IRIX® Checkpoint and Restart Operation Guide

007-3236-009

CONTRIBUTORS

Written by Bill Tuthill, Karen Johnson, and Terry Schultz

Production by Glen Traefald

Engineering contributions by Brent Casavant, Dean Roe, and Elwira Karwowski

COPYRIGHT

©1996-1998, 2000, 2002-2003 Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc

LIMITED RIGHTS LEGEND

The electronic (software) version of this document was developed at private expense; if acquired under an agreement with the USA government or any contractor thereto, it is acquired as "commercial computer software" subject to the provisions of its applicable license agreement, as specified in (a) 48 CFR 12.212 of the FAR; or, if acquired for Department of Defense units, (b) 48 CFR 227-7202 of the DoD FAR Supplement; or sections succeeding thereto. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy 2E, Mountain View, CA 94043-1351.

TRADEMARKS AND ATTRIBUTIONS

Silicon Graphics, SGI, the SGI logo, and IRIX are registered trademarks and ChallengeArray, Power ChallengeArray, and Trusted IRIX are trademarks of Silicon Graphics, Inc., in the United States and/or other countries worldwide.

Portions of the IRIX Checkpoint and Restart code are derived from MD5 Message-Digest Algorithm of RSA Data Security, Inc.

FLEXlm is a trademark of GLOBEtrotter Software, Inc. POSIX is a registered trademark of IEEE. R10000 is a trademark of MIPS Technologies, Inc. used under license by Silicon Graphics, Inc. UNIX and the X device are registered trademarks of The Open Group in the United States and other contries.

Cover Design By Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

New Features in This Guide

This rewrite of the *IRIX Checkpoint and Restart Operation Guide* supports the 6.5.22 release of the IRIX operating system.

New Features Documented

None for this release.

Major Documentation Changes

The IRIX 6.5.22 release adds information about using the REPLACE keyword in "FILE Policy" on page 12.

Updated front matter and made miscellaneous editing and formatting changes.

Record of Revision

Version	Description
001	November 1996 Initial revision
002	December 1997 Second revision
003	June 1998 Third revision
004	April 2000 Updated for the IRIX 6.5.8 release
005	May 2002 Updated for the IRIX 6.5.16 release
006	November 2002 Updated for the IRIX 6.5.18 release
007	May 2003 Updated for the IRIX 6.5.20 release
008	August 2003 Updated for the IRIX 6.5.21 release
009	November 2003 Updated for the IRIX 6.5.22 release

Contents

New Features in This Guide . . . iii

Record of Revision . . . v

Figures . . . xiii

Tables . . . xv

Examples . . . xvii

About This Guide . . . xix
Intended Audience . . . xix
What This Guide Contains . . . xix
Resources for Further Information . . . xx
Related Publications . . . xx
Obtaining Publications . . . xx
Conventions . . . xxi
Reader Comments . . . xxii

1. Using Checkpoint and Restart . . . 1
What Is IRIX Checkpoint and Restart . . . 2
Definition of Terms . . . 2
Verifying CPR Installation . . . 4
Checkpointing Processes . . . 4
- Naming the Checkpoint Image . . . 5
- Job Control Shells . . . 5

Restarting Processes . . . 6
- Persistence of Statefiles . . . 7
- Job Control Option . . . 7
- Memory Migration Option . . . 7

Querying Checkpoint Status . . . 8

Deleting Statefiles . . . 8
Graphical Interface–`cview` . . . 8
Checkpoint and Restart Attributes . . . 11
FILE Policy . . . 12
WILL Policy . . . 13
CDIR or RDIR Policy . . . 13
FORK Policy . . . 14
PLACEMENT Policy . . . 14
Example Attribute File . . . 15

2. Administering Checkpoint and Restart . . . 17
Responsibilities of the Administrator . . . 17
Installing CPR . . . 18
Managing Checkpoint Images . . . 19
Statefile Location and Content . . . 19
Monitoring a Checkpoint . . . 19
Removing Statefiles . . . 19
Disabling User Checkpoints . . . 20
Checkpointable Objects . . . 20
Non-Checkpointable Objects . . . 21
Troubleshooting . . . 22
Failure to Checkpoint . . . 22
Failure to Restart . . . 23

3. Programming Checkpoint and Restart . . . 25
Design of Checkpoint and Restart . . . 26
POSIX Compliance . . . 26
IRIX Extensions . . . 26

Programming Issues . . . 26
CPR Library Interfaces . . . 27
SIGCKPT and SIGRESTART . . . 27
Adding Event Handlers . . . 28
Preparing for Checkpoint . . . 29
Handling a Checkpoint . . . 29
Checkpoint Time-outs . . . 30
Handling a Restart . . . 31
Checkpoint and Restart of System Objects . . . 32
Checkpoint-Safe Objects . . . 32
Supported Process Groupings . . . 32
User Memory . . . 32
System States in Kernel . . . 32
System Calls . . . 33
Signals . . . 33
Open Files and Devices . . . 33
Open Pipes . . . 34
Shared Memory and Semaphores . . . 34
Application Licensing . . . 34
Network Applications Using Array Services . . . 35
Other Supported Command . . . 35
Compatibility Between Releases . . . 35
Limitations and Caveats . . . 36
SVR4 Semaphores and Messages . . . 36
Networking Socket Connections . . . 36
Other Special Devices . . . 36
Graphics . . . 36
Miscellaneous Restrictions . . . 37
Saving State Using `ckpt_create()` . . . 37
Resuming Using `ckpt_restart()` . . . 39
Checking Status Using `ckpt_stat()` . . . 40
Removing Checkpoints Using `ckpt_remove()` . . . 41
Preparing Checkpoints Using `ckpt_setup()` . . . 41

Using CPR in Trusted IRIX . . . 41
Restoring Trusted IRIX Attributes for Processes . . . 42
Restoring Trusted IRIX Attributes for Files, Pipes, and Named Pipes . . . 42
Restoring Trusted IRIX Attributes for System V Shared Memory Segments . . . 43
Restrictions When Using CPR on a Trusted IRIX System . . . 43
A. Online Help . . . 45
Overview . . . 45
How to Checkpoint. . . . 45
How to Restart . . . 46
Querying a Statefile . . . 46
Deleting a Statefile . . . 46
Checkpoint Widgets . . . 47
Step I Button . . . 47
User Drop Pocket . . . 47
User Name . . . 47
User Recycle . . . 47
Process List . . . 47
Process Types . . . 48
Statefile Field . . . 48
Statefile Drop Pocket . . . 48
Statefile Name . . . 49
System Upgrade . . . 49
Step II Button . . . 49
Exit or Continue . . . 49
File Dispositions . . . 49
Open File List . . . 50
OK Button . . . 50
Cancel Button . . . 50
Tab Controls . . . 50

Restart Widgets 51
List Button 51
Finder Drop Pocket 51
Finder Pathname 51
Finder Recycle 51
Statefile List 51
Process ID Menu 51
Original Working Directory 52
Original Root Directory 52
Restart Button 52
Tell Me More Button 52
Remove Statefile Button 52

Index 53

Figures

Figure 1-1 Checkpoint Control Panel (`cview`) 9
Figure 1-2 Restart Control Panel (`cview`) 10

Tables

Table 1-1 IDtype Modifier Options 11
Table 1-2 Policy Names and Actions. 12
Table 2-1 CPR Product Subsystems 18
Table 2-2 Checkpoint Failure Messages 22
Table 2-3 Restart Failure Messages 23

Examples

Example 3-1 Checkpoint and Restart Event Handling 29
Example 3-2 Routine to Handle Checkpoint 30
Example 3-3 Setting an Alarm in Callback 30
Example 3-4 Routine to Handle Restart 31
Example 3-5 Sample Usage of the `ckpt_create()` Function 37
Example 3-6 Sample Usage of the `ckpt_restart()` Function 39
Example 3-7 Sample Usage of the `ckpt_stat()` Function 40
Example 3-8 Sample Usage of the `ckpt_remove()` Function 41
Example 3-9 Implementation of the `ckpt_setup()` Function 41

About This Guide

IRIX Checkpoint and Restart (IRIX CPR) is a facility for saving the state of running processes, and for later resuming execution where it left off. Based on the POSIX 1003.1m standard, this facility was initially implemented in IRIX release 6.4.

This *IRIX Checkpoint and Restart Operation Guide* describes how to use and administer IRIX CPR, and how to program checkpointing applications.

Intended Audience

This document is intended for anyone who needs to checkpoint and restart processes, including users, administrators, and application programmers.

What This Guide Contains

Here is an overview of the material in this book:

- Chapter 1, “Using Checkpoint and Restart,” explains how to checkpoint and restart a process, and how to set CPR control options.
- Chapter 2, “Administering Checkpoint and Restart,” describes how to install and administer CPR, and how to configure state files.
- Chapter 3, “Programming Checkpoint and Restart,” talks about how to program checkpoints into applications.
- Appendix A, “Online Help”, describes the help screens accesible through the `cview` window’s Help menu.

Resources for Further Information

The cpr(1) man page describes the usage and options of the cpr command. The ckpt_create(3) man page documents the CPR programming interface; ckpt_setup(3), ckpt_restart, ckpt_stat(3), and ckpt_remove(3) are links to the same page.

The atcheckpoint(3c) man page describes how to set up checkpoint and restart event handlers; atrestart(3c) is a link to that page.

Related Publications

The following documents contain additional information that may be helpful:

- *IRIX Admin: Software Installation and Licensing*—Explains how to install and license software that runs under the IRIX operating system, the SGI implementation of the UNIX operating system. Contains instructions for performing miniroot and live installations using the inst command. Identifies the licensing products that control access to restricted applications running under IRIX and refers readers to licensing product documentation.
- *Message Passing Toolkit: PVM Programmer's Manual*—Documents the Message Passing Toolkit for IRIX (MPT) 1.5 implementation of PVM-3 supported on SGI MIPS based systems running IRIX release 6.5 or later. No technical changes have been made for MPT 1.5.
- *IRIX Admin: Resource Administration*— Provides an introduction to system resource administration and describes how to use and administer various IRIX resource management features, such as IRIX process limits, IRIX job limits, the Miser Batch Processing System, the Cpuset System, Comprehensive System Accounting (CSA), IRIX memory usage, and Array Services.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing man <title> on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
GUI element	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy, M/S 535
  Mountain View, California 94043-1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Using Checkpoint and Restart

This chapter introduces the IRIX Checkpoint and Restart (IRIX CPR) facility. It describes how to checkpoint and restart a process, and how to set IRIX CPR control options.

