IRIX Checkpoint and Restart™ Operation Guide

Document Number 007-3236-003

CONTRIBUTORS

Written by Bill Tuthill
Engineering contributions by Ken Beck, Jack Jia, and Weibao Wu
St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica

© 1997-1998, Silicon Graphics, Inc.— All Rights Reserved
The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND
Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

OpenGL, Silicon Graphics, and the Silicon Graphics logo are registered trademarks, and CHALLENGEarray, IRIS InSight, IRIX, IRIX Checkpoint and Restart, IRIX CPR, POWER CHALLENGEarray, and XFS are trademarks of Silicon Graphics, Inc.MIPS is a registered trademark, and R10000 is a trademark of MIPS Technologies, Inc.

Cisco is a registered trademark of Cisco Systems, Inc. Informix is a registered trademark of Informix Software, Inc. Motif is a registered trademark of the Open Software Foundation. NFS is a registered trademark of Sun Microsystems, Inc. Oracle is a registered trademark of Oracle Corporation. POSIX is a registered trademark of IEEE, Inc. Sybase is a registered trademark of Sybase, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd. X Window System is a trademark of the Massachusetts Institute of Technology.

IRIX Checkpoint and Restart™ Operation Guide
Document Number 007-3236-003

Contents

List of Figures vii

List of Tables ix

List of Examples xi

About This Guide xiii
Intended Audience xiii
What This Guide Contains xiii
Resources for Further Information xiii
Conventions Used in This Guide xiv

1. Using Checkpoint and Restart 1
Definition of Terms 1
Verifying eoe.sw.cpr 2
Checkpointing Processes 3
Naming the Checkpoint Image 3
Job Control Shells 4
Restarting Processes 4
Persistence of Statefiles 5
Job Control Option 5
Querying Checkpoint Status 5
Deleting Statefiles 5
Graphical Interface—cview 6
Checkpoint and Restart Attributes 8
FILE Policy 9
WILL Policy 9
CDIR or RDIR Policy 10
FORK Policy 10
Example Attribute File 10

2. Administering Checkpoint and Restart 11
Responsibilities of the Administrator 11
Installing eoe.sw.cpr 11
Managing Checkpoint Images 12
Statefile Location and Content 12
Monitoring a Checkpoint 13
Removing Statefiles 13
Disabling User Checkpoints 13
Checkpointable Objects 14
Non-Checkpointable Objects 14
Troubleshooting 15
Failure to Checkpoint 15
Failure to Restart 16

3. Programming Checkpoint and Restart 17
Design of Checkpoint and Restart 17
POSIX Compliance 17
IRIX Extensions 17
Programming Issues 18
CPR Library Interfaces 18
SIGCKPT and SIGRESTART 18
Adding Event Handlers 19
Preparing for Checkpoint 20
Handling a Checkpoint 20
Checkpoint Time-outs 21
Handling a Restart 22

Checkpoint and Restart of System Objects 22
Checkpoint-Safe Objects 23
Supported Process Groupings 23
User Memory 23
System States in Kernel 23
System Calls 23
SIgnals 24
Open Files and Devices 24
Open Pipes 24
Shared Memory and Semaphores 24
Application Licensing 24
Network Applications Using Array Services 25
Other Supported Functionality 25
Compatibility Between Releases 25
Limitations and Caveats 26
SVR4 Semaphores and Messages 26
Networking Socket Connections 26
Other Special Devices 26
Graphics 26
Miscellaneous Restrictions 27
Saving State With ckpt_create() 27
Resuming With ckpt_restart() 28
Checking Status With ckpt_stat() 29
Removing Checkpoints With ckpt_remove() 30
Preparing Checkpoints With ckpt_setup() 30
A. Online Help 31
Overview 31
How to Checkpoint 31
How to Restart 31
Querying a Statefile 32
Deleting a Statefile 32
Index 33

List of Figures

Figure 1-1 Checkpoint Control Panel *(cview)* 6
Figure 1-2 Restart Control Panel *(cview)* 7

List of Tables

Table 1-1 IDtype Modifier Options 8
Table 1-2 Policy Names and Actions 8
Table 2-1 CPR Product Subsystems 11
Table 2-2 Checkpoint Failure Messages 15
Table 2-3 Restart Failure Messages 16

List of Examples

Example 3-1 Checkpoint and Restart Event Handling 20
Example 3-2 Routine to Handle Checkpoint 21
Example 3-3 Setting an Alarm in Callback 21
Example 3-4 Routine to Handle Restart 22
Example 3-5 Sample Usage of ckpt_create() Function 27
Example 3-6 Sample Usage of ckpt_restart() Function 28
Example 3-7 Sample Usage of ckpt_stat() Function 29
Example 3-8 Sample Usage of ckpt_remove() Function 30
Example 3-9 Implementation of ckpt_setup() Function 30

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.

