r Option

To generate a custom ICRASH report from the IRIX command line, provide a command file parameter after the `-r` flag. This file should contain valid ICRASH commands. Use the `-w` flag to write the custom report output to a file on disk.

3.1.1.3 Executing ICRASH Commands form the IRIX Command Line

The `-e` command line option allows one or more ICRASH commands to be included on the IRIX command line. A command must be included inside of single or double quotation marks except when there are no arguments. Multiple commands can also be included, as long as they are separated by semicolons.

Note: This feature is available only with the IRIX 6.4 version of ICRASH

3.1.2 Interactive Core Dump Analysis

The ICRASH CORE FILE REPORT contains very specific information about a system crash. In some cases, there is enough information in the report to identify the failure that caused the crash. In most cases, however, you must examine the dump further to determine the cause.

A large number of ICRASH features are specifically designed for interactive system core dump analysis:

- ICRASH has a command-driven ASCII user interface. Command line editing and history facilities like those found in the `tcsh` shell (an enhanced version of the Berkeley UNIX C shell) have been included to provide more flexibility.
- Output from all ICRASH commands can be piped to any UNIX utility (`more`, `grep`, `pg`, and so forth) or written to a file on disk. Output from an entire ICRASH session can also be redirected to a file on disk with the `-w` command line option.
- An online help facility provides information about each ICRASH command. Help output includes a list of all valid command line arguments, followed by a list of any possible parameters, and a brief description of the command and its use.
- ICRASH can load commands from a file and run them as if they were entered from a keyboard. This feature can either be initiated at startup with the `-f` command line option or be initiated as an ICRASH command.
- ICRASH can execute commands on the command line with the `-e` option.
- ICRASH can suppress all output except command output (used in conjunction with the `-e` option)

3.1.3 Interactive Analysis of a Live System

ICRASH can also analyze a live system in much the same way that a system core dump is analyzed. You can easily obtain information about active processes and system resources with ICRASH functions. Some of the information ICRASH can extract from a live system includes:

- A stack trace for any process that appears to be hung (to see where in the kernel the process is sleeping)
- A list of open kernel files for a particular process
- The amount of virtual and physical memory a process has allocated to it
- CPU usage information

3.2 Starting the ICRASH Utility

Start the ICRASH utility by entering the following command at the IRIX command line.

```
# icrash unix.n vmcore.n.comp
```

Where `n` is the numeric extension for a `unix` and `vmcore` image files created by `savecore`. The following message is displayed while ICRASH is initializing:

```
Please wait.....................................................
```

followed by two greater than prompts.

```
>>
```

Alternately, use the `-n` command line option

```
# icrash -n 0
```

3.3 ICRASH Commands

3.3.1 ICRASH Command Interface

[This section is under development.]

3.3.2 Stack Trace Commands

ICRASH contains a number of commands that are specifically designed to assist in the process of displaying kernel stack traces. A *stack trace* is an ordered listing of the kernel function calls that have been issued up to the moment when a panic occurred or when the process/kthread went to sleep.

With access to kernel source code, you may actually "walk" (sequentially examine) the stack trace, routine by routine, reconstructing exactly how a panic occurred. With a little effort, it is even possible to determine the values assigned to the function parameters and local variables (refer to Chapter 4).

Example 3-1 Example stack trace generated by ICRASH

```
STACK TRACE FOR PROCESS 0xa8000000061fd000 (sendmail, PID=635):

1 swtch[../os/swtch.c: 546, 0xa80000000034e99c]
2 mon_trace_switch[../os/ksync/mutex.c: 140, 0xa8000000002bf56c]
3 sv_queue[../os/ksync/mutex.c: 1590, 0xa8000000002c1980]
4 sv_timedwait_sig[../os/ksync/mutex.c: 2071, 0xa8000000002c26ac]
5 sv_wait_sig[../os/ksync/mutex.c: 1461, 0xa8000000002c1508]
6 uipc_accept[../bsd/socket/uipc_socket1.c: 90, 0xa80000000027c888]
7 laccept[../bsd/vsock/lsock.c: 165, 0xa80000000027c3c0]
8 accept[../bsd/socket/uipc_syscalls.c: 174, 0xa800000000268904]
9 syscall[../os/trap.c: 2515, 0xa8000000002cc6c8]
10 systrap[../ml/LOCORE/systrap.s: 421, 0xa8000000001464b0]
       r0/zero:0000000000000000  r1/at:000000000fb43b80  r2/v0:0000000000000441
         r3/v1:0000000000000000  r4/a0:0000000000000005  r5/a1:0000000010116c88
         r6/a2:000000007fff10d8  r7/a3:0000000000000001  r8/a4:000000000fb43b80
         r9/a5:000000000fb43b80 r10/a6:0000000000000001 r11/a7:0000000000000000
        r12/t0:000000000000ff00 r13/t1:0000000000000001 r14/t2:0000000000046481
        r15/t3:000000000000ff00 r16/s0:000000000fb43b80 r17/s1:ffffffffffffffff
        r18/s2:000000007fff10d8 r19/s3:0000000000000004 r20/s4:000000001010f35c
        r21/s5:0000000010116c88 r22/s6:0000000000000010 r23/s7:000000007fff1094
        r24/t8:0000000010015310 r25/t9:000000000fa5b6d4 r26/k0:0000000000000000
        r27/k1:0000000000000020 r28/gp:000000000101186f0 r29/sp:000000007fff1040
        r30/s8:0000000000000000 r31/ra:0000000010015310    EPC:000000000fa5b6dc
            CAUSE=ffffffff90000004, SR=4ffb3, BADVADDR=1010f2e4
```

Each line in the stack trace contains the following information:

- The stack frame level number (the lowest number represents the most recent function call)
- The name of the kernel routine being called
- The source module the function is contained in, along with the line number containing that instruction
- The virtual address of the current instruction (which should be a jump to the next routine in the stack)

Stack traces, such as in the preceding example following line 10, sometimes contain one or more exception frames (*eframes* or register dumps). The eframe contains the exact state of the CPU registers at the time a kernel trap occurs. Values contained in the various registers may be useful in the process of reconstructing the failing sequence.

3.3.2.1 trace Command

The trace command displays a stack trace for each kthread included on the command line. If no kthreads are specified and defkthread is set, then a stack trace for the default kthread is displayed. If defkthread is not set, trace determines whether a process pointer has been stored in dumpproc. If there is a dumpproc, then a trace is displayed for it. If there is no dumpproc, then a trace for the CPU that generated the core dump is displayed.

Example 3-2 trace command output

```
>> trace a800000000fa2d00
============================================================================
Stack Trace for sthread 0xa800000000fa2d00d (sched):

1 istswitch[../os/swtch.c: 903, oxc000000000644efc]
2 swtch[../os/swtch.c: 431, 0xc000000000643730]
3 mon_trace_switch[../os/ksync/mutex.c: 147, 0xc000000000532a5c]
4 sv_queue[../os/ksynch/mutex.c: 1613, 0xc000000000537758]
5 sv_timedwait[../os/ksynch/mutex.c: 2078, 0xc000000000538e58]
6 sv_wait[../os/ksynch/mutex.c: 1469, 0xc000000000536ecc]
7 sched[../os/sched.c: 168, 0xc00000000063a870]
8 sthread_launch[../IP27DEBUGbootarea/process.s: 2913, 0xc000000000301278]
============================================================================
```

3.3.2.2 ctrace Command

The `ctrace` command displays a stack trace for each CPU included on the command line. If no CPUs are indicated then a stack trace for all CPUs is displayed.

Example 3-3 ctrace command output

```
>> ctrace 0
============================================================================
Stack Trace for CPU0

1 syncreboot[../os/printf.c: 1091, 0xc000000000602348]
2 icmn_err[../os/printf.c: 284, 0xc0000000006001c0]
3 panic[../os/printf.c: 462, 0xc0000000006008b4]
4 syssgi[../os/syssgi.c: 591, 0xc00000000064c374]
5 syscall[../os/trap.c: 2515, 0xc00000000054fee4]
6 systrap[../IP27DEBUGbootarea/systrap.s: 2677, 0xc0000000003256ac]
      r0/zero:0000000000000000  r1/at:0000000010001034  r2/v0:0000000000000410
        r3/v1:0000000000000000  r4/a0:00000000000003ed  r5/a1:000000007fff2f84
        r6/a2:000000007fff2f8c  r7/a3:000000007fff2f8c  r8/a4:000000000fb603a4
        r9/a5:0000000000000000 r10/a6:0000000000000000 r11/a7:00000000004002d8
       r12/t0:000000000000ff00 r13/t1:0000000000000080 r14/t2:00000000000068a1
       r15/t3:000000000000ff00 r16/s0:0000000000000001 r17/s1:000000007fff2f84
       r18/s2:000000007fff2f8c r19/s3:000000007fff2fc8 r20/s4:0000000000000000
       r21/s5:0000000000000000 r22/s6:0000000000000000 r23/s7:0000000000000000
       r24/t8:000000000fbc80b8 r25/t9:000000000fa5242c r26/k0:0000000000000000
       r27/k1:0000000000000020 r28/gp:0000000010008ff0 r29/sp:000000007fff2f40
       r30/s8:0000000000000000 r31/ra:0000000000400974    EPC:000000000fa46e40
           CAUSE=20, SR=ffffff8000ffb3, BADVADDR=fa5242c
============================================================================
```

3.3.2.3 etrace command

The `etrace` command displays a stack trace using the Program Counter (PC) and Stack Pointer (SP) found in the exception frame pointed contained on the command line. The stack address is determined using the stack pointer from the exception frame.