This chapter contains the following sections:

- "What Is IRIX Checkpoint and Restart" on page 2
- "Definition of Terms" on page 2
- "Checkpointing Processes" on page 4
- "Restarting Processes" on page 6
- "Querying Checkpoint Status" on page 8
- "Deleting Statefiles" on page 8
- "Graphical Interface–cview" on page 8
- "Checkpoint and Restart Attributes" on page 11

What Is IRIX Checkpoint and Restart

IRIX Checkpoint and Restart (CPR) is a facility for saving a running process or set of processes and, at some later time, restarting the saved process or processes from the point already reached, without starting all over again. The checkpoint image is saved in a set of disk files, and restarted by reading the saved state from these files to resume execution.

The `cpr` command provides a command-line interface for checkpointing, restarting checkpointed processes, checking the status of checkpoint and restart operations, and deleting files that contain images of checkpointed processes.

Checkpointing is useful for halting and continuing resource-intensive programs that take a long time to run. IRIX CPR can help when you need to:

- Improve a system's load balancing and scheduling
- Run complex simulation or modeling applications
- Replace hardware for high-availability or fail-safe applications

Processes can continue to run after checkpoint and can be checkpointed multiple times.

The IRIX 6.5.20 release adds support for Trusted IRIX attributes to Checkpoint and Restart (CPR) processes. For more information, see "Using CPR in Trusted IRIX" on page 41.

Definition of Terms

A **statefile** is a directory containing information about a process or set of processes (including the names of open files and system objects). Statefiles contain all available information about a running process, to enable restart. The new process(es) should behave just as if the old process(es) had continued. Statefiles are stored as files inside a directory and are protected by normal IRIX security mechanisms.

A **checkpoint owner** is the owner of all checkpointed processes and the resulting statefiles. Only the checkpoint owner or superuser is permitted to perform a checkpoint. If targeted processes have multiple owners, only the superuser is permitted to checkpoint them. Only the checkpoint owner or superuser can restart checkpointed process(es) from a statefile. If the superuser performed a checkpoint, only the superuser can restart it.

A **process group** is a set of processes that constitute a logical job—they share the same process group ID. For example, modern UNIX shells arrange pipelined programs into a process group, so they all can be suspended and managed with the shell's job control facilities. You can determine the process group ID using the `-j` option of the `ps` command; for more information see the `ps`(1) man page. Programmers can change the process group ID using the `setpgid`( ) system call; for more information see the `setpgid`(2) man page.

A **process session** is a set of processes started from the same physical or logical terminal. Such processes share the same session ID. You can determine the process group ID and the session ID (SID) of any process by using the `-j` option to the `ps` command; for more information see the `ps`(1) man page. Programmers can change the session ID using the `setsid`( ) system call; for more information, see the `setsid`(2) man page.

An **IRIX array session** is a set of conceptually related processes running on different nodes in an array. Support is provided by the array services daemon, which knows about array configuration and provides functions for describing and administering the processes of a single job. The principal use of array services is to run jobs that are large enough to span two or more machines.

A **process hierarchy** is the set of all child processes with a common parent. The process hierarchy is identified by giving the process ID of the parent process. A process session is one example of a process hierarchy, but by no means the only example.

A **share group** is a group of processes created from a common ancestor by `sproc`( ) system calls; for more information see the `sproc`(2) man page. The `sproc`( ) call is like `fork`( ), except that after `sproc`( ), the new child process can share the virtual address space of the parent process. The parent and child each have their own program counter value and stack pointer, but text and data space are visible to both processes. This provides a mechanism for building parallel programs.

An **IRIX job** is a group of related processes all descended from a point of entry process and identified by a unique job ID. A job can contain multiple process groups, sessions, or array sessions, and all processes in one of these subgroups are always contained within one job. For more information see the `job_limits`(5) man page.

Verifying CPR Installation

To verify that CPR runs on your system, check that the `eoe.sw.cpr` subsystem is installed:

```
$ versions eoe.sw.cpr
I = Installed, R = Removed
   Name                    Date      Description
I  eoe                     09/28/96  IRIX Execution Environment, 6.3
I  eoe.sw                  09/14/96  IRIX Execution Environment Software
I  eoe.sw.cpr              09/14/96  Checkpoint and Restart
```

If no CPR subsystem is installed, see “Installing CPR” on page 18 for instructions on installing CPR.

Checkpointing Processes

To checkpoint a set of processes (one process or more), use the `-c` option of the `cpr` command, providing a statefile name, and specifying a process ID with the `-p` option. For example, to checkpoint process 1111 into statefile `ckptSep7`, enter the following:

```
$ cpr -c ckptSep7 -p 1111
```

To checkpoint all processes in a process group, enter the process group ID (for example, 123) followed by the :GID modifier:

```
$ cpr -c statefile -p 123:GID
```

To checkpoint all processes in a process session, enter the process session ID (for example, 345) followed by the :SID modifier:

```
$ cpr -c statefile -p 345:SID
```

To checkpoint all processes in an IRIX array session, enter the array session ID (for example, 0x8000abcd00001111) followed by the :ASH modifier:

```
$ cpr -c statefile -p 0x8000abcd00001111:ASH
```

To checkpoint all processes in a process hierarchy, enter the parent process ID (for example, 567) followed by the :HID modifier:

```
$ cpr -c statefile -p 567:HID
```

To checkpoint all processes in an `sproc( )` share group, enter the share group ID (for example, 789) followed by the :SGP modifier:

```
$ cpr -c statefile -p 789:SGP
```

To checkpoint all processes in an IRIX job, enter the job ID (for example, 0x8000abcd00001234) followed by the :JID modifier:

```
$ cpr -c statefile -p 0x8000abcd00001234:JID
```

It is possible to combine process designators using the comma separator, as in the following example. All processes are recorded in the same statefile.

```
$ cpr -c ckptSep8 -p 1113,1225,1397:HID
```

The `-w` option specifies that `cpr` use the attribute file located in the current working directory (versus `$HOME/.cpr`).

```
$ cpr -c -w ckptDec13 -p 1113
```

Naming the Checkpoint Image

You can place the statefile anywhere, provided you have write permission for the target directory, and provided there is enough disk space to store the checkpoint images. You might want to include the date as part of the statefile name, or you might want to number statefiles consecutively. The `-f` option of the `cpr` command forces an overwrite of an existing statefile.

Job Control Shells

The C shell (`csh`), Korn shell (`ksh` or, after IRIX 6.3, `sh`), Tops C shell (`tcsh`), and GNU shell (`bash`) all support job control. The Bourne shell (`bsh`, formerly `sh`) does not. Jobs can be suspended with `Ctrl+Z`, backgrounded with the `bg` built-in command, or foregrounded with `fg`. All job control shells provide the `jobs` built-in command with an `-l` option to list process ID numbers and a `-p` option to show the process group ID of a job.

Restarting Processes

To restart a set of processes (one process or more), use the `-r` option of the `cpr` command, providing just the statefile name. For example, to restart the set of processes checkpointed in `ckptSep7`, enter the following:

```
$ cpr -jm -r ckptSep7
```

Use the `-j` option if you want to perform interactive job control after restart. Otherwise, the process group restored belongs to `init`, effectively disabling job control.

Use the `-m` option if you want to migrate the checkpointed memory to the location in the system topology where the restart operation is executing.

```
$ cpr -c -w ckptDec13 -p 1113
```

Use `-w` option if you want to use the attribute file located in the current working directory (versus `$HOME/.cpr`).

You may restart more than one statefile with the same `cpr` command. If a restart involves more than one process, all restarts must succeed before any process is allowed to run; otherwise all restarts fail. Restart failure can occur for any of the following reasons:

unavailable PID
: The original process ID is not available (already in use), and the option to allow ANY process ID was not in effect.

component unavailable
: Application binaries or libraries are no longer available on the system, and neither the REPLACE nor SUBSTITUTE option was in effect.

security and data integrity
: The user lacks proper permission to restart the statefile, or the restart will destroy or replace data without proper authorization. Only the checkpoint owner and the superuser may restart a set of processes.

resource limitation
: System resources such as disk space, memory (swap space), or number of processes allowed, ran out during restart.

file contents change

If the CONTENTS action was used for FILE policies in the user's cpr attribute file, the restart could fail if file contents have changed between checkpoint and the restart. For more information, see "FILE Policy" on page 12.

other fatal failure

Some important part of a process restart failed for unknown reasons.

Persistence of Statefiles

The statefile remains unchanged after restart; cpr does not delete it automatically. To free disk space, use the -D option of cpr; for more information, see the section "Deleting Statefiles" on page 8.

Job Control Option

If a checkpoint is issued against an interactive process or a group of processes rooted at an interactive process, it can be restarted interactively using the with -j option of the cpr command. This option makes processes interactive and job-controllable. The restarted processes run in the foreground, even the original ones ran in the background. Users may issue job control signals to background the process if desired. An interactive job is defined as a process with a controlling terminal; for more information see the termio(7) man page. Only one controlling terminal is restored even if the original process had multiple controlling terminals.

Memory Migration Option

The -m option of the cpr command migrates process memory so it is restored to the location in the system topology where the restart operation is executing, for example, within a specific cpuset, within the global cpuset, and so on. Without this option, the default restart behavior on NUMA systems is to restore process memory back to where it was at the time of the checkpoint. See the migration(3) man page for scenarios that may prevent pages from migrating properly. This option has no effect on non-NUMA systems.

Querying Checkpoint Status

To obtain information about checkpoint status, employ the `-i` option of the `cpr` command, providing the statefile name. You may query more than one statefile at a time. For example, to get information about the set of processes checkpointed in `ckptSep7`, either before or after restart, enter the following command:

```
$ cpr -i ckptSep7
```

This displays information about the statefile revision number, process names, credential information for the processes, the current working directory, open file information, the time when the checkpoint was done, and so forth.