Resources for Further Information

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

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

The internal Web site http://taco.engr.sgi.com/CPR/ is worth visiting for updates about the product. There is no external Web site available yet.

Conventions Used in This Guide

The table below lists typographic conventions used in this guide.

Purpose	Example
Names of shell commands	The *cpr* command is a command-line interface for CPR.
Command-line options	The **-c** option checkpoints a process, and **-r** restarts it.
System calls and library routines	Processes can checkpoint themselves with **ckpt_create()**.
Filenames and pathnames	Statefile attributes are read from the *$HOME/.cpr* file.
User input (variables in italic)	**cpr -c** *statefile* **-p** *processID*
Exact quotes of computer output	`state10-19: Permission denied.`
Titles of manuals	Refer to *IRIX Admin: System Configuration and Operation*.
A term defined in the glossary	A *DSO* (dynamic shared object) is linkable at runtime.

Chapter 1

Using 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 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 modelling applications
- replace hardware for high-availability or failsafe applications

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

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; see ps(1). Programmers can change the process group ID using the **setpgid()** system call; see setpgid(2).

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 giving the **-j** option to the *ps* command; see ps(1). Programmers can change the session ID using the **setsid()** system call; see setsid(2).

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; see sproc(2). 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.

Verifying eoe.sw.cpr

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 eoe.sw.cpr" on page 11 for instructions on installing CPR.

Checkpointing Processes

To checkpoint a set of processes (one process or more), employ 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 111 into statefile *ckptSep7*, type this command:

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

To checkpoint all processes in a process group, type 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, type 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, type 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, type 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, type the share group ID (for example, 789) followed by the :SGP modifier:

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

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
```

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 forces 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), employ the **-r** option of the *cpr* command, providing just the *statefile* name. For example, to restart the set of processes checkpointed in *ckptSep7*, type this command:

```
$ cpr -j -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.

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.

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*; see the section "Deleting Statefiles."

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 with the **-j** option. 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; see termio(7). Only one controlling terminal is restored even if the original process had multiple controlling terminals.

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, type this 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*, type this 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 all features of the *cpr* command. Online help is available. The checkpoint control panel, shown in Figure 1-1, displays a list of processes that may be checkpointed.