Example 3-4 etrace command output

```
>> etrace 0xffffffffffffba60
================================================================
EXCEPTION FRAME FOR 0xffffffffffffba60:

     r0/zero:0000000000000000   r1/at:0000000000000001   r2/v0:0000000000000004
       r3/v1:0000000000000001   r4/a0:000000007fff0ca0   r5/a1:0000000000000358
       r6/a2:0000000000000001   r7/a3:0000000000000000   r8/a4:0000000000000358
       r9/a5:0000000000000000  r10/a6:a8000006004726c0  r11/a7:0000000010009760
      r12/t0:a800000600471800  r13/t1:0000000000000358  r14/t2:a800000600471800
      r15/t3:a8000006004726c0  r16/s0:000000007fff0ca0  r17/s1:000000007fff0ca0
      r18/s2:0000000000000000  r19/s3:000000000000000e  r20/s4:000000000000000e
      r21/s5:0000000000000000  r22/s6:0000000000000001  r23/s7:0000000000002000
      r24/t8:0000000000000002  r25/t9:0000000000000004  r26/k0:c000000000116954
      r27/k1:0000000000000020  r28/gp:c000000001334ef0  r29/sp:fffffffffffbbe0
      r30/s8:a800000600471800  r31/ra:c0000000001a3418     EPC:c000000000034220
          CAUSE=28, SR=ffa3, BADVADDR=7fff0ca0
 1 fubyte[../ml/usercopy.s: 844, 0xc000000000034220]
 2 fault_in[../os/probe.c: 537, 0xc0000000001a3410]
 3 fast_useracc[../os/probe.c: 156, 0xc0000000001a2db8]
 4 useracc[../os/probe.c: 94, 0xc0000000001a2c78]
 5 sendsig[../os/machdep.c: 449, 0xc000000000176710]
 6 psig[../os/sig.c: 1923, 0xc00000000012a420]
 7 postsysactions[../os/trap.c: 2211, 0xc000000000123628]
 8 syscall[../os/trap.c: 2574, 0xc000000000123b60]
 9 systrap[../ml/LOCORE/systrap.s: 419, 0xc0000000000396a0]
     r0/zero:0000000000000000   r1/at:0000000000000004   r2/v0:0000000000000004
       r3/v1:0000000000000004   r4/a0:0000000000000005   r5/a1:00000000101145e0
       r6/a2:000000007fff1080   r7/a3:0000000000000001   r8/a4:ffffffffffffffff
       r9/a5:000000000000002e  r10/a6:000000000faf3cc0  r11/a7:0000000000000000
      r12/t0:000000000fb4b350  r13/t1:000000000fb4db40  r14/t2:000000000fb55bc0
      r15/t3:000000003fff0000  r16/s0:ffffffffffffffff  r17/s1:0000000001010d1cc
      r18/s2:0000000000000000  r19/s3:00000000101145e0  r20/s4:0000000000000010
      r21/s5:000000007fff107f  r22/s6:0000000000800000  r23/s7:0000000000000001
      r24/t8:000000001000d0a0  r25/t9:000000000fae0460  r26/k0:0000000000000000
      r27/k1:0000000000000020  r28/gp:0000000010116054  r29/sp:000000007fff1000
      r30/s8:0000000000000070  r31/ra:000000001000d0a0     EPC:000000000fae15a8
          CAUSE=8000, SR=ffffffff8400ffb3, BADVADDR=fa45168
================================================================
```

3.3.2.4 ktrace Command

The `ktrace` command displays a stack trace for each kthread pointer included on the command line.

Example 3-5 ktrace command output

```
>> ktrace a800000203c8c000
================================================================
STACK TRACE FOR PROCESS 0xa800000203c8c000 (icrash, PID=889):

 1 swtch[../os/swtch.c: 577, 0xc0000000001a5cc4]
 2 mon_trace_switch[../os/ksync/mutex.c: 149, 0xc000000000114ab8]
 3 semawait[../os/ksync/sema.c: 583, 0xc000000000112e20]
 4 sleep[../os/slp.c: 82, 0xc000000000198898]
 5 strwaitsleep[../io/streams/strsubr.c: 1358, 0xc0000000001043ac]
 6 strwaitq[../io/streams/strsubr.c: 1508, 0xc000000000104780]
 7 strread[../io/streams/streamio.c: 1010, 0xc0000000000f57b4]
 8 spec_read[../fs/specfs/specvnops.c: 573, 0xc000000000220578]
 9 _read[../os/vncalls.c: 419, 0xc00000000016e000]
10 read[../os/vncalls.c: 458, 0xc00000000016e174]
11 syscall[../os/trap.c: 2532, 0xc000000000123ab8]
12 systrap[../ml/LOCORE/systrap.s: 419, 0xc0000000000396a0]
     r0/zero:0000000000000000   r1/at:0000000000000001   r2/v0:00000000000003eb
       r3/v1:000000000007650c   r4/a0:0000000000000000   r5/a1:000000007fff18b0
       r6/a2:0000000000000001   r7/a3:0000000000000000   r8/a4:0000000000000003
       r9/a5:0000000000000003  r10/a6:000000000fb521d5  r11/a7:000000000fb521d2
      r12/t0:000000000fb531d2  r13/t1:0000000000000010  r14/t2:00000000100ea7d0
      r15/t3:000000003fff0000  r16/s0:0000000000000001  r17/s1:000000007fff2f74
      r18/s2:000000007fff2f7c  r19/s3:000000007fff2fc4  r20/s4:0000000000000000
      r21/s5:0000000000000000  r22/s6:0000000000000000  r23/s7:0000000000000000
      r24/t8:c00000000016e628  r25/t9:000000000faa47e4  r26/k0:0000000000000000
      r27/k1:0000000000000020  r28/gp:0000000010105b18  r29/sp:000000007fff18b0
      r30/s8:0000000000000000  r31/ra:000000001009d644     EPC:000000000fae0e38
          CAUSE=8, SR=ffffffff8400ffb3, BADVADDR=100ea7d0
================================================================
```

3.3.2.5 strace Command

The `strace` command can be used to generate arbitrary stack traces. Several ways that the `strace` command can be used are:

- To display all complete and unique stack traces from a given stack
- To reconstruct a particular stack trace using a PC, SP and stack page address provided on the command line
- To generate a listing of all valid kernel code addresses that are contained in a given stack, along with their location in the stack, source file, and line number

Example 3-6 strace command output

```
>> strace c0000000001759e8 ffffffffffffb780 ffffffffffff8000
============================================================
 1 syncreboot[../os/printf.c: 1112, 0xc0000000001759e8]
 2 icmn_err[../os/printf.c: 299, 0xc0000000001743c4]
 3 cmn_err[../os/printf.c: 109, 0xc000000000173c68]
 4 panicregs[../os/trap.c: 233, 0xc000000000121530]
 5 k_trap[../os/trap.c: 512, 0xc0000000001215ac]
 6 trap[../os/trap.c: 615, 0xc000000000121aa0]
 7 VEC_trap[../ml/LOCORE/vec_trap.s: 61, 0xc000000000039b8c]
     r0/zero:0000000000000000   r1/at:0000000000000001   r2/v0:0000000000000004
       r3/v1:0000000000000001   r4/a0:000000007fff0ca0   r5/a1:0000000000000358
       r6/a2:0000000000000001   r7/a3:0000000000000000   r8/a4:0000000000000358
       r9/a5:0000000000000000  r10/a6:a8000006004726c0  r11/a7:0000000010009760
      r12/t0:a800000600471800  r13/t1:0000000000000358  r14/t2:a800000600471800
      r15/t3:a8000006004726c0  r16/s0:000000007fff0ca0  r17/s1:000000007fff0ca0
      r18/s2:0000000000000000  r19/s3:000000000000000e  r20/s4:000000000000000e
      r21/s5:0000000000000000  r22/s6:0000000000000001  r23/s7:0000000000002000
      r24/t8:0000000000000002  r25/t9:0000000000000004  r26/k0:c000000000116954
      r27/k1:0000000000000020  r28/gp:c000000001334ef0  r29/sp:ffffffffffffbbe0
      r30/s8:a800000600471800  r31/ra:c0000000001a3418     EPC:c000000000034220
          CAUSE=28, SR=ffa3, BADVADDR=7fff0ca0
 8 fubyte[../ml/usercopy.s: 844, 0xc000000000034220]
 9 fault_in[../os/probe.c: 537, 0xc0000000001a3410]
10 fast_useracc[../os/probe.c: 156, 0xc0000000001a2db8]
11 useracc[../os/probe.c: 94, 0xc0000000001a2c78]
12 sendsig[../os/machdep.c: 449, 0xc000000000176710]
13 psig[../os/sig.c: 1923, 0xc00000000012a420]
14 postsysactions[../os/trap.c: 2211, 0xc000000000123628]
15 syscall[../os/trap.c: 2574, 0xc000000000123b60]
16 systrap[../ml/LOCORE/systrap.s: 419, 0xc0000000000396a0]
     r0/zero:0000000000000000   r1/at:0000000000000004   r2/v0:0000000000000004
       r3/v1:0000000000000004   r4/a0:0000000000000005   r5/a1:00000000101145e0
       r6/a2:000000007fff1080   r7/a3:0000000000000001   r8/a4:ffffffffffffffff
       r9/a5:000000000000002e  r10/a6:000000000faf3cc0  r11/a7:0000000000000000
      r12/t0:000000000fb4b350  r13/t1:000000000fb4db40  r14/t2:000000000fb55bc0
      r15/t3:000000003fff0000  r16/s0:ffffffffffffffff  r17/s1:000000001010d1cc
      r18/s2:0000000000000000  r19/s3:00000000101145e0  r20/s4:0000000000000010
      r21/s5:000000007fff107f  r22/s6:0000000000800000  r23/s7:0000000000000001
      r24/t8:000000001000d0a0  r25/t9:000000000fae0460  r26/k0:0000000000000000
      r27/k1:0000000000000020  r28/gp:0000000010116054  r29/sp:000000007fff1000
      r30/s8:0000000000000070  r31/ra:000000001000d0a0     EPC:000000000fae15a8
          CAUSE=8000, SR=ffffffff8400ffb3, BADVADDR=fa45168
============================================================
```