Deleting Statefiles

To delete a statefile and its associated open files and system objects, use the `-D` option of the `cpr` command, providing a statefile name. You may delete more than one statefile at a time. For example, to delete the file `ckptSep7`, enter the following command:

```
$ cpr -D ckptSep7
```

Only the checkpoint owner and the superuser may delete a statefile directory. Once a checkpoint statefile has been deleted, restart is no longer possible.

Graphical Interface—`cview`

The `cview` command brings up a graphical interface for CPR and provides access to some features of the `cpr` command. As of the IRIX 6.5.16 release, new features are no longer being added to the `cview` command or interface. The `cview` command will be removed in the next major release of the IRIX operating system. The checkpoint control panel, shown in Figure 1-1, displays a list of processes that may be checkpointed.

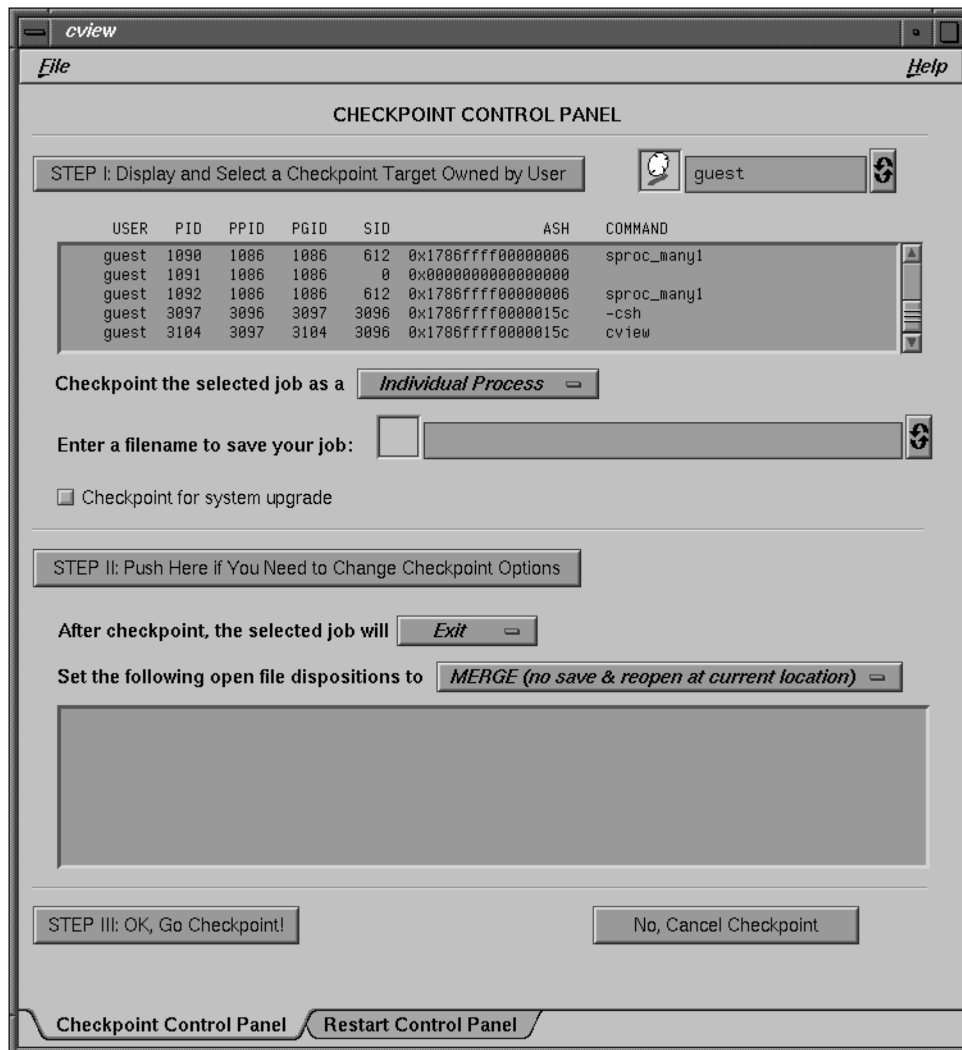

Figure 1-1 Checkpoint Control Panel (`cview`)

Checkpoint options may be set in step II, and are explained in the section "Checkpoint and Restart Attributes." Click the right tab at the bottom to switch panels.

The restart control panel, shown in Figure 1-2, displays a list of statefiles that may be restarted. The buttons near the bottom query checkpoints and delete statefiles.

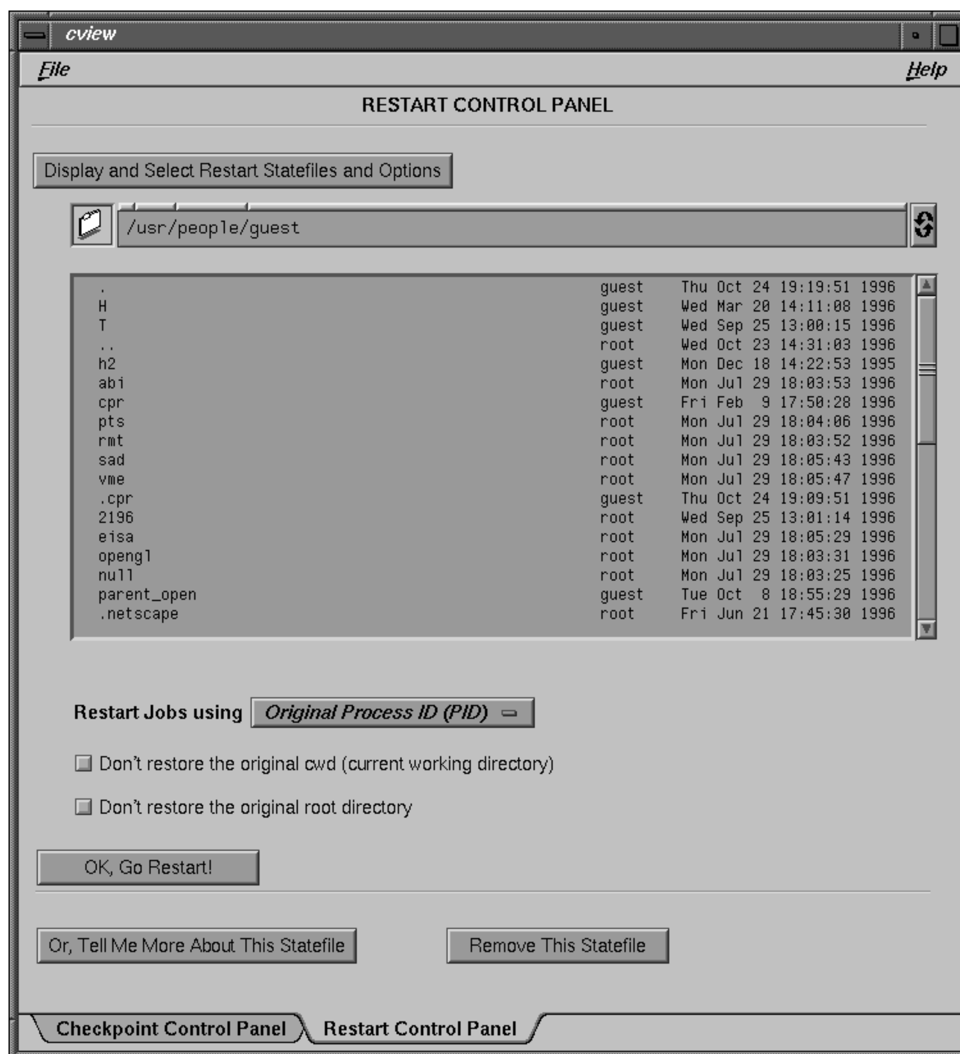

Figure 1-2 Restart Control Panel (`cview`)

Checkpoint and Restart Attributes

The cpr command reads an attribute file at start-up time to set checkpoint configuration and control restart behavior. Typical defaults are given in the /etc/cpr_proto sample file. You can control CPR behavior by creating a similar .cpr attribute file in your home directory (if $HOME is not set, cpr consults the password entry). The CPR attribute file consists of one or more CKPT attribute definitions, each in the following format:

```
CKPT IDtype IDvalue {
    policy:  instance:  action
    ...
}
```

Possible values for *IDtype* are similar to process ID modifiers for the -c option of cpr, and are shown in Table 1-1. *IDvalue* specifies the process ID or process set ID.

Table 1-1 IDtype Modifier Options

IDtype	Process Type Designation
PID	UNIX process ID or POSIX thread ID.
GID	UNIX process group ID; see setpgrp.
SID	UNIX process session ID; see setsid(2).
ASH	IRIX array session ID; see array_sessions(5).
JID	IRIX job ID: see job_limits(5).
HID	Process hierarchy (tree) rooted at the given process ID.
SGP	IRIX sproc() shared group; see sproc(2).
*	A wild card for anything.

The *policy* lines inside the CKPT block specify default actions for CPR to take. Possible values for *policy* are shown in Table 1-2.

Table 1-2 Policy Names and Actions

Policy Name	Domain of Action
FILE	Policies for handling open files.
WILL	Actions on the original process after checkpoint.
CDIR	Policy on the original working directory; see `chdir`(2).
RDIR	Policy on the original root directory; see `chroot`(2).
FORK	Policy on original process ID.
PLACEMENT	Policy on process and memory placement.

FILE Policy

The FILE policy can take an optional *instance* field. This field specifies files that have a unique disposition, other than the default action. For example, in one case you want to replace a file, but in another case you want to append to a file. The *instance* field is enclosed in double quotes and may contain wildcards. For example, `/tmp/*` identifies all files in the `/tmp` directory, and `/*` identifies all files in the system.

The following *action* keywords are available for the FILE policy:

MERGE
: No explicit file save at checkpoint. Upon restart, reopen the file and seek to the previous offset. This is the default file disposition. It may be used for files that are not modified after checkpoint, or for files where it is acceptable to overwrite changes made between checkpoint and restart time, particularly past the saved offset point. If programs seek before writing, changes preceding the offset point could be overwritten as well.

IGNORE
: No explicit file save at checkpoint. Upon restart, reopen the file as it was originally opened, at offset zero (even if originally opened for append). If the file was originally opened for writing, as with the `fopen()` “w” or “a” flag, this action has the effect of overwriting the entire file.

APPEND
: No explicit file save at checkpoint. Upon restart, append to the end of the file. This disposition is good for log files.