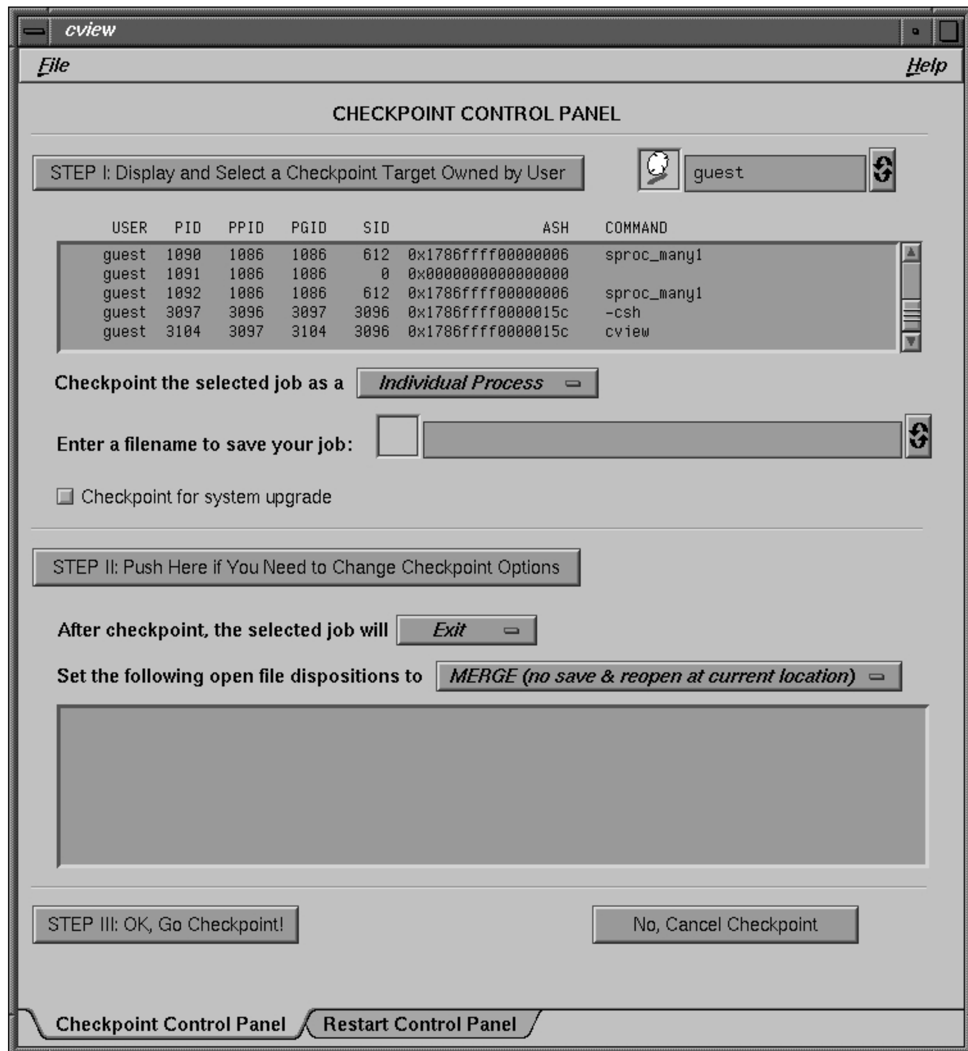

Figure 1-1 Checkpoint Control Panel (*cview*)

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.

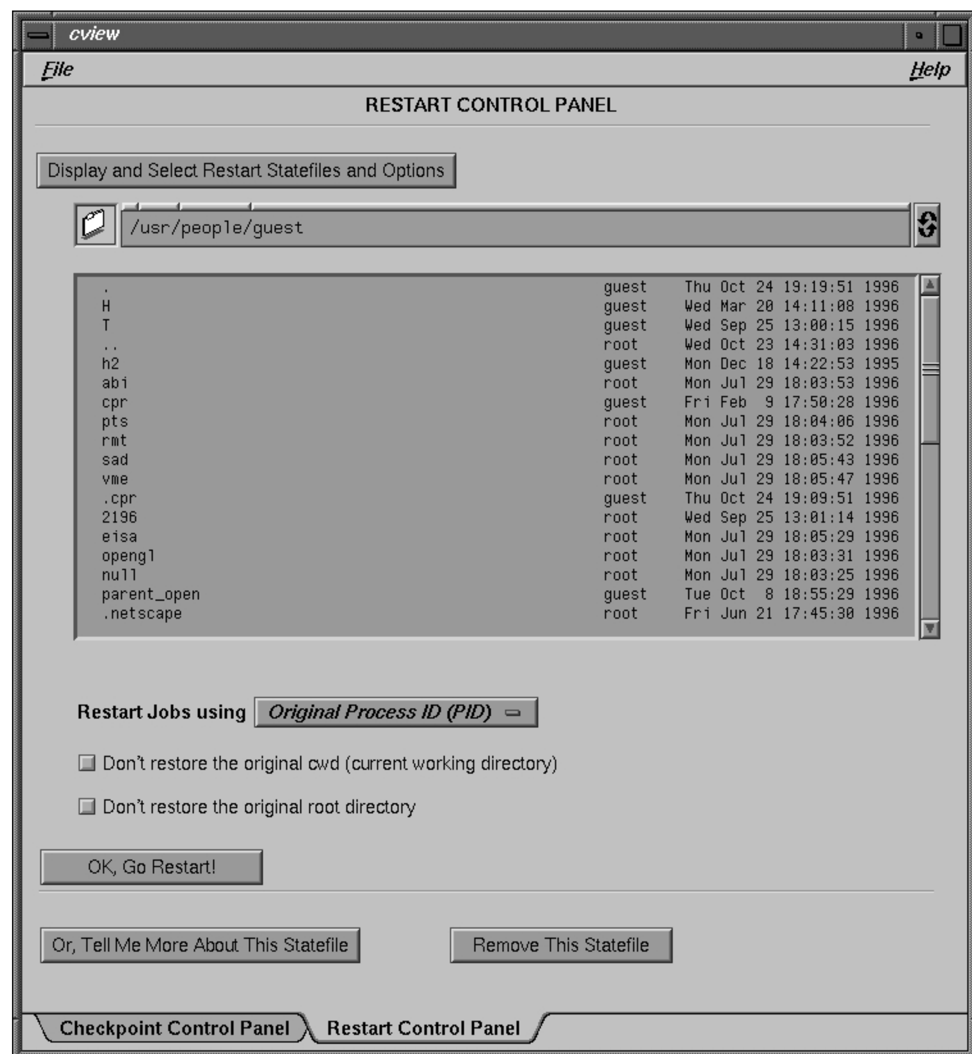

Figure 1-2 Restart Control Panel (*cview*)

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(2).
SID	UNIX process session ID; see setsid(2).
ASH	IRIX array session ID; see array_sessions(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.