Example 3-7 more strace command output

```
>> strace a800000000f40000

PC=0xc000000000600c0c, SP=0xa800000000f43810
========================================================================
1 errprintif[../os/printf.c: 574, 0xc000000000600c0c]
2 errvprintif[../os/printf.c: 525, 0xc000000000600adc]
3 icmn_err[../os/printf.c: 207, 0xc0000000005ffe70]
4 cmn_err[../os/printf.c: 108, 0xc0000000005ff8ec
5 dobsuerre[../ml/SN0/klerror.c: 359, 0xc000000000312084]
6 k_trap[../os/trap.c: 360, 0xc00000000054bb1c]
7 trap[../os/trap.c: 603, 0xc00000000054c208]
8 VEC_trap[../IP27DEBUGbootarea/vec_trap.s: 2608, 0xc00000000032589c]
       r0/zero:0000000000000000  r1/at:0000000000000008  r2/v0:0000000000000001
         r3/v1:0000000000000003  r4/a0:96000000ffffffff8  r5/a1:0000000000000008
         r6/a2:a800000000f43d38  r7/a3:c000000001d2b0e8  r8/a4:0000000000000001
         r9/a5:0000000000000000 r10/a6:a800000000f37400 r11/a7:c00000000053a3dc
        r12/t0:0000000000000001 r13/t1:00000000000000a0 r14/t2:a800000000fale00
        r15/t3:ffffffff80000002 r16/s0:0000000100000009 r17/s1:a800000000f37400
        r18/s2:0000000200d00000 r19/s3:c00000000053a0a8 r20/s4:0000000000000001
        r21/s5:c000000000637bd4 r22/s6:a800000000f37300 r23/s7:c00000000053a2cc
        r24/t8:0000000000000001 r25/t9:0000000000000001 r26/k0:a800000000fa5900
        r27/k1:0000000000000001 r28/gp:a800000000f00110 r29/sp:a800000000fa5768
        r30/s8:a800000000f00108 r31/ra:c00000000053a744    EPC:0000000000000000
           CAUSE=0, SR=1, BADVADDR=1
Trace stops because bad SP 0xa800000000fa5768
========================================================================
.
.
[Several traces missing.]
.
.
PC=0xc000000000602308, SP=0xa800000000f43db8
========================================================================
1 panicspin[../os/printf.c: 1075, 0xc000000000602308]
2 doacvec[../os/pda.c: 1682, 0xc0000000005f4108]
3 cpuintr[../ml/SN0/intr.c: 1290, 0xc00000000031a958]
4 handle_intpend0[../ml/SN0/intr.c: 978, 0xc000000000319ce4
5 intr[../ml/SN0/intr.c: 1136, 0xc00000000031a3dc]
6 VEC_int[../IP27DEBIGbootarea/vec_int.s: 2596, 0xc000000000300188]
      r0/zero:0000000000000000  r1/at:0000000000000008  r2/v0:0000000000006800
        r3/v1:fffffffffffc1a0  r4/a0:0000000000000000  r5/a1:a800000000f00108
        r6/a2:0000000000000000  r7/a3:0000000000000001  r8/a4:00000000000068a0
        r9/a5:0000000000000000 r10/a6:000000000000ff00 r11/a7:0000000000000000
       r12/t0:0000000000000000 r13/t1:000000000000ffa1 r14/t2:00000000000000a1
       r15/t3:ffffffff80000002 r16/s0:000000000000ffa3 r17/s1:a800000000f43e98
       r18/s2:000000000000ffa0 r19/s3:0000000000000001 r20/s4:0000000000000000
       r21/s5:0000000000000000 r22/s6:0000000000000000 r23/s7:0000000000000000
       r24/t8:0000000000000000 r25/t9:0000000000000000 r26/k0:0000000000000000
       r27/k1:0000000000000000 r28/gp:0000000000000000 r29/sp:a800000000f03f88
       r30/s8:0000000000000000 r31/ra:c0000000006032b0    EPC:c00000000053afb0
           CAUSE=400, SR=ffa3, BADVADDR=c00000000200001e
7 idlerung[../os/scheduler/rung.c: 337, 0xc00000000053afb0]
8 idle[../os/machdep.c: 329, 0xc0000000006032f0]
========================================================================
```

Chapter 4

Advanced Topics

4.1 Extracting Data from the Stack Frame

4.1.1 Procedure Parameters

Registers a0 through a7 ($4...$11) will hold as many as the first eight words of integer type arguments passed to a procedure. For single and double precision arguments, registers fa0 through fa7 ($f12 .. $f19) are used. The values contained in these registers are not guaranteed to be saved across procedure calls. If the original register values need to be saved, they must be saved in the stack. In older versions of IRIX (pre 6.x), the calling function had to allocate space in its stack frame for parameters. With the new N32 and N64 ABIs, it is the responsibility of the called function to allocate space for and save the contents of the registers that are used to pass parameters.

- The following stack trace fragment (from an IRIX 6.4 system dump) shows how function parameters can be determined. This stack trace fragment represents a single stack frame.

```
3 sv_queue[../os/ksync/mutex.c: 1613, 0xc000000000537758]

RA=0xc000000000538b7c, SP=0xffffffffffffbd50, FRAME SIZE=112

ffffffffffffbd50: a800000000eb0000 0000000000000000
ffffffffffffbd60: 0000000000000008 ffffffffffffbd80
ffffffffffffbd70: 80006800ffffbd80 c000000000538b7c
ffffffffffffbd80: 0000000000000000 ffffffff80006800
ffffffffffffbd90: a800000000eb00b8 a800000001000000
ffffffffffffbda0: c000000000535fe4 a800000000eb0140
ffffffffffffbdb0: 0000000000000080 a80000000000ff10
```

- The kernel source code at the declaration of the function sv_queue() shows that the function requires six parameters.

```
Static void
v_queue(
    register sv_t *svp,
    int flags,
    unlock_func_t unlock_func,
    void *lock,
    uint lockbit,
    int ospl)
{
```

- To determine the storage location of the parameter values in the stack frame, disassemble the first ten instructions of the function. If the parameter values were saved in the registers, then this would be indicated by the first ten instructions. (You may need to disassemble more than the first ten instructions to see all the relevant instructions).

```
>> dis sv_queue 10
    ================================================================================
      [sv_queue:1539, 0xc000000000537308]   daddiu  sp,sp,-112
      [sv_queue:1539, 0xc00000000053730c]   move    ra,ra
      [sv_queue:1539, 0xc000000000537310]   sd      ra,40(sp)
      [sv_queue:1539, 0xc000000000537314]   sd      a0,64(sp)
      [sv_queue:1539, 0xc000000000537318]   sw      a1,76(sp)
      [sv_queue:1539, 0xc00000000053731c]   sd      a2,80(sp)
      [sv_queue:1539, 0xc000000000537320]   sd      a3,88(sp)
      [sv_queue:1539, 0xc000000000537324]   sw      a4,100(sp)
      [sv_queue:1539, 0xc000000000537328]   sw      a5,108(sp)
      [sv_queue:1540, 0xc00000000053732c]   li      v0,-16336
    ================================================================================
```

- As shown in the previous example, all of the registers that are used to pass parameters (a0 through a5) plus the return address (ra) are saved in the current stack frame. You can use the dump command to obtain each saved value by addressing the memory location pointed to by the stack pointer plus the frame offset indicated in the assembly language instruction. Note that if the stack address is from the kernel stack, defkthread must be set equal to the correct kthread (process) in order for addresses to be translated properly.

```
>> defkthread a800000000eb0000
Default kthread is 0xa800000000eb0000

>> od 0xffffffffffffbd50+64
Dumping memory starting at address : 0xffffffffffffbd90
ffffffffffffbd90: a800000000eb00b8 |........

>> od 0xffffffffffffbd50+76
Dumping memory starting at address : 0xffffffffffffbd9c
ffffffffffffbd9c: 01000000 |....

>> od 0xffffffffffffbd50+80
Dumping memory starting at address : 0xffffffffffffbda0
ffffffffffffbda0: c000000000535fe4 |.....S_.

>> od 0xffffffffffffbd50+88
Dumping memory starting at address : 0xffffffffffffbda8
ffffffffffffbda8: a800000000eb0140 |.......@

>> od 0xffffffffffffbd50+100
Dumping memory starting at address : 0xffffffffffffbdb4
ffffffffffffbdb4: 00000080 |....

>> od 0xffffffffffffbd50+108
Dumping memory starting at address : 0xffffffffffffbdbc
ffffffffffffbdbc: 0000ff10 |....
```

- Based on the output of the preceding example, the value of each of the sv_queue() parameters is as follows:

```
svp = a800000000eb00b8
fags = 0x1000000
unlock_func = c000000000535fe4
lock = a800000000eb0140
lockbit = 0x80
ospl = 0xff10
```

- As a sort of sanity check, to verify that these values are valid, disassemble the unlock_func function pointer to see if it is from a kernel function that makes sense:

```
>> dis c000000000535fe4
================================================================================
[sv_bitunlock:1266, 0xc000000000535fe4] daddiu sp,sp,-64
================================================================================
```

4.1.2 Local Variables

[This section is under development.]

4.1.3 Saved Registers

[This section is under development.]

Chapter 5

Guide to Core Dump Analysis

This chapter describes a methodology for analyzing IRIX system core dumps and describes each of the five stages of core dump analysis. At this point, the user should be somewhat familiar with the various ICRASH features.

5.1 An Overview of the Five Stages of Core Dump Analysis

STAGE 1: Gather relevant information.

Collect as much information as possible about the circumstances surrounding the system crashes and the site where the computer is installed. The more complete the information, the easier it will be to identify the cause of the problem.

STAGE 2: Take a first look.

The goal of this stage is to gain a basic understanding of what was happening in the system at the time the panic occurred. This quick overview might highlight symptoms of a problem that is already known and fixed, and it will provide a basis of how to focus the investigation.

STAGE 3: Determine WHAT caused the crash.

Analyze the kernel source code and figure out exactly what caused the system to crash. You need to have a more specific knowledge of the analysis tools, coupled with a good understanding of the internal structure and flow of the kernel at this stage. Note that the methodology for the analysis of system panic is slightly different than for a system hang.

STAGE 4: Determine WHY the crash occurred.

Determine how the actual cause of the problem relates to the source module where the problem originated.

STAGE 5: Fix the problem.

After you have determined the cause of the problem, correct it. If the problem was caused by faulty hardware, then replace it. If the problem was caused by software, then a modification to the source module where the problem originated, coupled with a patch, is in order. (Stage 5 is not truly a part of the analysis, but it concludes and verifies the analysis process.)