REPLACE — Explicitly save the file at checkpoint. Upon restart, replace the original file with the saved one. Any changes made to the original file between checkpoint and restart time are overwritten by the saved file.

For system security, when restarting a `setuid` or `setgid` process, REPLACE actions are changed to SUBSTITUTE actions for files that have been modified or deleted, or a MERGE action for all other files. If a SUBSTITUTE action is performed, a notice specifying the location of the substituted file is displayed. It is the responsibility of the user to pick up any output file thus substituted. Applications that reopen a substituted file by its original name may not operate as expected.

SUBSTITUTE — Explicitly save the file at checkpoint. Upon restart, reopen the saved file as an anonymous substitution for the original file. This is similar to the REPLACE mode except that the original file remains untouched, unless specifically altered by the program.

CONTENTS — Calculate checksum (currently MD5) on the file at checkpoint. Upon restart, detect if the file has been modified between begin-of-file and file-size-at-checkpoint; if the file has been modified in this area, the process is refused restart, otherwise, seek to the previous offset and continue.

WILL Policy

The following *action* keywords are available for the WILL policy:

EXIT — The original process exits after checkpoint. This is the default action.

KILL — Same as above. Has the same effect as the `cpr -k` option.

CONT — The original process continues to run after checkpoint. Has the same effect as the `cpr -g` option.

CDIR or RDIR Policy

The following *action* keywords are available for the CDIR and RDIR policies:

REPLACE — Set the current working directory (CDIR) or the root directory (RDIR) to those of the original process. This is the default action.

IGNORE — Ignore the current working directory (CDIR) or the root directory (RDIR) of the original process, and restart processes based on the current working directory or the root directory of the new process environment.

FORK Policy

The FORK policy can take an optional *instance* field, either PID or JID. If no instance is specified, the specified action is applied to all instances. The following *action* keywords are available for the FORK policy:

ORIGINAL — Do a special `fork()` to recover the original process ID. This is the default action.

ANY — This says it is acceptable for the application to have any process ID as its underlying process if the original process ID is already taken by another running process. In other words, the application itself, internally and in its relationship to other processes, is PID-unaware. If a set of processes is PID-unaware, the ANY action can be specified to avoid PID collisions.

There is no attribute equivalent to the `cpr -u` option for operating system upgrade.

PLACEMENT Policy

At restart, CPR uses node and CPU information to restore memory locations, MLDs, processes, and so on back to locations in the machine topology. The PLACEMENT policy provides some flexibility in restoring the machine topology in cases where the machine configuration has changed. The following *action* keywords are available for the PLACEMENT policy:

STRICT — Upon restart, restore the memory of process(es) according to the placement policies saved at the time of checkpoint. The restart may fail if memory can no longer be placed according to the checkpointed placement policies due to machine configuration changes, lack of available memory, and so on. For more information on process memory placement, see the `mmci`(5) man page.

Upon restart, restore process(es) that were restricted to a specific CPU at the time of checkpoint to the same CPU. The restart may fail if a process can no longer be restricted to that CPU due to machine configuration changes and so on. For more information on CPU restriction, see the `sysmp`(2) and `runon`(1) man pages.

FLEXIBLE Upon restart, if process memory placement fails when adhering to the checkpointed placement policies, attempt to place process memory according to a basic memory placement algorithm (TOPOLOGY_FREE) and print a message stating that this action was taken. For more information on process memory placement, see the mmci(5) man page.

Upon restart, if a process cannot be restricted to the CPU it was restricted to at the time of checkpoint, allow the process to run on any CPU and print a message stating that this action was taken. For more information on CPU restriction, see sysmp(2) and runon(1).

This is the default action.

Example Attribute File

The $HOME/cpr file specifies a user's CPR default attributes. Here is an example of a custom .cpr attribute file:

```
CKPT PID 1111 {
    FILE:   "/tmp/*":  REPLACE
    WILL:              CONT
    FORK:   PID:       ANY
}
```

This saves and restores all /tmp files, allows the process to continue after checkpoint, and permits process ID substitution if needed.

Administering Checkpoint and Restart

This chapter describes how to install and administer IRIX Checkpoint and Restart (CPR), and how to configure statefiles. It contains the following sections:

- "Responsibilities of the Administrator" on page 17
- "Installing CPR" on page 18
- "Managing Checkpoint Images" on page 19
- "Checkpointable Objects" on page 20
- "Non-Checkpointable Objects" on page 21
- "Troubleshooting" on page 22

Responsibilities of the Administrator

The system administrator is responsible for the following CPR tasks:

- Install CPR software on server systems as required
- Help users employ CPR on server systems and workstations
- Prevent statefiles from filling up available disk space
- Delete, or encourage users to delete, unneeded old statefiles

Installing CPR

The subsystems that make up CPR are listed in Table 2-1.

Table 2-1 CPR Product Subsystems

Subsystem Name	Contents
`eoe.sw.cpr`	Checkpoint and restart software
`eoe.man.cpr`	CPR reference manual pages
`eoe.books.cpr`	This guide as an InfoSearch document

If CPR is not already installed, follow this procedure to install the software:

1. Load the IRIX software distribution CD-ROM.
2. On the server, become superuser and invoke the `inst` command, specifying the location of the CD-ROM software distribution:

   ```
   $ /bin/su -
   Password:
   # inst -f /CDROM/dist
   ```

3. Prevent installation of all default subsystems using the `keep` subcommand:

   ```
   Inst> keep *
   ```

   For additional information on `inst`, see the *IRIX Admin: Software Installation and Licensing Guide*, or the `inst`(1M) man page.

4. Make subsystem selections. To install CPR software, the man pages, and the CPR manuals for IRIS InSight, enter the following commands:

   ```
   Inst> install eoe.*.cpr
   Inst> list i
   Inst> go
   ```

   The `list` subcommand with the `i` argument displays all the subsystems marked for installation. The `go` subcommand starts installation, which takes some time.

   For additional information on available subsystems, see the *IRIX Release Notes*.

5. Ensure that the following line exists in the `/var/sysgen/system/irix.sm` file (change `cprstub` to `cpr` if necessary):

   ```
   USE: cpr
   ```

Managing Checkpoint Images

Because of their potential size and longevity, checkpoint images (statefiles) are one aspect of CPR where intervention by the system administrator may be required.

Statefile Location and Content

The statefile can exist anywhere on a filesystem where the user has write permission, provided there is enough disk space to store it. Statefiles tend to be slightly larger than their checkpointed process.

As the system administrator, you might want to create a policy saying that checkpoint images stored in temporary directories (such as `/tmp` or `/var/spool`) are not guaranteed to remain there. If users want to preserve a statefile indefinitely, they should place it in a permanent directory that they own themselves, such as their home directory.

Checkpoint images contain much information about a process, including process set IDs, copies of user data and stack memory, kernel execution states, signal vectors, a list of open files and devices, pipeline setup, shared memory, array job states, and so on.

Monitoring a Checkpoint

To obtain information about a statefile directory, run the `cpr` command with the `-i` option:

$ **cpr -i** *statefile* **...**

This displays information about the statefile revision number, process names, credential information for the processes, the current working directory, open file information, the time when the checkpoint was done, and so forth.

There is no automated way to tell if a user has restarted a statefile or not. You need to ask.

Removing Statefiles

First check with the checkpoint owner to request that they remove unneeded statefiles. If there is no reply, and checkpoints are overflowing disk space, look for the oldest statefiles, especially ones in a series, as the best candidates for removal.

To delete an entire statefile directory, run the `cpr` command with the `-D` option:

```
$ cpr -D statefile ...
```

Only the checkpoint owner and the superuser may delete a statefile. Once a checkpoint has been deleted, it cannot be restarted until the statefile is restored from backups.

Disabling User Checkpoints

If you want to restrict user access to CPR, or if some users abuse the facility by leaving around large statefile directories, you can follow this procedure:

1. Create a “cpr” group in the CPR server’s `/etc/group` file, listing the users who should have access to CPR.

   `cpr::100:`*user1*`,`*user2*`,`*user3*`,`*user4*`,`*user5*`,`*user6*

2. Make the `cpr` command group “cpr” and mode 4750.

   ```
   # chgrp cpr /usr/sbin/cpr
   # chmod 4750 /usr/sbin/cpr
   ```

To temporarily disable CPR, make the `/usr/sbin/cpr` command 000 mode. To permanently shut off CPR, use the `inst` command to remove the `eoe.sw.cpr` subsystem.

Checkpointable Objects

The following system objects are checkpoint safe. See “Checkpoint-Safe Objects” on page 32 for complete coverage of checkpoint safety.

- UNIX processes, process groups, terminal control sessions, IRIX array sessions, process hierarchies, `sproc( )` groups (see the `sproc` (2) man page), and random process sets
- All user memory area, including user stack and data regions
- System states, including process and user information, signal disposition and signal mask, scheduling information, owner credentials, accounting data, resource limits, current directory, root directory, locked memory, and user semaphores
- System calls, if applications handle return values and error numbers correctly, although slow system calls may return partial results

- Undelivered and queued signals are saved at checkpoint and delivered at restart
- Open files (including NFS-mounted files), mapped files, file locks, and inherited file descriptors; this includes open pipes with pipeline data
- Special files /dev/tty, /dev/console, /dev/zero, /dev/null, and ccsync(7M)
- UNIX System V shared memory (but the original shared memory ID is not restored); see the shmop(2) man page
- IRIX jobs; see the job_limits(5) man page
- Jobs started with ChallengArray services, provided they have a unique ASH number; see the array_services(5) man page
- Applications using the prctl( ) PR_ATTACHADDR option; see the prctl(2) man page
- Applications using blockproc( ) and unblockproc( ); see the blockproc(2) man page
- The Power Fortran join synchronization accelerator; see the ccsync(7M) man page
- R10000 counters; see the libperfex(3c) and perfex(1) man pages

Non-Checkpointable Objects

The following system objects are not checkpoint safe. See “Limitations and Caveats” on page 36 for more complete coverage of unsupported system objects.

- Network socket connections; see the socket(2) man page
- X terminals and X11 client sessions
- Special devices such as tape drivers and CD-ROMs
- Files opened with setuid credential that cannot be reestablished
- UNIX System V semaphores and messages (as opposed to System V shared memory); see the semop(2) and msgop(2) man pages

Troubleshooting

This section provides a guide to various error messages that could appear during checkpoint and restart operations, and what these messages might indicate.