FILE Policy

The FILE policy is the only one that takes 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.

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.

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 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.

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                ANY
}
```

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

Chapter 2

Administering Checkpoint and Restart

Responsibilities of the Administrator

IRIX Checkpoint and Restart (CPR) adds the following responsibilities to the system administrator's already long list:

- 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 eoe.sw.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 IRIS InSight 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) reference page.

4. Make subsystem selections. To install CPR software, the reference manual pages, and the CPR manuals for IRIS InSight, enter these 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 23 for complete coverage of checkpoint safety.

- UNIX processes, process groups, terminal control sessions, IRIX array sessions, process hierarchies, **sproc()** groups (see sproc(2)), 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)
- System V shared memory (but the original shared memory ID is not restored); see shmop(2)
- jobs started with CHALLENGEarray services, provided they have a unique ASH number; see array_services(5)
- applications using the **prctl()** PR_ATTACHADDR option; see prctl(2)
- applications using **blockproc()** and **unblockproc()**; see blockproc(2)
- the Power Fortran join synchronization accelerator; see ccsync(7M)
- R10000 counters; see libperfex(3C) and perfex(1)

Non-Checkpointable Objects

The following system objects are not checkpoint safe. See “Limitations and Caveats” on page 26 for more complete coverage of unsupported system objects.

- network socket connections; see socket(2)
- X terminals and X11 client sessions

- special devices such as tape drivers and CD-ROM
- files opened with setuid credential that cannot be reestablished
- System V semaphores and messages (as opposed to System V shared memory); see semop(2) and msgop(2)

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.

Chapter 3

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.

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 Silicon Graphics 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 cpr(1). 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 ckpt_stat(3). 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 26 for a list of non-checkpointable objects.

CPR Library Interfaces

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

- **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 ckpt_create(3).

- **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 discussion below, *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, graphic state, or file pointers to 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 after SIGCKPT. 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, 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.

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 arrive, 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 one-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 below.

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), POSIX threads (see pthreads(5)), IRIX **sproc()** share groups (see sproc(2)), 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 usnewsema(3P)), 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 sleep(3C). Restart of the **alarm()** system call is similar—the remainder of time recorded at checkpoint elapses before it times out; see alarm(2).

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 8.

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*, and ccsync(7M).

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; see shmop(2). 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; see psema(D3X) or usinit(3P), 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; see array_services(5). Array services jobs may use several methods to generate a new ASH, including calling **newarraysess()**; see newarraysess(2).

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 Functionality

Applications using **blockproc()** and **unblockproc()** are checkpoint safe; see blockproc(2).

Memory regions added by calling **prctl()** with the PR_ATTACHADDR argument can be safely checkpointed and restarted; see prctl(2).

The Power Fortran join synchronization accelerator is checkpoint safe; see ccsync(7M).

Applications using R10000 counters are checkpoint safe; see libperfex(3C) or perfex(1).

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 below.

SVR4 Semaphores and Messages

Applications using SVR4 semaphores, or SVR4 messages, cannot be checkpointed and restarted; see semop(2) or msgop(2), respectively.

Networking Socket Connections

Generally speaking, an application with open socket connections (see socket(2)) 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 mpi(5).

Other Special Devices

Any device or special file not listed in section "Open Files and Devices" on page 24 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; see directory(3C).