5.1.1 Gather Relevant Information

This subsection offers some guidelines for gathering information relevant to a system crash. It outlines which questions you should ask the customer and presents ways of determining the severity of the problem. In the event that the cause of the problem cannot be determined, the information that you gather should prove very useful to the person to whom the problem is escalated.

1. Ask some basic questions to understand the scope of the problem.

   Perhaps the most important step when dealing with a site that has been experiencing panics or hangs is to gather all relevant information about the system that has been experiencing the problem. The best source of this information is the person who administers the system. You should ask the system administrator the following standard questions. Following each question is a brief discussion about why the question is relevant.

   - How is the system failing?

     You need to know exactly what a customer means when he or she says that the system is "crashing." To some customers, this means that a key application stops working or dumps core. To others, it means that the system just locks up. Still others may refer to a system that has panicked and dumped core.

   - When did the system start having problems?

     It is important to know the time when a stable machine became unstable. With this information, you can focus on configurative changes to the system (such as hardware, software, or a patch) that might be associated with the failure.

   - How many times has the system crashed?

     There is a big difference between a system that crashes once and a system that crashes three times a day. The number of failures not only increases the priority of the problem, it also increases the depth of analysis possible. Often, you need more than one dump to pinpoint the cause of a problem.

   - What is the frequency of the crashes?

     This question is basically a continuation of the previous question.

   - What version of OS was running at the time of the crash?

     You need to know the OS version to determine which source tree to match a stack trace against. You also need this information if you are trying to match symptoms (panic string, error messages, and so forth) to known bugs. If the problem has already been fixed, you need to know the OS revision to determine which patch number fixes the problem.

   - Are there any patches installed on the system?

     Because each patch is built from a separate source tree, you must use the proper patch source tree to step back through a trace. It is also helpful to know if a patch was installed to resolve a recurrent problem; it is possible that the problem causing the crash was actually introduced by a patch.

   - Are there any third-party device drivers installed on the system?

A fatal error that occurs in a device driver usually brings down the entire system. There have been cases where third-party developers have reported system panics and sent in dumps, only to learn later that their device driver (which was under development) caused the system to crash. These types of problems should be directed to the developer. If the driver is part of an SGI product, then it should be handled just like any other kernel-related crash.

- Have any hardware or software changes recently been made to the system?

  Try to identify a connection between a change in the system and the time a system started to fail. For example, consider a situation in which a new FDDI board is installed in a machine on July 1 and this installation is the only change made to the hardware and software configuration of that system. The day after the new board is installed, the system begins to panic several times a day. Knowing the answer to this question (along with answers to the previous questions) should indicate that the FDDI board was involved in the recent string of crashes.

  The same sort of test can be applied to the software running on a system (for example, if a customer upgrades, or installs a new CAD/CAM package).

- Have there been any power failures, network modifications, movements of the machine, or any other reconfiguration since the crashes began?

  This question attempts to identify external factors that might be associated with the recurring system failures. For example, if there was a power outage on a system that had no UPS backup, the power supply might have been damaged, resulting in periodic system crashes. Another example might be one in which a new server was placed on the same circuit as the SGI system, which caused it to receive too little power and resulted in periodic crashes.

- Are the crashes reproducible?

  Ask the customer if he/she can reproduce the crashes. This step is a key to solving the problem more quickly. Bug reports that have a test case are much more likely to be resolved in a timely manner. Even if specific steps to reproduce the problem are not known, try to determine events that seem to correlate with the times of the crashes. For example, if a system is hanging frequently, you might find out (using netstat) that mbuf memory runs short just before the hangs occur.

- Is there anything else that might be a factor?

  Many times, asking the system administrators or users about what else might be involved in the system crashes can bring up additional useful information about the circumstances of the system crashes. You cannot have too much information.

2. Determine the severity of the impact on the customer.

   What is the customer's level of concern? Is the customer willing to monitor the situation? Or, is the customer likely to blow up and turn the call into a hot escalation? To a certain degree, the customer's behavior depends on how much damage to the data and operation the crashes cause and how frequently the crashes occur. From an analyst's point of view, it is better to have multiple core dumps that represent the same problem. With multiple core dumps, you will be better able to identify any failure patterns that exist.

3. Gather the dumps.

Find out where the dumps are located and gather all the information necessary to log on and take a look. In the case of a system that has been hanging, have the customer force an NMI panic (if the hung system has that capability). If the dumps cannot be viewed online or by modem, arrange to get them on tape and load them on an accessible machine.

Note that certain sites are considered secure. These customers may not allow any core dumps or electronic files to leave the premises. In most of these situations, it is possible for a properly cleared SGI support person to examine the core files on site. Also, the ICRASH CORE FILE REPORT, as well as other ICRASH output, can usually be screened by security and released in paper form.

5.1.2 Take a First Look at the Evidence

The following two items are examples of what you should be looking for when you analyze The ICRASH CORE FILE REPORT contains useful information about the event (panic or hang) leading up to the system core dump. The first two sections in the report are general in nature. They provide information about the system, the OS revision level, the time the dump was taken, and whether the dump was caused by a panic or an NMI. The remainder of the report contains specific information about the events leading up to a system panic and which jobs were running on each CPU at the time of the crash.

It is a good practice to review ICRASH reports from as many core dumps as possible to determine whether any similarities exist. The following list explains what you should look for:

1. Check the general information section to see if it contains any information that you did not obtain in the initial information-gathering stage. Useful information in this section includes the system name, the operating system release, the name of the system, the type of system, and the names and creation times of the `namelist` and `corefile`.
2. Determine if the core dump is the result of an internal panic or an NMI.

   There is a big difference in the way you attempt to analyze a core dump that resulted from a system panic as compared to one caused by an NMI. With a panic, there is a sequence of events that bring the entire system down. These events should be reflected in the stack trace of the process running at the time of the panic (or interrupt stack if no process was running). In the case of an NMI generated dump, the system did not crash (panic). Rather, it stopped performing tasks that users could recognize. It may even have stopped servicing interrupts (for example, attempts to `ping` the machine fail). It is possible that no processes are active and/or running in the CPUs. As you proceed, you should always be aware of the cause of the dump.
3. Check the FRU Analyzer output (if applicable) to see if any boards were flagged.

   Note that if the ICRASH report does not contain any output from the FRU analyzer, and the system is a CHALLENGE L or CHALLENGE XL server or an Onyx workstation, check to see whether a separate FRU report exists that has the same extension as the `unix` and `vmcore` analysis files.
4. Analyze the contents of the putbuf.

The first thing to look for in the putbuf is the panic string. The panic string should provide an indication of what caused the failure (if the dump was not induced by an NMI). At a minimum, it will provide you with a good place to start. After you have analyzed the panic string, check to see if any console error messages exist in the putbuf. Such messages may provide insight into the events that lead up to the crash.

Example 5-1 uncompvm output

```
kernel putbuf:
pb 0: single ended internal min xfer period 100ns
pb 1: S1 - WD95A SCSI controller 1 - differential internal min xfer period 100ns
pb 2: Configuring EPC in IO4 slot 15 padap 1 as et0
pb 3: root on dev 128,272; boot swap file on /dev/swap
pb 4:
pb 5: <4>WARNING: CPU 0 Bus Error Exception in User mode, eframe: 0xffffd1c0, EPC:
0x4034c4
pb 6: HARDWARE ERROR STATE:
pb 7: +  MC3 in slot 1
pb 8: +      MA EBus Error register: 0x4
pb 9: +        2: My EBus Data Error
pb 10: +  IP19 in slot 7
pb 11: +    CC in IP19 Slot 7, cpu 0
pb 12: +      CC ERTOIP  Register: 0x40
pb 13: +         6:MyResponse D-Resource TimeOut in A chip
pb 14: +  IO4 board in slot 15
pb 15: +      IA EBUS Error Register: 0x100
pb 16: +         8: Read Resource Time Out
pb 17:
pb 18: <0>PANIC: CPU 0: Bus Error Exception in User mode, eframe: 0xffffd1c0, EPC:
0x4034c4
pb 19:
pb 20: Dumping to dev 0x2000111 at block 0, space: 0x49bd0 pages
```

Example 5-2 uncompvm output, example 2

```
kernel putbuf:
pb 0:
pb 1: <4>WARNING: Paging Deamon (vhand) not running. NFS server down?
pb 2:
pb 3:
pb 4: <4>WARNING: Paging Deamon (vhand) not running. NFS server down?
pb 5:
pb 6:
pb 7: <4>WARNING: Paging Deamon (vhand) not running. NFS server down?
pb 8:
pb 9:
pb 10: <4>WARNING: Paging Deamon (vhand) not running. NFS server down?
pb 11:
pb 12:
pb 13: <4>WARNING: Paging Deamon (vhand) not running. NFS server down?
pb 14:
pb 15:
pb 16: <4>WARNING: Paging Deamon (vhand) not running. NFS server down?
```

Example 5-3 uncompvm command output, example 3

```
kernel putbuf:
pb 0:
pb 1: <0>PANIC: KERNEL FAULT
pb 2: PC: 0x880ec708 ep:ffffca80
pb 3: EXC code:16, `Read Address Error ''
pb 4: Bad addr: 0xa0f37739, cause: 0x80000010
pb 5: sr: 0xe003
pb 6:
pb 7:
pb 8: Dumping to dev 0x2000011 at block 0, space: 0x27ff pages
```

5. Check the status of each CPU
   - A one-line display indicates what was running on each CPU at the time the dump occurred. This section is particularly useful when the core dump was caused by an NMI.
   - Examine the stack trace that lead up to the system panic (except when the dump was caused by an NMI)
   - Check the stack trace that was generated for each CPU (especially with NMI induced dumps).
6. Analyze the panic string.
   - The panic string can provide an indication of the cause of the system crash if the core dump was not induced by an NMI.
   - Panic strings resulting from hardware errors.
   - The following five examples of panic strings all indicate failures that were caused by faulty hardware. With Bus Errors and ECC errors, you should check the results of the FRU analysis to try to determine which hardware component is bad. In the final example below, the case of the bad SIMM, the solution (indicating which SIMM is bad) is actually part of the panic string.

```
<0>PANIC: CPU 2: Bus Error in User mode, eframe:0xffffd1c0 EPC:0x10014e68

<0>PANIC: CPU 9: Bus Error in Kernel mode, eframe:0xffff3c68 EPC:0xa8008efc

<0>PANIC: CPU 7: G-cache (odd) parity error: hardware error (CPU 1 in slot
8), CAUSE 20000 SR 222007ff13

<0>PANIC: CPU 17: Uncorrectable ecc/parity error

<0>PANIC: IRIX Killed due to Memory Error in SIMM  S2A [HELP=MEM_ERROR]
```

   - Panic strings resulting from calls to the kernel `panic()` routine.