Failure to Checkpoint

Checkpointing can fail for any of the reasons shown in Table 2-2.

Table 2-2 Checkpoint Failure Messages

Error Message	Problem Indicated
Permission denied	Search permission denied on a pathname component of statefile.
Resource busy	A resource required by the target process is in use by the system.
Checkpoint error	An uncheckpointable resource is associated with the target process.
File exists	The pathname designated by statefile already exists.
Invalid argument	An invalid argument was passed to a function call.
Too many symbolic links	A symbolic link loop occurred during pathname resolution.
No such file or directory	The pathname to statefile is nonexistent.
Not a directory	A component of the path prefix is not a directory.
Filename too long	The pathname to statefile exceeds the maximum length allowed.
No space left on device	Space remaining on disk is insufficient for the statefile.
Operation not permitted	The calling process does not have appropriate privileges.
Read-only file system	The requested statefile would reside on a read-only filesystem.
No such process	The process or process group specified by ID does not exist.

Failure to Restart

Restart can fail for any of the reasons shown in Table 2-3.

Table 2-3 Restart Failure Messages

Error Message	Problem Indicated
Permission denied	Search permission denied on a path component of statefile.
Resource temporarily unavailable	Total number of processes for user exceeds system limit.
Checkpoint error	An unrestartable resource is associated with target process.
Resource deadlock avoided	Attempted locking of a system resource would have resulted in a deadlock situation.
Invalid argument	An invalid argument was passed to the function call.
Too many symbolic links	A symbolic link loop occurred during pathname resolution.
Filename too long	The pathname to statefile exceeds the maximum length.
No such file or directory	The pathname to statefile is nonexistent.
Not enough space	Restarting the target process requires more memory than allowed by the hardware or by available swap space.
Not a directory	A component of the path prefix is not a directory.
Operation not permitted	The real user ID of the calling process does not match the real user ID of one or more processes recorded in the checkpoint, or the calling process does not have appropriate privileges to restart one or more of the target processes.

Programming Checkpoint and Restart

This chapter describes how to write applications that checkpoint and restart processes gracefully. Code samples are provided, and code fragments at the end of the chapter show sample usage of IRIX CPR library routines.

For applications with checkpoint-unsafe objects, the principal programming concern is setting up event handlers to perform clean-up at checkpoint time and to restore network sockets, graphic state, tape I/O, and CD-ROM status (and so on) at restart time.

This chapter contains the following sections:

- "Design of Checkpoint and Restart" on page 26
- "Programming Issues" on page 26
- "Checkpoint and Restart of System Objects" on page 32
- "Saving State Using ckpt_create()" on page 37
- "Resuming Using ckpt_restart()" on page 39
- "Checking Status Using ckpt_stat()" on page 40
- "Removing Checkpoints Using ckpt_remove()" on page 41
- "Preparing Checkpoints Using ckpt_setup()" on page 41
- "Using CPR in Trusted IRIX" on page 41

Design of Checkpoint and Restart

This section describes some design issues that governed the implementation of CPR.

POSIX Compliance

IRIX Checkpoint and Restart is based on POSIX 1003.1m draft 11, and was initially implemented in IRIX release 6.4. Because POSIX draft standards often change radically from inception to approval, the interfaces in IRIX release 6.5 are not guaranteed to be fully compliant, nor can SGI make any assurance that they will conform to the POSIX 1003.1m standard when it is eventually approved.

IRIX Extensions

The `cpr` command is not specified in POSIX 1003.1m draft 11. It is an IRIX specific command provided for the convenience of customers; see the `cpr`(1) man page. The POSIX draft standard covers only the programming interfaces for checkpoint and restart.

The `ckpt_stat`( ) function, which returns information about the status of checkpoint statefiles, is not specified in POSIX 1003.1m draft 11; see the `ckpt_stat`(3) man page. The `ckpt_setup`( ) function specified in the POSIX draft is unimplemented; when applications call this routine, it is a no-op.

Programming Issues

This section describes the CPR library interfaces and signals, and shows how to write programs that set up event handlers using `atcheckpoint`( ) to prepare for a checkpoint and using `atrestart`( ) to restore non-checkpointable system objects at restart time. See “Limitations and Caveats” on page 36 for a list of non-checkpointable objects.

CPR Library Interfaces

Application interfaces for adding CPR event handlers are contained in the C library, and are listed below. For more information, see the `atcheckpoint`(3C) man page.

- `atcheckpoint()` - add an event handler function for checkpointing
- `atrestart()` - add an event handler function for restarting

The checkpoint and restart library interfaces are contained in the `libcpr.so` dynamic shared object (DSO). When using this library, include the `<ckpt.h>` header file:

```
#include <ckpt.h>
```

The available library routines are listed below. For more information, see the `ckpt_create`(3) man page.

- `ckpt_create()` - checkpoint a process or set of processes into statefiles
- `ckpt_restart()` - resume execution of checkpointed process or process group
- `ckpt_stat()` - retrieve status information about a checkpoint statefile
- `ckpt_remove()` - delete a checkpoint statefile directory
- `ckpt_setup()` - control checkpoint creation attributes (currently a no-op)

In the following discussion, “set of processes” can mean one process or a group of processes.

SIGCKPT and SIGRESTART

When a program (such as the `cpr` command) calls `ckpt_create()` to create a checkpoint, that function sends a SIGCKPT signal to the set of processes specified by the checkpoint ID argument to `ckpt_create()`. Applications add an event handler to catch SIGCKPT if they need to restore non-checkpointable objects such as network sockets, a graphic state, or file pointers to a CD-ROM. The default action is to ignore SIGCKPT.

After sending a SIGCKPT signal, `ckpt_create()` waits for the application to finish its signal handling before CPR proceeds with further checkpoint activities. At restart time, the first thing `ckpt_restart()` runs is the application’s SIGRESTART signal handler, if one exists. This implies that checkpoint and restart can “get stuck” in the SIGCKPT and SIGRESTART handling routines.

When a program calls `ckpt_restart()` to resume execution from a checkpoint, the restart function sends a SIGRESTART signal to the set of processes checkpointed in the statefile specified by the *path* argument to `ckpt_restart()`. Applications add an event handler to catch SIGRESTART if they need to restore non-checkpointable objects such as sockets, a graphic state, or CD-ROM files. The default action is to ignore SIGRESTART.

Adding Event Handlers

The SIGCKPT and SIGRESTART signals are not intended to be handled directly by an application. Instead, CPR provides two C library functions that allow applications to establish a list of functions for handling checkpoint and restart events.

The `atcheckpoint()` routine takes one parameter—the name of your application's checkpoint handling function—and adds this function to the list of functions that get called upon receipt of SIGCKPT. Similarly, the `atrestart()` routine registers the specified callback function for execution upon receipt of SIGRESTART.

These functions are recommended for use during initialization when applications expect to be checkpointed but contain checkpoint-unsafe objects. An application may register multiple checkpoint event handlers to be called when checkpoint occurs, and multiple restart event handlers to be called when restart occurs.

At checkpoint time and at restart time, registered functions are called in the same order as the first-in-first-out order of their registration with `atcheckpoint()` or `atrestart()`, respectively. This is an important consideration for applications that need to register multiple callback handlers for checkpoint or restart events.

Use of `atcheckpoint()` and `atrestart()` ensures that registered signal handlers are invoked only when a checkpoint or restart of the application is in progress (as opposed to the user sending the signals directly via a function such as `sigsend()`).

Caution: If applications catch the SIGCKPT and SIGRESTART signals directly, it could undo all of the automatic CPR signal handler registration provided by `atcheckpoint()` and `atrestart()`, including CPR signal handlers that some libraries may reserve without the application programmer's knowledge.

Preparing for Checkpoint

If an application needs to restore network sockets, graphic state, tape I/O, CD-ROM mounts, or some other non-checkpointable system object, it should set up automatic checkpoint and restart event handlers using the recommended library routines.

The following sample code calls `atcheckpoint()` and `atrestart()` to set up functions for handling checkpoint and restart events. It is possible for this setup to fail on operating systems that do not (yet) support CPR.

Example 3-1 Checkpoint and Restart Event Handling

```
#include <stdlib.h>
#include <ckpt.h>

extern void ckptSocket(void);
extern void ckptXserver(void);
extern void restartSocket(void);
extern void restartXserver(void);

main(int argc, char *argv[])
{
    int err = 0;

    if ((atcheckpoint(ckptSocket) == -1) ||
        (atcheckpoint(ckptXserver) == -1) ||
        (atrestart(restartSocket) == -1) ||
        (atrestart(restartXserver) == -1))
            perror("Cannot setup checkpoint and restart handling");
    /*
     *  processing ...
     */
    exit(0);
}
```

Handling a Checkpoint

Suppose your program mounts an ISO 9660 format CD-ROM, from which it reads data as a basis for more complex processing. Since the CD-ROM is not a checkpointable object, your program needs to record the file pointer position, close all open files on CD-ROM, and perhaps unmount the CD-ROM device.

The following sample code marks the current file position in the open *cdFile*, saves it for restoration at restart time, closes *cdFile*, and unmounts the CD-ROM.

Example 3-2 Routine to Handle Checkpoint

```
#include <sys/types.h>
#include <sys/mount.h>
#include <stdio.h>

extern char *cdFile;
extern FILE fpCD;
long cdOffset;

catchCKPT()
{
    cdOffset = ftell(fpCD);
    fclose(fpCD);
    umount("/CDROM");
    exit(0);
}
```

Note: The checkpoint event handler should return directly to its calling routine—it must not contain any `sigsetjmp()` or `siglongjmp()` code.

Checkpoint Time-outs

For programs that must wait for some external condition before exiting the checkpoint event handling function, it might be wise to set a time-out. For example, if a program is waiting for data to arrive over a TCP socket that must be shut down before checkpoint, and the data never arrives, the program should not wait forever.

The `alarm()` system call sends a SIGALRM signal to the calling program after a specified number of seconds. Since the default action for SIGALRM is for the program to exit, put this call near the top of the checkpoint handling routines to set a 1-minute time-out.