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 With 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 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 With 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 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 flag 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.

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 With 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 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 With 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 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 With 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 ckpt_setup() Function

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

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, 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.

Index

A

alarm() system call, 21
ANY action keyword, 10
APPEND action keyword, 9
array services, safe, 25
array session, defined, 2
ASH modifier, 3
atcheckpoint() library routine, 18
atrestart() library routine, 18
attribute file, CPR, 8, 10
audience type, xiii

B

blockproc, safe, 25
Bourne shell, 4

C

ccsync, safe, 25
CDIR policy action keywords, 10
CDROM checkpointing, 20
CDROM containing IRIX, 11
checkpointable objects, 14
checkpoint and restart, defined, 1
checkpoint failure messages, 15
checkpointing processes, 3
checkpoint owner, defined, 1
checkpoint-safe objects, 23
checkpoint-unsafe objects, 26
ckpt_create() library routine, 18, 27
ckpt_remove() library routine, 18, 30
ckpt_restart() library routine, 18, 28
ckpt_setup() library routine, 17, 18, 30
ckpt_stat() library routine, 17, 18, 29
CKPT attribute definitions, 8
<ckpt.h> header file, 18
compatibility of releases, 26
CONT action keyword, 9
content overview, xiii
-c option (checkpoint), 3
.cpr attribute file, 8, 10
cpr command, 1
.cpr example, 10
C shell, 4
cview command, 6

D

deleting statefiles, 5
design of checkpoint and restart, 17
devices and files, safe, 24
disabling user checkpoints, 13
-D option (delete statefile), 5, 13
DSO libcpr.so, 18

E

eoe.sw.cpr subsystem, 2, 11
/etc/cpr_proto sample file, 8
EXIT action keyword, 9
extensions to CPR in IRIX, 17

F

failure to checkpoint, reasons, 15
failure to restart, reasons, 4, 16
FILE policy action keywords, 9
files and devices, safe, 24
-f option (force overwrite), 3
FORK policy action keywords, 10

G

GID modifier, 3
-g option (go continue), 9
graphical user interface, *cview*, 6
graphics state, unsafe, 26
group cpr, creating, 13

H

handling a checkpoint, 20
handling a restart, 22
HID modifier, 3
$HOME/.cpr attribute file, 8
$HOME/.cpr example, 10

I

IDtype modifier options, 8
IDvalue process set ID, 8
IGNORE action keyword, 9, 10
information about statefiles, 13
installing checkpoint and restart, 11
inst command, 12
intended audience, xiii
Internet resources, xiv
-i option (status information), 5, 13
IRIX array session, defined, 2
IRIX extensions to CPR, 17
IRIX software distribution CDROM, 11

J

job control shells, 4
-j option (interactive job control), 4, 5

K

kernel states, safe, 23
KILL action keyword, 9
-k option (kill process), 9
Korn shell, 4

L

libcpr.so DSO, 18
library routine atcheckpoint(), 18
library routines ckpt_*, 18

M

memory, safe, 23
MERGE action keyword, 9
monitoring a checkpoint, 13

N

network sockets, unsafe, 26
non-checkpointable objects, 14

O

ORIGINAL action keyword, 10
overview of contents, xiii

P

perfex library routines, safe, 25
persistence of statefiles, 5
pipes and pipe data, safe, 24
policy names and actions, 8
-p option (process ID), 3
POSIX 1003.1m standard, 17
Power Fortran join accelerator, safe, 25
PR_ATTACHADDR, safe, 25
prctl, safe, 25
preventing checkpoint usage, 13
process group, defined, 2
process groupings, safe, 23
process hierarchy, defined, 2
process session, defined, 2

Q

querying checkpoint status, 5

R

R10000 counters, safe, 25
RDIR policy action keywords, 10
release compatibility, 26
removing statefiles, 13
REPLACE action keyword, 9, 10
responsibilities of administrator, 11
restart failure, reasons, 4
restart failure messages, 16
restarting processes, 4
-r option (restart), 4

S

setuid restrictions, 27
SGP modifier, 3
share group, defined, 2
shells and job control, 4
SID modifier, 3
SIGCKPT signal, 18, 20
signals, safe, 24
SIGRESTART signal, 19, 22
socket connections, unsafe, 26
special devices, safe, 24
special devices, unsafe, 26
sproc share group, defined, 2
statefile, defined, 1
statefile deletion, 5
statefile location and content, 12
statefile persistence, 5

status of checkpoint, 5
SUBSTITUTE action keyword, 9
system administrator responsibilities, 11
system calls, safe, 23
System V messages, unsafe, 26
System V semaphores, unsafe, 26
System V shared memory, safe, 24

T

timeouts for checkpointing, 21
troubleshooting checkpoint, 15
troubleshooting restart, 16
typographic conventions, xiv

U

unblockproc, safe, 25
-u option (upgrade OS), 10
user memory, safe, 23
users in cpr group, 13

W

Web pages about PCP, xiv
WILL policy action keywords, 9

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3236-003.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389