     The five panic strings below result from a direct call to the kernel panic() routine. Panic strings of this form make very good search strings when you are searching in OasisIII (SGI technical support information). Typically, these panic strings will be present in call and bug text and will identify other cases for comparison. In some instances, you may even find a solution to the problem at hand (e.g., a patch fixes the problem).

```
<0>PANIC: CPU 6: stack underflow/overflow

<0>PANIC: CPU 3: efs_sync: exlist trash
```

```
<0>PANIC: CPU 0: uipc 3

<0>PANIC: CPU 1: sat_pnconvert: outpath conversion overflow

<0>PANIC: CPU 3: receive 1
```

- Panic strings resulting from a KERNEL TRAP.

  The following panic message is generic. It indicates that a bad address or event was detected by the CPU rather than the IRIX kernel. You need to look at the Program Counter and the exception frame pointer (ep) that are part of the panic message to learn which routine was executing when the error was detected.

```
<0>PANIC: CPU 0: KERNEL FAULT
PC: 0xa8000000000ed504 ep: 0xffffffffffffbad8
EXC code:20, `Write Address Error '
Bad addr: 0x7265612f7470697c, cause: 0x30000014<CE=3,EXC=WADE>
sr: 0x6483<IM7,IM6,IM3,IPL=???,KX,MODE=KERNEL,EXL,IE>
```

7. Analyze Messages in the Putbuf.

   All console messages are automatically posted to the putbuf. Although these messages may provide useful information about the cause of a system panic or hang, they should always be viewed skeptically. The messages in the putbuf have no time or date stamp. Therefore, there is no way to tell whether a particular message was written just before a panic occurred or if it has been there for days. Try to match putbuf messages to entries in the SYSLOG (which do contain date and time stamps). Also, try to determine whether putbuf messages relate to the type of problem indicated by the panic string.

8. Search for a known problem.

   You should always try to match core dump details with open bugs or calls in the support database. In particular, you should look for other core dumps that:

   - Contain the same panic string
   - Have the same, or similar backtrace leading up to the panic
   - Are centered in particular parts of the kernel (for example in the socket subsystem)
   - Have similar types of data corruption (such as a bad socket pointer in a vnode)

9. Select a search string.

   Virtually any string of characters can be used as a search string. You will discover, however, that certain strings work better than others. For example, if you search on a string like "PANIC," you are likely to retrieve a large number of hits. Most or all of these hits will be irrelevant to the problem you are researching. A suggested list of suitable search strings includes:

   - Some or all of a panic string (this doesn't work with KERNEL TRAP panics)
   - Kernel function names from the stack backtrace
   - Source file names
   - Kernel structure names
   - Kernel subsystems

   You may have to try several possibilities before you find something useful.

10. Use `pv` and OasisIII.

    The SGI bug database (`pv`) and OasisIII are the two most useful tools to identify known problems that result in core dumps. Because all the bug-related information of `pv` is accessible through OasisIII, it can be used exclusively to perform the search. You can use the OasisIII application, or you can use the new Oasis page on the World Wide Web at:

    http://www-oasis.corp.sgi.com/about.oasis.html

    Regardless of which search tool you use, the process is the same. You select a unique character string from the putbuf, stack trace, etc., and use `pv` or OasisIII to search for matches in bugs, calls, or articles. There may be a match on a call or bug that describes the same or similar problem as the one that caused the dumps.

11. Check to see if a patch is available.

    If you discover a bug that has been fixed with a patch, determine the patch number and arrange to have it installed on the failing machine.

5.1.3 Determine WHAT Caused the Crash

[This section is under development.]

5.1.4 Determine WHY the Crash Occurred

[This section is under development.]

5.1.5 Fix the Problem

[This section is under development.]

5.2 What to do when a System Hang Occurs

Unlike a system panic, when a system hang occurs, no core dump is generated. The system is still running; it simply is not doing anything useful from the user's point of view. Usually, the system administrator can do little to get the system running again. The best course of action is to determine the cause of the hang. Depending on what type of system is hung, the following steps will help you determine what may be causing the hang:

1. Look at the serial console if one is available. Check to see if there are any messages, such as kernel notice or warning messages that indicate the cause of the hang. If there is a panic string, or if the running kernel is a debug kernel and there is a DBG: prompt, the system did not hang. The system panicked and at least tried to dump core.
2. Type a few characters on the console keyboard and see if the typed characters are echoed back to the console screen. If the characters are echoed back, the system is still able to service interrupts. The cause of the hang is probably a software problem. Initiate an NMI core dump (if that feature is supported on the hung system).

3. If the system has a power meter, determine whether there is any movent. Also check the disk lamps to see if any of the disk LEDs are blinking. (Make sure that the LED is not simply stuck on.) Any movement in the power meter or disk LEDs indicates that at least one process is able to run. The cause of the hang is probably a software problem. Initiate an NMI core dump (if that feature is supported on the hung system).

4. If the hung system is on a network, have another host on the network `ping` the hung system to see if it responds. If there is not a 100% loss of ping packets, the hung system is able to service interrupts. The cause of the hang is probably software. Initiate an NMI core dump if that feature is supported on the hung system.

5. If you have not determined the cause of the problem in the previous four steps, and if the hung system is a CHALLENGE L or CHALLENGE XL server, an Onyx workstation, or an Origin series system, initiate an NMI core dump. If one NMI causes the system to panic and issues the following message:

   ```
   PANIC: User requested vmcore dump (NMI)
   ```

   the processors are responding normally. The problem is presumably caused by software and possibly related to the network.

   If there is no response to the first NMI, issue a second NMI. The bootmaster processor will try to enter the IP19/IP21 PROM-resident power-on diagnostic monitor called POD. If the second NMI is successful, some hardware condition has probably caused the system to hang. You should analyze the hardware error state with POD to determine the cause.

6. If there is no response to the second NMI, try to reset the system into POD mode. Any error bits that were set before you reset the system will remain set after the system comes up in POD mode. An analysis of these bits should indicate which board failed. If no hardware error bits are set, it is likely that a software problem, perhaps corrupted memory, caused the hang.

   Note: It is possible that a hardware problem that prevents one CPU from executing can result in a system hang, even when other CPUs continue to function normally.

Chapter 6

Examining a Sample Core Dump

Using a sample core dump, this Chapter explains how to use ICRASH to perform system core dump analysis.

6.1 Look at the Dump Header

Before you use ICRASH to look at a core dump, enter the `/etc/uncompvm` command to examine the dump header. The `uncompvm` command expands a compressed IRIX vmcore dump of the operating system. Enter the following command:

```
# uncompvm -h corename
```

where `corename` is the name of the core image that you want to analyze. The `uncompvm` command displays the contents of the dump header which contains some general information about the core dump, and a copy of the putbuf (a circular output buffer) at the time the dump occurred. The following example illustrates typical output from this command:

```
# uncompvm -h vmcore.5.comp
        Dump Header Information
------------------------------------------------------
uname:        IRIX sgigate 5.3 12200158 IP19
physical mem: 512 megabytes
phys start:   0x0
page size:    4096 bytes# uncompvm -h vmcore.5.comp
dump version: 1# uncompvm -# uncompvm -h vmcore.5.comph # uncompvm -h
vmcore.5.compvmcore.5.comp
dump size:    262144 k
crash time:   Tue Aug  1 12:47:49 1995
panic string: <0>PANIC: CPU 3: KERNEL FAULT
kernel putbuf:
pb 0: r/people (dev: 128/272)
pb 1: <5>NOTICE: Start mounting filesystem: /tmp
pb 2: <5>NOTICE: Ending xFS recovery for filesystem: /tmp (dev: 128/23)
pb 3: <5>NOTICE: Start mounting filesystem: /var/spool/news
pb 4: <5>NOTICE: Ending xFS recovery for filesystem: /var/spool/news
(dev: 192/4)
pb 5: dks2d1vh: Recovered Error: Read data recovered with retries
(asc=0x17, asq=0x9), (data byte 11), Block #394922
pb 6: ot 5 padap 1 as et0
pb 7: root on dev 128,16; boot swap file on /dev/swap
```

```
pb 8: <5>NOTICE: Start mounting filesystem: /
pb 9: <5>NOTICE: Ending xFS recovery for filesystem: / (dev: 128/16)
pb 10: <5>NOTICE: Start mounting filesystem: /var
pb 11: <5>NOTICE: Ending xFS recovery for filesystem: /var (dev: 128/22)
pb 12: <5>NOTICE: Start mounting filesystem: /usr/local/news
pb 13: <5>NOTICE: Ending xFS recovery for filesystem: /usr/local/news
pb 14: <0>PANIC: CPU 3: KERNEL FAULT
pb 15: PC: 0x80120d7c ep:ffffcb68
pb 16: EXC code:128, `Software detected SEGV '
pb 17: Bad addr: 0x763, cause: 0x8<CE=0,EXC=RMISS>
pb 18: sr: 0x6403<IM7,IM6,IM3,IPL=???,MODE=KERNEL,EXL,IE>
pb 19:
pb 20:
pb 21: Dumping to dev 0x2000011 at block 0, space: 0x10000 pages
```

Notice that the copy of the putbuf placed in the dump header is somewhat closer to the actual time of panic than the one that ends up in the `vmcore` image. In certain circumstances, information in the dump header is overwritten before the putbuf is saved to the `vmcore` file.