Example 3-3 Setting an Alarm in Callback

```
extern int sock; /* file descriptor for socket */

catchCKPT()
{
    alarm(60);
    close(sock);
    alarm(0);
}
```

Handling a Restart

Suppose your program that unmounted the ISO 9660 CD-ROM at checkpoint time is restarted with the `cpr` command. Now it needs to ensure that the CD-ROM is mounted, reopen the formerly active file, and seek to the previous file offset position. Once it accomplishes all that, your program is ready to continue reading data from the CD-ROM.

The following sample code waits for the CD-ROM to become mounted, then reopens the *cdFile*, and seeks to the remembered offset position in *cdFile*.

Example 3-4 Routine to Handle Restart

```
#include <unistd.h>
#include <stdio.h>

extern char *cdFile;
extern FILE fpCD;
extern long cdOffset;

catchRESTART()
{
    while (access("/CDROM/data", R_OK) == -1) {
        perror("please insert CDROM");
        sleep(60);
    }
    if ((fpCD = fopen(cdFile, "r")) == NULL)
        perror("cannot open cdFile"), exit(1);
    if (fseek(fpCD, cdOffset, SEEK_SET))
        perror("cannot seek to cdOffset"), exit(1);
    /*
     * etc. */
}
```

Note: The restart event handler should return directly to its calling routine—it must not contain any `sigsetjmp()` or `siglongjmp()` code.

Checkpoint and Restart of System Objects

Due to the nature of UNIX process checkpoint and restart, it is hard, if not impossible, to claim that everything that an original process owns or connects with can be restored. The following list defines what is clearly supported (checkpoint safe), and what limitations are known to exist. For items not listed, application writers and customers must decide what is checkpoint-safe.

Checkpoint-Safe Objects

All known checkpoint-safe entities are listed in the following sections.

Supported Process Groupings

CPR works on UNIX processes, process groups, terminal control sessions, array sessions, process hierarchies (trees of processes started from a common ancestor), IRIX jobs, POSIX threads (see the `pthreads`(5) man page), IRIX `sproc`( ) share groups (see the `sproc`(2) man page), and random process sets.

User Memory

All user memory regions are saved and restored, including user stack and data regions. Note that user text, without being saved at checkpoint time, is remapped directly at restart from the application binaries and libraries. However, by using REPLACE as the file disposition default, even user texts can be saved. The saved texts may not replace the originals if the originals are not changed after the checkpoint. Locked memory regions are restored to remain locked at restart.

System States in Kernel

Most of the important kernel states are restored at restart to be identical to the original ones, such as basic process and user information, signal disposition and signal mask, scheduling information, owner credentials, accounting data, resource limits, current working directory, root directory, user semaphores (see the `usnewsema`(3P) man page), and so on.

System Calls

All system calls are checkpoint safe as long as the applications are handling the system call returns and error numbers correctly. Fast system calls are allowed to finish before checkpoint proceeds. Slow system calls are interrupted and may return to the calling routine with partial results. Applications using system calls that can return partial results need to check for and be prepared to deal with partial results. Slow system calls with no results are transparently reissued at restart.

A number of selected system calls are handled individually. The `sleep`( ) system call is reissued for the amount of time remaining at checkpoint time; see the `sleep`(3C) man page. Restart of the `alarm`( ) system call is similar; the remainder of time recorded at checkpoint elapses before it times out; for more information see the `alarm`(2) man page.

Signals

Undelivered signals and queued signals are saved at checkpoint and delivered at restart.

Open Files and Devices

Processes with regular open files or mapped files, including NFS mounted files, can be checkpointed and restarted without many restrictions as long as users choose the correct file disposition in the CPR attribute file, as described in the section "Checkpoint and Restart Attributes" on page 11.

All file locks are also restored at restart. If the file regions that the restarting process needs to lock have already been locked by another process, CPR tries to acquire the locks a few times before it aborts the restart.

Supported special files are:

- `/dev/tty`
- `/dev/console`
- `/dev/zero`
- `/dev/null`
- `ccsync` (see the `ccsync`(7M) man page).

Inherited file descriptors are restored at restart. Applications using R10000 counters through the `/proc` interface are checkpoint safe, provided the `/proc` file descriptor is closed.

Open Pipes

Applications with SVR3 or SVR4 pipes open can be checkpointed and restarted without restrictions. Pipeline data and streams pipe message modes are also saved and restored.

Shared Memory and Semaphores

Applications using SVR4 shared memory can be checkpointed and restarted; for more information see the `shmop`(2) man page. The original shared memory ID (`shmid`) is now restored—this was not the case in the IRIX 6.4 release.

Applications using POSIX semaphores, or shared arena semaphores and locks, can be checkpointed and restarted. For more information, see the `psema`(D3X) or `usinit`(3P) man pages, respectively.

Application Licensing

Applications using node-lock licenses (one license per machine) are generally safe for checkpoint and restart. Applications using floating licenses may be safe for checkpoint and restart, depending on the license library implementation. In IRIX 6.5 and later, the FLEXlm library includes `atcheckpoint()` and `atrestart()` event handlers.

If your license library employs open-and-warm sockets without CPR-aware handlers, you should do one of the following:

- Add `atcheckpoint()` and `atrestart()` event handlers to your application. The `atcheckpoint()` handler should disconnect license checking, and the `atrestart()` handler should reconnect license checking.
- Ask your license software vendor to add similar handlers to their license library.

Network Applications Using Array Services

Jobs started with Power ChallengeArray or ChallengeArray services can be checkpointed and restarted, provided the jobs have a unique ASH (array session handle) number; for more information see the `array_services`(5) man page. Array services jobs may use several methods to generate a new ASH, including calling `newarraysess( )`. For more information, see the `newarraysess`(2) man page.

During an array checkpoint, a checkpoint server is responsible for starting, monitoring, and synchronizing all checkpoint clients running on its different machines based on the given ASH. Statefiles are saved locally on each machine for all processes with the given ASH running on that machine. Restart occurs in a similar fashion, with the restart server synchronizing with all local restart clients to restore all processes on different machines.

An interactive array job with a controlling terminal on a given machine has to be checkpointed and restarted from that very same machine. Otherwise the controlling terminal cannot be restored.

Other Supported Command

Applications using `blockproc( )` and `unblockproc( )` are checkpoint safe; for more information see the `blockproc`(2) man page.

Memory regions added by calling `prctl( )` with the PR_ATTACHADDR argument can be safely checkpointed and restarted. For more information, see the `prctl`(2) man page.

The Power Fortran join synchronization accelerator is checkpoint safe. For more information, see the `ccsync`(7M) man page.

Applications using R10000 counters are checkpoint safe. For more information, see the `libperfex`(3C) or `perfex`(1) man page.

Compatibility Between Releases

A statefile checkpointed in any current release will most likely be able to restart in future releases, owing to the object-oriented architecture of the CPR implementation.

With certain limitations, an object of system functionality available in any current release will be remapped to some new replacement object at restart if the original object becomes obsolete in a future release.

Limitations and Caveats

Various CPR restrictions and warnings are listed in the following sections.

SVR4 Semaphores and Messages

Applications using SVR4 semaphores, or SVR4 messages, cannot be checkpointed and restarted; for more information see the semop(2) or msgop(2) man pages, respectively.

Networking Socket Connections

Generally speaking, an application with open socket connections (see the socket(2) man page) should not be checkpointed and restarted without special CPR-aware signal handling code. An application needs to catch SIGCKPT and SIGRESTART, and run signal handlers to disconnect any open socket before checkpoint, and reconnect the socket after restart.

Since the MPI (message passing interface) library uses sockets for network connections to the array services daemon arrayd, it is generally not possible to checkpoint MPI code. For more information, refer to the *MPI and PVM User's Guide*, or see the mpi(5) man page.

Other Special Devices

Any device or special file not listed in the section "Open Files and Devices" on page 33 as a checkpoint-safe device can be considered not supported for checkpoint and restart. This includes tape, CD-ROM, and other special real or pseudo devices. Again, applications need to close these devices before checkpoint by catching SIGCKPT, and reopen them after restart by catching SIGRESTART.

Graphics

X terminals, and other kinds of graphics terminals, are not supported. Applications with these devices open have to be CPR-aware and do proper clean-up by catching SIGCKPT and SIGRESTART and calling appropriate signal handling routines. (This is similar to how socket connections should be handled.)

Miscellaneous Restrictions

Applications with open directories cannot be properly checkpointed; for more information see the directory(3C) man page.

A potential problem exists with setuid() programs. When restarting resources such as file descriptors, locks acquired with a different (especially higher) privilege may not succeed. For example, a root process may first open some files, and then call setuid(*guest*). If this process is checkpointed after setuid(), the corresponding restart fails because the files opened by root cannot be accessed by guest. Similar restrictions apply for a non-root process' inherited resources, such as file descriptors from a privileged process.

Saving State Using ckpt_create()

The ckpt_create() function checkpoints a process or set of processes into a statefile. The following code shows sample usage of the ckpt_create() function.

Example 3-5 Sample Usage of the ckpt_create() Function

```
#include <ckpt.h>

static int
do_checkpoint(ckpt_id_t id, u_long type, char *pathname)
{
    int rc;

    printf("Checkpointing id %d (type %s) to directory %s\n",
        id, ckpt_type_str(CKPT_REAL_TYPE(type)), pathname);

    if ((rc = ckpt_create(pathname, id, type, 0, 0)) != 0) {
        printf("Failed to checkpoint process %lld\n", id);
        return (rc);
    }
    return (0);
}
```

The global variable *cpr_flags*, defined in `<ckpt.h>`, permits programmers to specify checkpoint-related options. The following flags may be bitwise ORed into *cpr_flags* before a call to `ckpt_create`( ):

CKPT_CHECKPOINT_CONT
: Have checkpoint target processes continue running after this checkpoint is finished. This overrides the default WILL policy, and the WILL policy specified in a user's CPR attribute file.

CKPT_CHECKPOINT_KILL
: Kill checkpoint target processes after this checkpoint is finished. This is the default WILL policy, but overrides a CONT setting in a user's CPR attribute file.

CKPT_CHECKPOINT_UPGRADE
: Use this flag only when issuing a checkpoint immediately before an operating system upgrade. This forces a save of all executable files and DSO libraries used by the current processes, so that target processes can be restarted in an upgraded environment. This flag must be used again if restarted processes are again checkpointed in the new environment.