6.2 Start the ICRASH Utility

Start the ICRASH utility by entering the following command:

```
# icrash unix.5 vmcore.5.comp

corefile = vmcore.5.comp, namelist = unix.5, outfile = stdout
```

where `vmcore.5.comp` is the file that contains the system memory image, and `unix.5` is a copy of the `unix (unix.N)` file that was executing at the time of the crash. Refer to the ICRASH(1M) man page for a detailed description of this instruction.

While the program is processing the following message appears on the display screen:

```
Please wait.............................

>>
```

6.3 Get the System Status

Determine the state of the system at the time of the crash. The ICRASH `stat` command provides some basic information about the system that panicked. Specifically, it includes the:

- System name
- OS release
- Time of crash
- Age of system

This information is followed by the contents of the internal circular buffer called the *putbuf*. This buffer contains the most recent system ERROR and NOTICE messages. Normally, the putbuf is where you would find a copy of the actual messages associated with the panic. This buffer may also contain other messages that indicate why the system went down. There is no guarantee, however, that messages in the putbuf (except for the panic string) have anything to do with the cause of the crash. Because messages in the putbuf do not contain a time stamp, there is no way to know when they were entered.

```
>> stat

system name:    IRIX
release:        5.3
node name:      sgigate
version:        12200158
machine name:   IP19
time of crash:  Aug  01 12:47:49 1995
system age:     3 day, 19 hour, 51 min.
putbuf:
<0>PANIC: CPU 3: KERNEL FAULT
PC: 0x80120d7c ep:ffffcb68
EXC code:128, `Software detected SEGV '
Bad addr: 0x763, cause: 0x8<CE=0,EXC=RMISS>
sr: 0x6403<IM7,IM6,IM3,IPL=???,MODE=KERNEL,EXL,IE>

Dumping to dev 0x2000011 at block 0, space: 0x10000 pages
Dumping unmapped memory from page 0x0 to page 0x8d5..... /tmp
<5>NOTICE: Ending xFS recovery for filesystem: /tmp (dev: 128/23)
<5>NOTICE: Start mounting filesystem: /var/spool/news
<5>NOTICE: Ending xFS recovery for filesystem: /var/spool/news (dev: 19 2/4)
dks2d1vh: Recovered Error: Read data recovered with retries (asc=0x17, asq=0x9),
(data byte 11), Block   #394922
ot 5 padap 1 as et0
root on dev 128,16; boot swap file on /dev/swap
<5>NOTICE: Start mounting filesystem: /
<5>NOTICE: Ending xFS recovery for filesystem: / (dev: 128/16)
<5>NOTICE: Start mounting filesystem: /var
<5>NOTICE: Ending xFS recovery for filesystem: /var (dev: 128/22)
<5>NOTICE: Start mounting filesystem: /usr/local/news
<5>NOTICE: Ending xFS recovery for file

ERROR DUMPBUF:

0x801d0a40 = ""
```

In the example above, the panic is caused by a kernel trap (a KTRAP is an error that is detected by a CPU and not an error detected by the kernel), that occurred when the kernel attempted to read from address 0x763. The actual panic string exists in the putbuf as:

```
<0>PANIC: CPU 3: KERNEL FAULT
PC: 0x80120d7c ep:ffffcb68
EXC code:128, `Software detected SEGV '
Bad addr: 0x763, cause: 0x8<CE=0,EXC=RMISS>
sr: 0x6403<IM7,IM6,IM3,IPL=???,MODE=KERNEL,EXL,IE>
```

6.4 Determine Which Processes Were Running at the Time of the Crash

There are several ways to determine which processes were running at the time of the crash:

First, determine the default process slot (automatically set to the process slot for `dumpproc`, the process running at the time of the system panic):

```
>> defproc
Default process slot is 210

>> proc 210
PROC TABLE SIZE = 2214
PROC ST   PID  PPID  PGID  SUID   UID PRI CPU       EVENT    FLAGS NAME
================================================================================
 210  2 13849 13848  1648     0     0  61   2    00000000 30000030 fuser
================================================================================
1 proc struct found
Or enter the curproc command to see the process that is currently running on each CPU:
>> curproc
CPU0: CURPROC = 0x0

CPU1: CURPROC = 0x8037a2c0
PROC ST   PID  PPID  PGID  SUID   UID PRI CPU    EVENT FLAGS    NAME
================================================================================
  57  2  9404  9403     0  7298  7298  61   2 00000000 20000030 sadc

CPU2: CURPROC = 0x0

CPU3: CURPROC = 0x8039abe8
PROC ST   PID  PPID  PGID  SUID   UID PRI CPU    EVENT FLAGS    NAME
================================================================================
 210  2 13849 13848  1648     0     0  61   2 00000000 30000030 fuser
```

6.5 Get a Stack Trace for the Currently Running Process

Get a stack trace for the currently running process. The `trace` command can also be used to display the stack trace for `defproc`, which is by default the process that was running when the panic occurred. The output from the `trace` command is as follows:

```
>> trace
  ================================================================
  STACK TRACE FOR PROCESS 210 (fuser, PID=13849):

   0 syncreboot[../os/printf.c: 851, 0x8002f168]
   1 icmn_err[../os/printf.c: 255, 0x8002cc00]
   2 cmn_err[../os/printf.c: 100, 0x8002c7f4]
   3 panicregs[../os/trap.c: 194, 0x80049f58]
   4 k_trap[../os/trap.c: 358, 0x8004a2e8]
   5 trap[../os/trap.c: 398, 0x8004a350]
   6 VEC_trap[../ml/locore.s: 4023, 0x8000f39c]
         r0/zero:0000000000000000  r1/at:0000000000000001  r2/v0:000000000000ff01
           r3/v1:ffffffffb8000828  r4/a0:ffffffffffffffff  r5/a1:ffffffff8000de50
           r6/a2:0000000000000004  r7/a3:ffffffff9410778a  r8/t0:0000000000000001
           r9/t1:ffffffff00006401 r10/t2:0000000000000074 r11/t3:000000000000ff00
```

```
        r12/t4:0000000000000001 r13/t5:0000000000000000 r14/t6:0000000000000004
        r15/t7:0000000000000000 r16/s0:fffffffffffffffc r17/s1:ffffffff8000de50
        r18/s2:ffffffff8d626164 r19/s3:ffffffff80b49d80 r20/s4:ffffffff82fca360
        r21/s5:ffffffff94107778 r22/s6:ffffffff801ccfd8 r23/s7:ffffffff801ac958
        r24/t8:0000000000000000 r25/t9:0000000000000000 r26/k0:00000064886abb68
        r27/k1:0000000000000020 r28/gp:803bc6189410777a r29/sp:fffffffffffccc8
        r30/fp:0000000010000314 r31/ra:ffffffff80120d7c
            EPC=ffffffff80120d7c, CAUSE=8, SR=6403, BADVADDR=763
 7 io_splockspl[../ml/EVEREST/llsclocks.s: 297, 0x80120d7c]
 8 psema[../os/sema.c: 1221, 0x80039300]
 9 uipc_vget[../bsd/socket/uipc_vfsops.c: 104, 0x800dad10]
10 lookupanon[../os/utssys.c: 173, 0x8005eedc]
11 uts_fusers[../os/utssys.c: 135, 0x8005edc0]
12 utssys[../os/utssys.c: 106, 0x8005ed7c]
13 syscall[../os/trap.c: 2067, 0x8004c398]
14 systrap[../ml/locore.s: 4256, 0x8000f4cc]
================================================================
```

The stack trace provides a roadmap of the steps that lead up to the system panic. It is helpful to have access to kernel source code. With the source code you can actually "walk" (sequentially examine) the trace, routine by routine to reconstruct exactly how the panic occurred. You can even, with a little effort, determine the values assigned to various parameters and local variables. Each line in the stack trace contains the following information:

- The name of the kernel routine being called
- The source module in which the function is contained and the line number containing that instruction
- The virtual address of the current instruction (which should be a jump to the next routine in the stack)

Some stack traces (as in this example) include one or more Exception Frames (*eframes* or register dumps). The eframe contains the exact state of the CPU registers at the time a KERNEL TRAP occurs. Values contained in the various registers may be useful when you are trying to reconstruct exactly what happened.

6.6 Determine the Level of the Stack where the Panic Occurred

You should know how to find the stack frame that was active when the panic occurred. After all, that is probably where the 'what' question will be answered. In general, the stack level that precedes the `k_trap` in the stack (but is displayed one address below on the screen) is the stack level that was active when the panic occurred. (This assumption would be different for panics that originated from the IRIX code itself). In the example above, the active stack level when the panic occurred is the following line:

```
7 io_splockspl[../ml/EVEREST/llsclocks.s: 297, 0x80120d7c]
```

Note that all stack levels above this point (that is, more recent stack entries) relate only to the process of shutting the system down, syncing disks, dumping core, and so forth.

6.7 View the Process Table Entry for the Suspected Process

The process table provides the balance of information available about the process running at the time the panic occurred. Some useful information that you can obtain by issuing this command includes:

- The current state of the process (usually "2" for running processes)
- The process identifier (PID) for the process and the parent process (you can use the PID to search for any children of this parent process)
- The kernel address of the 'event' the process (disk buffer, STREAM queue, and so forth) is sleeping on if the process is currently asleep
- The process regions (pregions) currently mapping the process address space

To view the process table entry for the process that was running at the time of the panic, enter the following command:

```
>> proc -f 210

PROC TABLE SIZE = 2214
PROC ST   PID  PPID  PGID  SUID   UID PRI CPU    EVENT FLAGS    NAME
================================================================================
 210  2 13849 13848  1648     0     0  61   2 00000000 30000030 fuser

W2CHAN: 0x0
PROC: 0x8039abe8, PARENT: 0x803c3d50, CHILD: 0x0, SIBLING: 0x0
UBLOCK: 0xd35bd000 (PFN: 68112), STACK: 0xffffc000 (PFN: 124689)
UTIME: 1, STIME: 6, SIZE: 210, RSS: 5, MAXRSS: 127040, NLOCKPG: 0
SONPROC: 3, LASTRUN: 3, MUSTRUN: -1, RESIDENT: 0, SQSELF: 1, ONRQ: 0

P_SEMA:

          SEMA   COUNT      LOCK     QUEUE      QEND     OWNER  FLAGS
  ------------------------------------------------------------------------
  8039acc0       1  8039acc4         0         0         0      4

P_WAIT:

          SEMA   COUNT      LOCK     QUEUE      QEND     OWNER  FLAGS
  ------------------------------------------------------------------------
  8039acd0       0  8039acd4         0         0         0      0

PREGION TABLE:

  PREG   PREGION     REGVA    REGION   PGLEN  VALID  TYPE  FLAGS      PMAP
  ------------------------------------------------------------------------
     0  945102d0     400000  82ea0aa0       2      2     9      0  80c54594
     1  8c75a438    fa80000  99072a50     193     48     9      0  80c54594
     2  8c20dab0    fb50000  918d7780       7      5     9      1  80c54594
     3  8fffa7e0    fb60000  81f49550      80     27     9      0  80c54594
     4  94510708    fbc0000  8bb23140       2      2     9      1  80c54594
     5  96934828   10000000  95063370       8      3     9      1  80c54594
     6  9dfc03f0   7fff8000  9ae48a00       8      7     3      1  80c54594

  ================================================================================
  1 proc struct found
```

6.8 View the User Area for the Currently Running Process

The `user` command displays the user area for a given process. Some of the information that you can obtain with this command includes:

- The full command string for the process (including arguments)
- All the kernel files the process has open

To view the user area for a given process, enter the following command, including the process number:

```
>> user -f 210

PROC      USER      PROC      BASE COUNT ERROR  COMMAND
=============================================================================
 210 d35bd000 8039abe8         0     0     0  fuser

PS ARGS:
/sbin/fuser gate-sgigate:11751/tcp

CDIR=0x80b340c0, RDIR=0x0

FILE DESCRIPTOR TABLE:

  FD      FILE  RCNT       VNODE        OFFSET     FLAGS
  ======================================================
   0  8ef260f0     4    80f9f6c0             0         1
   1  86f92118     3    95f4da80             2         2
   2  86f92118     3    95f4da80             2         2
=============================================================================
1 user struct found
```

You should now have the following information about the process that was running when the panic occurred:

- The command in execution was `fuser`, which was invoked with the command line: `/sbin/fuser gate-sgigate:11751/tcp`
- This process has three open kernel files

6.9 Print the List of Files that Process 210 Has Opened

Enter the following command to list the files that process 210 has opened:

```
>> file -p 210

OPEN FILES FOR PROC 210:

FD      FILE  RCNT       VNODE        OFFSET     FLAGS
=====================================================
 0  8ef260f0     4    80f9f6c0             0         1
 1  86f92118     3    95f4da80             2         2
 2  86f92118     3    95f4da80             2         2
=====================================================
3 file structs found
```

6.10 Begin Analyzing Source Code

Now you need to begin analyzing source code in order to determine exactly which data or error condition is responsible for the crash. You can approach this analysis in one of two ways: either from the top down or from the bottom up.

First, the terms *top* and *bottom* need clarification. Because the stack grows down in memory, the "top" of the stack actually has a higher memory address than the "bottom" of the stack. The `trace` command, however, places the top of the stack at the bottom of the display screen. This arrangement can be somewhat confusing. In this document, the terms "top" and "bottom" refer to a stack frame's relation to the screen. You can literally start at the first or last function call in the stack trace and work your way (stepping through the code as you go) to the place where the crash occurred, although this approach is usually not necessary. There are a number of short cuts that can help you isolate the routine or routines that are involved in the crash.

Hint: Unless you know the code for the modules involved in the trace very well, it is better to go through the levels one-by-one to properly reconstruct the steps that lead up to the panic.

As mentioned earlier, one of the first things you need to do is to identify the exact instruction where the failure occurred. This task is easy when you are dealing with a panic caused by a KTRAP because the Program Counter is included in the panic message. Identifying the instruction is only slightly more difficult when the crash resulted from a direct call to `panic()`. In our example, a KTRAP panic, the crash occurred at Line 297 in the `io_splocksplx()` routine (the exact instruction address is 0x80120d7c). With this knowledge, skip the first 7 frames (0 through 6) in the traceback. Except for the values stored in the exception frame, frames 0 through 6 contain very little useful information. They relate only to the acts of panicking and dumping core.

6.11 Walk the Stack Trace

Display the stack trace that lead to the panic by entering:

```
>> trace
================================================================
STACK TRACE FOR PROCESS 210 (fuser, PID=13849):

 0 syncreboot[../os/printf.c: 851, 0x8002f168]
 1 icmn_err[../os/printf.c: 255, 0x8002cc00]
 2 cmn_err[../os/printf.c: 100, 0x8002c7f4]
 3 panicregs[../os/trap.c: 194, 0x80049f58]
 4 k_trap[../os/trap.c: 358, 0x8004a2e8]
 5 trap[../os/trap.c: 398, 0x8004a350]
 6 VEC_trap[../ml/locore.s: 4023, 0x8000f39c]
      r0/zero:0000000000000000   r1/at:0000000000000001   r2/v0:000000000000ff01
        r3/v1:ffffffffb8000828   r4/a0:ffffffffffffffff   r5/a1:ffffffff8000de50
        r6/a2:0000000000000004   r7/a3:ffffffff9410778a   r8/t0:0000000000000001
        r9/t1:ffffffff00006401  r10/t2:0000000000000074  r11/t3:000000000000ff00
       r12/t4:0000000000000001  r13/t5:0000000000000000  r14/t6:0000000000000004
       r15/t7:0000000000000000  r16/s0:fffffffffffffffc  r17/s1:ffffffff8000de50
       r18/s2:ffffffff8d626164  r19/s3:ffffffff80b49d80  r20/s4:ffffffff82fca360
```

```
        r21/s5:ffffffff94107778  r22/s6:ffffffff801ccfd8  r23/s7:ffffffff801ac958
        r24/t8:0000000000000000  r25/t9:0000000000000000  r26/k0:00000064886abb68
        r27/k1:0000000000000020  r28/gp:803bc6189410777a  r29/sp:fffffffffffccc8
        r30/fp:0000000010000314  r31/ra:ffffffff80120d7c
            EPC=ffffffff80120d7c, CAUSE=8, SR=6403, BADVADDR=763
 7 io_splockspl[../ml/EVEREST/llsclocks.s: 297, 0x80120d7c]
 8 psema[../os/sema.c: 1221, 0x80039300]
 9 uipc_vget[../bsd/socket/uipc_vfsops.c: 104, 0x800dad10]
10 lookupanon[../os/utssys.c: 173, 0x8005eedc]
11 uts_fusers[../os/utssys.c: 135, 0x8005edc0]
12 utssys[../os/utssys.c: 106, 0x8005ed7c]
13 syscall[../os/trap.c: 2067, 0x8004c398]
14 systrap[../ml/locore.s: 4256, 0x8000f4cc]
================================================================
```

Determine the level in the trace where the panic occurred. In the case of a KERNEL FAULT panic, the value of the Program Counter that initiated the panic is located in the panic string:

```
<0>PANIC: CPU 3: KERNEL FAULT
PC: 0x80120d7c ep:ffffcb68
EXC code:128, `Software detected SEGV '
Bad addr: 0x763, cause: 0x8<CE=0,EXC=RMISS>
sr: 0x6403<IM7,IM6,IM3,IPL=???,MODE=KERNEL,EXL,IE>
```

In this example, the address of the instruction that caused the panic is 0x80120d7c. This instruction is from the kernel routine `io_splockspl()` (line 297 in the source file), which is level 7 in the stack trace.

Determine which source tree to use when examining the source code of individual kernel routines. Reference the kernel source tree for this example by using the following relative pathname:

```
bonnie.engr.sgi.com:/proj/irix5.3/isms/irix/kern
```

Look at the source code for the routine. You first need to determine which source tree to look at. In the preceding example, the OS running on the system is revision 5.3 (it has no rollup patches installed).

Unlike most kernel routines, the `io_splockspl()` routine is written in assembly language. It is a very compact routine that determines whether the `spinlock` passed in is busy and that spins until the lock is free. When the `io_splockspl()` routine finally acquires the lock, it calls the kernel routine that the second parameter points to and sets an interrupt protection level.

Use following command to disassemble the instruction that caused the panic:

```
>> dis 0x80120d7c
================================================================================
  [io_splockspl:297] 0x80120d7c:        c20c0000        ll      t4,0(s0)
================================================================================
```

Hint: The `io_splockspl()` routine and the routine that called it (`psema()`) are called so frequently in the kernel that it is unlikely that either one caused the panic. Otherwise, there would have been major problems throughout the kernel.

```
Bug Report
----------

Incident # : 295611
Submitter : yakker
Submitter Domain : csd
Submitter Machine : awesome
Opened Date : 08/08/95
Category : software
Classification : bug
Assigned Group : os-net
Alpha # : 1232225736
Product :
Product Version :
Command :
Summary : race condition with uipc_vnlist[] causes system panic
Priority : 2
Machine :
Model CPU :
Model GFX :
Peripheral :
Doc Affected :
Reproducible :
SGI_only :
CSD Priority :
message_id :
Newsgroups :
Released Product :
Customer Reported :
TVBUG ID :
Description :
There is a race condition in the UIPC code where a process will free up a socket
pointer (with soclose()) in uipc_close(), and before it can remove the vnode pointer
from the uipc_vnlist[], someone else obtains the LOCK_VNLIST() and accesses the
socket pointer from the soon-to-be freed vnode pointer (which has been freed up), and
causes a system panic.
```