CKPT_OPENFILE_DISTRIBUTE
: Instead of saving open files under `statefile`, save open files in the same directory where they reside, and assign a unique name to identify them. For example, if a checkpointed process had the `/etc/passwd` file open with this flag set, the open file would be saved in `/etc/passwd.ckpt.pidXXX`. Although security could be a concern, this mode is useful when disk space is at a premium.

Since *cpr_flags* is a process-wide global variable, make sure to reset or clear flags appropriately before a second call to `ckpt_create`( ).

Resuming Using `ckpt_restart()`

The `ckpt_restart()` function resumes execution of a checkpointed process or processes. The following code shows sample usage of the `ckpt_restart()` function.

Example 3-6 Sample Usage of the `ckpt_restart()` Function

```
#include <ckpt.h>

static int
do_restart(char *path)
{
    printf("Restarting processes from directory %s\n", path);
    if (ckpt_restart(path, 0, 0) < 0) {
        printf("Restart %s failed\n", path);
        return (-1);
    }
}
```

The global variable *cpr_flags*, defined in `<ckpt.h>`, permits programmers to specify restart-related options. The following flags may be bitwise ORed into *cpr_flags* before a call to `ckpt_restart()`:

CKPT_RESTART_INTERACTIVE
: Make a process or group of processes interactive (that is, subject to UNIX job control), if the original processes were interactive. The calling process or the calling process' group leader becomes the group leader of restarted processes, but the original process group ID cannot be restored. Without this flag, the default is to restart target processes as an independent process group with the original group ID restored.

CKPT_RESTART_MIGRATE
: Migrate process memory so it is restored to the location in the system topology where the restart operation is executing, for example, within a specific cpuset, within the *global cpuset*, and so on. The global cpuset is the pool of CPUs not assigned to any specific named cpuset. Without this option, the default restart behavior on NUMA systems is to restore process memory back to where it was at the time of the checkpoint. See the `migration`(3) man page for scenarios that may prevent pages from migrating properly. This option has no effect on non-NUMA systems.

Since *cpr_flags* is a process-wide global variable, make sure to reset or clear flags appropriately before a second call to `ckpt_restart()`.

Checking Status Using `ckpt_stat()`

The `ckpt_stat()` function retrieves status information about a checkpoint statefile. The following code shows sample usage of the `ckpt_stat()` function.

Example 3-7 Sample Usage of the `ckpt_stat()` Function

```
#include <ckpt.h>

static int
ckpt_info(char *path)
{
    ckpt_stat_t *sp, *sp_next;
    int rc;

    if ((rc = ckpt_stat(path, &sp)) != 0) {
        printf("Cannot get information on CPR file %s\n", path);
        return (rc);
    }
    printf("\nInformation About Statefile %s (%s):\n",
        path, rev_to_str(sp->cs_revision));
    while (sp) {
        printf(" Process:\t\t%s\n", sp->cs_psargs);
        printf(" PID,PPID:\t\t%d,%d\n", sp->cs_pid, sp->cs_ppid);
        printf(" PGRP,SID:\t\t%d,%d\n", sp->cs_pgrp, sp->cs_sid);
        printf(" Working at dir:\t%s\n", sp->cs_cdir);
        printf(" Num of Openfiles:\t%d\n", sp->cs_nfiles);
        printf(" Checkpointed @\t%s\n", ctime(&sp->cs_stat.st_mtime));
        sp_next = sp->cs_next;
        free(sp);
        sp = sp_next;
    }
    return (0);
}
```

Removing Checkpoints Using ckpt_remove()

The ckpt_remove( ) function deletes a checkpoint statefile directory.

The following code shows sample usage of the ckpt_remove( ) function.

Example 3-8 Sample Usage of the ckpt_remove() Function

```
#include <ckpt.h>

static int
do_remove(char *path)
{
    int rc = 0;

    if ((rc = ckpt_remove(path)) != 0) {
        printf("Remove checkpoint statefile %s failed\n", path);
        return (rc);
    }
}
```

Preparing Checkpoints Using ckpt_setup()

This function, described in the POSIX draft standard, is implemented as a no-op.

The following code shows the current implementation of the ckpt_create( ) function.

Example 3-9 Implementation of the ckpt_setup() Function

```
int ckpt_setup(struct ckpt_args *args[], size_t nargs)
{
    return(0);
}
```

Using CPR in Trusted IRIX

The IRIX 6.5.20 release added support for Trusted IRIX attributes to Checkpoint and Restart (CPR) processes. If enabled, capabilities, mandatory access control (MAC) labels, and access control lists (ACLs) are saved at the time of checkpoint and restored upon restart. Capabilities are saved and restored for processes and files. MAC labels are saved

and restored for processes, files, pipes, named pipes, and System V shared memory segments. ACLs are saved and restored for files and named pipes. For more information about these attributes, see the `capabilities`(4), `DOMINANCE`(5), and `acl`(4) man pages

This section provides detailed information on how these attributes are restored for processes, files, pipes, named pipes, and System V shared memory segments.

Restoring Trusted IRIX Attributes for Processes

The restored Trusted attributes for a process are as follows:

- The capability set of a restarted process is identical to the capability set of that process at the time of checkpoint.
- The MAC label of a restarted process is identical to the MAC label of that process at the time of checkpoint.

Restoring Trusted IRIX Attributes for Files, Pipes, and Named Pipes

The restored Trusted IRIX attributes for files, pipes, and named pipes are as follows:

- When the `REPLACE` action keyword is used, the following occurs:
  - The capabilities associated with a restored file are identical to the capabilities associated with the file at the time of checkpoint.
  - The MAC label associated with a restored file, pipe, or named pipe is identical to the MAC label associated with that file, pipe, or named pipe at the time of checkpoint.
  - The ACL associated with a restored file or named pipe is identical to the ACL associated with that file at the time of checkpoint.
- When the `MERGE`, `IGNORE`, `SUBSTITUTE`, `CONTENTS`, or `APPEND` action keywords are used, the following occurs:
  - The capabilities of files associated with the process being restarted are not affected.
  - The MAC label of files, pipes, or named pipes associated with the process being restarted are not affected.
  - The ACL of files or named pipes associated with the process being restarted are not affected.

Restoring Trusted IRIX Attributes for System V Shared Memory Segments

The restored Trusted IRIX attribute for System V shared memory segments is as follows:

- The MAC label of a System V shared memory segment being restored is identical to the MAC label of the System V shared memory segment at the time of checkpoint.

Restrictions When Using CPR on a Trusted IRIX System

For security reasons, extra restrictions exist when using CPR on a Trusted IRIX system, as follows:

- To checkpoint a process, the MAC label of the user must equal or dominate the MAC label of the process being checkpointed.
- To restart a process, the MAC label of the user must equal or dominate the MAC label of the checkpointed process.
- To restart a checkpointed process which has any effective, permitted, or inheritable capabilities, both of the following must be true:
  – The effective capability set of the checkpointed process must be a (possibly empty) subset of the union of the current effective and permitted capability sets of the on-disk executable.
  – The permitted and inheritable capability set of the checkpointed process must be a (possibly empty) subset of the permitted and inheritable capability set, respectively, of the on-disk executable.
- To restart a checkpointed process which has no effective, permitted, or inheritable capabilities, both of the following must be true:
  – The effective capability set of the checkpointed process must be a (possibly empty) subset of the union of the current effective and permitted capability sets of the process owner.
  – The permitted and inheritable capability set of the checkpointed process must be a (possibly empty) subset of the current permitted and inheritable capability set, respectively, of the process owner.
- To restore the contents of an existing file with an ACL, the ACL for the file at restart must include the user associated with the process being restored.
- To restore the contents of a file with a MAC label, the MAC label of the file at the time of the restart must equal the MAC label of the file at the time of the checkpoint.

- To reattach to an existing System V shared memory segment, the MAC label of the restarting process must equal the MAC label of the System V shared memory segment.

Online Help

This appendix contains help screens accessible from the `cview` window's Help menu.

Overview

IRIX Checkpoint and Restart (CPR) is a facility for saving a running process or set of processes and, at some later time, restarting the saved process(es) from the point already reached. A checkpoint image is saved in a directory, and restarted by reading saved state from this directory to resume execution.

The `cview` window provides a graphical user interface for checkpointing, restarting checkpoints, querying checkpoint status, and deleting statefiles. Two tabs at the bottom of the `cview` window select either the checkpoint or restart control panel.

How to Checkpoint

Under the **STEP I** button, select a process or set of processes from the list. To checkpoint a process group, a session group, an IRIX array session, a process hierarchy, an IRIX job, or an **sproc** shared group, select a category from the **Individual Process drop-down** menu. In the filename field below, enter the name of a directory for storing the statefile.

Click the **STEP II** button if you want to change checkpoint options, such as whether to exit or continue the process, or control open file and mapped file dispositions.

Click the **STEP III** OK button to initiate the checkpoint, or the **Cancel Checkpoint** button to discontinue.

How to Restart

Click the **Restart Control Panel** tab at the bottom of the `cview` window.

From the scrolling list of files and directories, select a statefile to restart. Note that all files and directories are shown, not just statefile directories. If a statefile is located somewhere besides your home directory, change directories using the icon finder at the top.

Select any options you want, such as whether to retain the original process ID, whether to restore the original working directory, or whether to restore the original root directory.

Click the **OK Go Restart** button to initiate restart.

Querying a Statefile

Click the **Restart Control Panel** tab at the bottom of the `cview` window.

From the scrolling list of files and directories, select a statefile to query. Note that all files and directories are shown, not just statefile directories. If a statefile is located somewhere besides your home directory, change directories using the icon finder at the top.

At the bottom of the `cview` window, click the **Tell Me More About This Statefile** button.

Deleting a Statefile

Click the **Restart Control Panel** tab at the bottom of the `cview` window.

From the scrolling list of files and directories, select a statefile to delete. Note that all files and directories are shown, not just statefile directories. If a statefile is located somewhere besides your home directory, change directories using the icon finder at the top.

At the bottom of the `cview` window, click the **Remove This Statefile** button.

Checkpoint Widgets

Step I Button

Click this button to poll the system for processes owned by the user listed on the right.

User Drop Pocket

This is the drop pocket for the process owner.

User Name

This is the text entry field for the process owner. To look at processes for a different user, type a valid user name into this text entry field, and press Enter or click the **STEP I** button.

User Recycle

This is the recycle button for the process owner. Each time you change the process owner, the recycle list grows, providing a shortcut next time you want to switch process owners.

Process List

This is a list of processes on the system owned by the specified user. Column headings indicate the following values:

USER	user name
PID	process ID
PPID	parent process ID
PGID	process group ID
SID	session group ID
JID	IRIX job ID

ASH	IRIX Array Session ID
COMMAND	The command string with arguments

Process Types

This **drop-down** menu controls whether to checkpoint an individual process, a process group, a process session, an IRIX array session, a process hierarchy, or a *sproc* shared group. Be sure to select a set of processes from the list above that can be checkpointed as the process type you select. Process types are as follows:

- UNIX process ID; see `ps`(1)
- UNIX process group ID; see `setpgrp`(2)
- UNIX process session ID; see `setsid`(2)
- IRIX array session ID; see `array_sessions`(5)
- Process hierarchy (tree) rooted at the given process ID
- IRIX `sproc( )` shared group; see `sproc`(2)
- IRIX job ID; see `job_limits`(5)

Statefile Field

A statefile is a directory containing information about a process or set of processes, including the names of open files and system objects. Statefiles contain all available information about a running process, to enable restart. Statefiles are stored as files inside a directory, protected by normal IRIX security mechanisms.

Statefile Drop Pocket

This is the drop pocket for the statefile name. You may drag a directory icon from another desktop application and drop it here.

Statefile Name

This is the text entry field for the statefile name. Enter a pathname here, and a statefile directory will be created in the location you specify.

System Upgrade

Click here if you intend to upgrade operating system software before restarting this set of processes. When you checkpoint for system upgrade, CPR saves not only open file and process states, but also any system commands and libraries that are necessary to restart the statefile accurately.

Step II Button

This section of the checkpoint control panel sets checkpoint attributes, similar to the attributes controlled by the `cpr` command's *`$HOME/.cpr`* file. Click this button to display open files and mapped files in the scrolling list below.

Exit or Continue

This drop-down menu controls whether the selected process(es) exit after checkpointing, or whether they continue running. The default is to exit.

File Dispositions

This **drop-down** menu controls how CPR treats files that the selected process has open at checkpoint time. The five choices are as follows:

MERGE No explicit file save at checkpoint. Upon restart, reopen the file and seek to the previous offset. This may be used for files that are not modified after checkpoint, or for files where it is acceptable to overwrite changes made between checkpoint and restart time, particularly past the saved offset point. If programs seek before writing, changes preceding the offset point could be overwritten as well.

IGNORE No explicit file save at checkpoint. Upon restart, reopen the file as it was originally opened, at offset zero (even if originally opened for append). If the file was originally opened for writing, as with the `fopen()` "w" or "a" flag, this action has the effect of overwriting the entire file.

APPEND No explicit file save at checkpoint. Upon restart, append to the end of the file. This disposition is good for log files.

REPLACE Explicitly save the file at checkpoint. Upon restart, replace the original file with the saved one. Any changes made to the original file between checkpoint and restart time are overwritten by the saved file.

SUBSTITUTE Explicitly save the file at checkpoint. Upon restart, reopen the saved file as an anonymous substitution for the original file. This is similar to the **REPLACE** mode except that the original file remains untouched, unless specifically altered by the program.

Open File List

After you click the Step II button, this list shows all open files for the selected process(es), and the open file dispositions in effect for each file.

OK Button

Click this button to initiate a checkpoint with the options you have selected above.

Cancel Button

Click this button to cancel the checkpoint options you have selected.

Tab Controls

The `cview` window contains two control panels in one: the **Checkpoint Control Panel** and **Restart Control Pane**l. Click one of these tabs to select the control panel you want.

Restart Widgets

List Button

Click this button to list files and directories in the pathname displayed just below.

Finder Drop Pocket

This is the drop pocket for the pathname. You may drag a directory icon from another desktop application and drop it here.

Finder Pathname

This is the text field for the current pathname. To look at files in a different directory, modify the displayed pathname as you wish, and press Enter or click the button above.

Finder Recycle

This is the recycle button for the pathname. Each time you change the pathname, the recycle list grows, providing a shortcut next time you want to switch pathnames.

Statefile List

This is a list of all files in the directory specified in the finder pathname above. File details such as owner and modification time are shown on the right. Select a statefile from this list, or change directories by changing the finder pathname.

Process ID Menu

This drop-down menu controls how processes are forked at restart time. The default is to restart using the original process IDs. If this proves impossible, select the Any Process ID option instead.

Original Working Directory

If you don’t care about the working directory for restarted processes, click the checkbox saying “Don’t restore the original working directory.”

Original Root Directory

If you don’t care about the root directory for restarted processes, click the checkbox saying “Don’t restore the original root directory.”

Restart Button

Click this button to initiate restart from the selected statefile.

Tell Me More Button

Click this button to query the selected statefile for status information.

Remove Statefile Button

Click this button to delete the selected statefile from the filesystem.

Index

A

alarm() system call, 30
ANY action keyword, 14
APPEND action keyword, 12
array services, safe, 35
array session, defined, 3
ASH modifier, 4
atcheckpoint() library routine, 27
atrestart() library routine, 27
attribute file, CPR, 11, 15
audience type, xix

B

blockproc, safe, 35
Bourne shell, 5

C

-c option (checkpoint), 4
C shell, 5
ccsync, safe, 35
CDIR policy action keywords, 13
CDROM checkpointing, 29
CDROM containing IRIX, 18
checkpoint and restart, defined, 2
checkpoint failure messages, 22
checkpoint owner, defined, 2
checkpointable objects, 20
checkpointing processes, 4
checkpoint-safe objects, 32
checkpoint-unsafe objects, 36
CKPT attribute definitions, 11
ckpt_create() library routine, 27, 37
ckpt_remove() library routine, 27, 41
ckpt_restart() library routine, 27, 39
ckpt_setup() library routine, 26, 27, 41
ckpt_stat() library routine, 26, 27, 40
<ckpt.h> header file, 27
compatibility of releases, 35
CONT action keyword, 13
content overview, xix
CONTENTS action keyword, 7, 13
CPR and Trusted IRIX, 42
.cpr attribute file, 11, 15
cpr command, 2
.cpr example, 15
cview command, 8

D

-D option (delete statefile), 8, 20
deleting statefiles, 8
design of checkpoint and restart, 26
devices and files, safe, 33

disabling user checkpoints, 20
DSO libcpr.so, 27

E

eoe.sw.cpr subsystem, 4, 18
/etc/cpr_proto sample file, 11
EXIT action keyword, 13
extensions to CPR in IRIX, 26

F

-f option (force overwrite), 5
failure to checkpoint, reasons, 22
failure to restart, reasons, 6, 23
FILE policy action keywords, 7, 12
files and devices, safe, 33
FLEXIBLE action keyword, 15
FORK policy action keywords, 14

G

-g option (go continue), 13
GID modifier, 4
graphical user interface, `cview`, 8
graphics state, unsafe, 36
group cpr, creating, 20

H

handling a checkpoint, 29
handling a restart, 31
HID modifier, 4
$HOME/.cpr attribute file, 11
$HOME/.cpr example, 15

I

-i option (status information), 8, 19
IDtype modifier options, 11
IDvalue process set ID, 11
IGNORE action keyword, 12, 14
information about statefiles, 19
`inst` command, 18
installing checkpoint and restart, 18
intended audience, xix
IRIX array session, defined, 3
IRIX extensions to CPR, 26
IRIX job, defined, 3
IRIX software distribution CDROM, 18

J

-j option (interactive job control), 6, 7
JID, see job ID, 11
job control option, 7
job control shells, 5
job ID, defined, 11
job, defined, 3

K

-k option (kill process), 13
kernel states, safe, 32
KILL action keyword, 13
Korn shell, 5

L

libcpr.so DSO, 27
library routine atcheckpoint(), 27
library routines ckpt_*, 27

M

-m option (migrate the checkpointed memory), 6
memory migration option, 7
memory, safe, 32
MERGE action keyword, 12
monitoring a checkpoint, 19

N

network sockets, unsafe, 36
non-checkpointable objects, 21

O

ORIGINAL action keyword, 14
overview of contents, xix

P

-p option (process ID), 4
perfex library routines, safe, 35
persistence of statefiles, 7
pipes and pipe data, safe, 34
PLACEMENT policy action keywords, 14
policy names and actions, 12
POSIX 1003.1m standard, 26
Power Fortran join accelerator, safe, 35
PR_ATTACHADDR, safe, 35
prctl, safe, 35
preventing checkpoint usage, 20
process group, defined, 3
process groupings, safe, 32
process hierarchy, defined, 3
process session, defined, 3

Q

querying checkpoint status, 8

R

-r option (restart), 6
R10000 counters, safe, 35
RDIR policy action keywords, 13
release compatibility, 35
removing statefiles, 19
REPLACE action keyword, 13
responsibilities of administrator, 17
restart failure messages, 23
restart failure, reasons, 6
restarting processes, 6
restrictions on using CPR and Trusted IRIX, 43

S

setuid restrictions, 37
SGP modifier, 5
share group, defined, 3
shells and job control, 5
SID modifier, 4
SIGCKPT signal, 27, 29

signals, safe, 33
SIGRESTART signal, 28, 31
socket connections, unsafe, 36
special devices, safe, 33
special devices, unsafe, 36
sproc share group, defined, 3
statefile deletion, 8
statefile location and content, 19
statefile persistence, 7
statefile, defined, 2
status of checkpoint, 8
STRICT action keyword, 14
SUBSTITUTE action keyword, 13
system administrator responsibilities, 17
system calls, safe, 33
System V messages, unsafe, 36
System V semaphores, unsafe, 36
System V shared memory, safe, 34

T

timeouts for checkpointing, 30
troubleshooting checkpoint, 22
troubleshooting restart, 23
Trusted IRIX and CPR, 42

U

-u option (upgrade OS), 14
unblockproc, safe, 35
user memory, safe, 32
users in cpr group, 20
using CPR with Trusted IRIX, 41

W

`-w` option (specify attribute file), 5, 6
WILL policy action keywords, 13