6.12 Problem Summary

This subsection summarizes the details of the example failure:

```
OS: IRIX 5.3 with XFS
PATCHES: 183, 204, 281, 466, 481, 530, 534, 602, 603, 624, 632
CORES: dcoserv.corp:/usr/people/assist/sgigate #4 and #5

>> pda
CPU       PDA  STKFLG     STACK   CURPROC   LTICKS   CURLOCK  LASTLOCK
==========================================================================
  0  8026a000       2  801ad250         0        3         0         0
  1  80272000       2  808ec000         0        3         0         0
  2  8027a000       1  ffffc000  80380c58        2         0         0
  3  80282000       2  808d9000         0        3         0         0
==========================================================================
4 pda_s structs found
```

```
There is one process running:

>> t
================================================================
STACK TRACE FOR PROCESS 88 (fuser, PID=2566):

0 syncreboot[../os/printf.c: 851, 0x8002f168]
1 icmn_err[../os/printf.c: 255, 0x8002cc00]
2 cmn_err[../os/printf.c: 100, 0x8002c7f4]
3 panicregs[../os/trap.c: 194, 0x80049f58]
4 k_trap[../os/trap.c: 358, 0x8004a2e8]
5 trap[../os/trap.c: 398, 0x8004a350]
6 VEC_trap[../ml/locore.s: 4023, 0x8000f39c]
      r0/zero:0000000000000000  r1/at:0000000000000001  r2/v0:000000000000ff01
        r3/v1:ffffffffb8000828  r4/a0:ffffffffffffffff  r5/a1:ffffffff8000de50
        r6/a2:0000000000000004  r7/a3:0000000000000000  r8/t0:0000000000000001
        r9/t1:ffffffff00006401 r10/t2:0000000000000074 r11/t3:000000000000ff00
       r12/t4:0000000000000001 r13/t5:0000000000000000 r14/t6:ffffffff80380ce9
       r15/t7:ffffffff80380ce8 r16/s0:fffffffffffffffc r17/s1:ffffffff8000de50
       r18/s2:ffffffff90cc8ca4 r19/s3:ffffffff80c6b880 r20/s4:ffffffff9c6d33c0
       r21/s5:ffffffff8e13db40 r22/s6:ffffffff801ccfd8 r23/s7:ffffffff801ac95
       r24/t8:0000000000000000 r25/t9:0000000000000000 r26/k0:ffffffff823f15a0
       r27/k1:0000000000000020 r28/gp:ffffffff801ccfd8 r29/sp:fffffffffffccc8
       r30/fp:0000000010000314  r31/ra:ffffffff80120d7c
           EPC=ffffffff80120d7c, CAUSE=8, SR=6403, BADVADDR=734
7 io_splockspl[../ml/EVEREST/llsclocks.s: 297, 0x80120d7c]
8 psema[../os/sema.c: 1221, 0x80039300]
9 uipc_vget[../bsd/socket/uipc_vfsops.c: 104, 0x800dad10]
10 lookupanon[../os/utssys.c: 173, 0x8005eedc]
11 uts_fusers[../os/utssys.c: 135, 0x8005edc0]
12 utssys[../os/utssys.c: 106, 0x8005ed7c]
13 syscall[../os/trap.c: 2067, 0x8004c398]
14 systrap[../ml/locore.s: 4256, 0x8000f4cc]
================================================================
```

`Fuser` is the process that caused the system panic. Fuser is failing because of a spurious call to `psema()` within `uipc_vget()`. The relevant code in `uipc_vget()` is:

```
[...]

96  LOCK_VNLIST();
97
98  for (vp = uipc_vnlist.vl_next; vp != (struct vnode *) &uipc_vnlist;
99        vp = vp->v_next) {
100
101       so = vp->v_data;
102       version = vp->v_version;
103
104       SOCKET_LOCK(so, s);

[...]
```

The `LOCK_VNLIST()` is obtained, and it then begins to go through all of the `uipc_vnlist[]` vnodes to perform some operation. It then tries to perform a `SOCKET_LOCK()` on the `vp->v_data`, which points to a free block and dies.

The process that is waiting for the `LOCK_VNLIT()` (the one that released the socket pointer and is trying to free and release the vnode pointer from the `uipc_vnlist[])` is:

```
>> t 296
================================================================
STACK TRACE FOR PROCESS 296 (betasockd, PID=2563):

0 swtch[../os/swtch.c: 376, 0x800291e4]
1 semawait[../os/sema.c: 770, 0x80038a64]
2 psema[../os/sema.c: 1248, 0x80039374]
3 dropsockvp[../bsd/socket/uipc_vfsops.c: 64, 0x800dac44]
4 uipc_close[../bsd/socket/uipc_vnodeops.c: 68, 0x800db018]
5 closef[../os/fio.c: 443, 0x80071d90]
6 close[../os/vncalls.c: 242, 0x8005100c]
7 syscall[../os/trap.c: 2067, 0x8004c398]
8 systrap[../ml/locore.s: 4256, 0x8000f4cc]
================================================================
```

The relevant code here for `uipc_close()` and `dropsockvp()` is:

```
50  /* ARGSUSED */
51  int
52  uipc_close(struct vnode *vp,
53         int flag,
54         lastclose_t lastclose,
55         off_t offset,
56         struct cred *cr)
57  {
58      int error;
59
60      if (lastclose == L_FALSE)
61          return 0;
62                  .
63      if (vp->v_data) {
64          error = soclose(vp->v_data);
65      } else
66          error = 0;
67
68      dropsockvp(vp);
69      return error;
70  }
```

and:

```
58  /*
59   * Drop vp from the active socket vnode list.
60   */
61  void
62  dropsockvp(struct vnode *vp)
63  {
64      LOCK_VNLIST();
65      vn_unlink(vp);
66      uipc_vncount--;
67      UNLOCK_VNLIST();
68      vp->v_data = 0;
69  }
```

Note how in `uipc_close()` we use the `soclose()` instruction on the `vp->v_data`, and then call the `dropsockvp()` routine. The `betasockd` process (process slot 296) is waiting for the `LOCK_VNLIST()` at line 64.

The cause of the panic is understood. The only solution now is to set the `LOCK_VNLIST()` around the `soclose()` operation to avoid the failure.

Appendix A

ICRASH Command Listing

This appendix contains lists of commands that are used in several versions of ICRASH. Refer to the icrash(1M) man page for additional information on each of these commands

A.1 ICRASH (Revision 5.3) Commands

base	bfind	bhash (bufhash)	buf
bufhash	cprf	curproc	dblock
defproc	dis	dump	eframe
eval	f (file)	file	findpde
findsym	fpde (findpde)	from	fs (fstype)
fstype	fsym (findsym)	h (history)	help
history	i (inode)	inode	in (inpcb)
inpcb	lastproc	mblock	mbstat
mbuf	nm (symbol)	od (dump)	outfile
p (proc)	pda	pde	pfdat
pfdathash	pfind	phash (pfdathash)	pmap
preg (region)	pregion	priv (private)	private
proc	ptov	q (quit)	queue
quit	reg (region)	region	mode
runq	search	sema	sh
sizeof	snode	soc (socket)	socket
source	stat	str (stream)	stream
string	strings (string)	strst (strstat)	strstat
strace	swap	sym (symbol)	symbol
t (trace)	tcp	tcpcb (tcp)	tlb
tlbdump (tlb)	trace	u (user)	uipcvn
un (unpcb)	unpcb	use (source)	user
v (vnode) vfs	vfs	vnode	vtop
zone	?		

A.2 ICRASH (Revisions 6.2 and 6.3) Commands

?	history	printd	straem
avlnode	i	printo	string
base	in	printx	strings
block	inode	proc	strst
bucket	inpcb	ptov	strstat
ctrace	kthread	px	struct
curproc	ktrace	q!	swap
d	lastproc	q	sym
debug	mbstat	queue	symbol
defslot	mbuf	quit	t
defproc	mempool	reg	tcp
die	nm	region	tcpcb
dis	od	report	tlb
dump	outfile	mode	tlbdump
eframe	p	runq	trace
etrace	page	sbe	type
eval	pager	search	u
f	pd	sema	uipcvn
file	pda	sh	un
findsym	pde	sizeof	unpcb
from	pfdat	slpproc	user
fs	pfdathash	snode	v
fstype	pfind	soc	vfs
fsym	phash	socket	vnode
func	po	stack	vtop
h	preg	stat	whatis
help	pregion	str	zone
hinv	print	strace	

A.2.1 Commands Added Since 5.3 Version

avlnode

block Debug)

bucket (debug)

ctrace

debug

die

etrace

func

hinv

kthread

ktrace

mempool (debug)

page (debug)

pager

print

printd

printo

printx

report

sbe

slpproc

stack

struct

type

whatis

A.2.2 Commands Removed Since 5.3 Version

bfind

bufhash

buf

cprf

dblock

findpde

source

mblock

pmap

private

A.2.3 Commands Functionality Changed Since 5.3 Version

eval	now an alias for print

A.3 [ICRASH (Revision 6.4) Commands

?	history	printo	string
avlnode	hwpath	printx	strings
base	i	proc	strst
block	in	ptov	strstat
bucket	inode	px	struct
config	inpcb	q!	swap
ctrace	ithread	q	sym
curkthread	kthread	queue	symbol
d	ktrace	quit	t
debug	mbuf	reg	tcp
defkthread	mem	region	tcpcb
defk	memory	report	tlb
defproc	mempool	rnode	tlbdump
die	nm	runq	trace
dis	nodepda	s	type
dump	od	sbe	un
eframe	outfile	search	unpcb
etrace	p	sema	vertex
eval	page	sh	vfs
f	pager	sizeof	vnode
file	pd	slpproc	vproc
findsym	pda	snode	vsocket
from	pde	soc	vtop
fs	pfdat	socket	w
fstype	pid	stack	walk
fsym	po	stat	whatis
func	preg	sthread	zone
h	pregion	str	
help	print	strace	
hinv	printd	stream	

A.3.1 Commands Added Since 6.2/6.3 Version

config	
curkthread	moved from defpro
defkthread	moved from defslot
hwpath	
ithread	
memory	
nodepda	
pid	
sthread	
vertex	
vproc	
vsocket	
walk	

A.3.2 Commands Removed Since 6.2/6.3 Version

curproc	changed to curkthread
defslot	changed to defkthread
lastproc	obsolete
mbstat	currently not working
pfdathash	currently not working
pfind	currently not working
uipcvn	obsolete
user	obsolete

A.3.3 Command Functionality Changed Since 6.2/6.3 Version

hinv	now just points to config and memory
struct	new functionality