CONVEX Checkpoint Restart Guide

Order No. DSW-350

First Edition
November, 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX
Checkpoint Restart Guide

Order No. DSW-350
Part No. 710-006530-001

Copyright 1990 CONVEX Computer Corporation.
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation (CONVEX) does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

Unless provided otherwise in writing with CONVEX Computer Corporation (CONVEX), the program described herein is provided as is without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Some states do not allow the exclusion of implied warranties. The above exclusion may not be applicable to all purchasers because warranty rights can vary from state to state. In no event will CONVEX be liable to anyone for special, collateral, incidental or consequential damages, including any lost profits or lost savings, arising out of the use or inability to use this program. CONVEX will not be liable even if it has been notified of the possibility of such damage by the purchaser or any third party.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.

COVUE is a trademark of CONVEX Computer Corporation. COVUE products consist of COVUEbatch, COVUEbinary, COVUEedt, COVUElib, COVUEnet, and COVUEshell.

UNIX is a trademark of AT&T Bell Laboratories.
Printed in the United States of America

Revision Information for

CONVEX Checkpoint Restart Guide

Edition	Document No.	Description
1st	710-006530-001	Released with CONVEXOS V9.0 November, 1990.

Contents

Checkpoint Restart Overview 1-1

What is Checkpoint Restart? 1-1
  Utilities 1-2
  Library Routines 1-2
  Checkpoint Restart and CXbatch 1-2

Limitations 1-3
  Performance 1-3
  Uncheckpointable Processes 1-3
  Files Used by Checkpointed Process 1-3
  pid Conflict 1-4
  Compatibility 1-4
  CR and Processes Using the Tape System 1-4
  Accounting 1-5

What Happens During Checkpointing 1-6
  Checkpointing Process Hierarchies 1-6
  Files, Pipes, and Devices 1-7
  Access Permissions 1-7

The Checkpoint File 1-9
  Checkpoint File Name 1-9
    Default Checkpoint File Name 1-9
    Specifying a Checkpoint file Name 1-10
  Checkpoint File Format 1-10
  Checkpoint File Size 1-10

What Happens During Restart 1-12
  File Access During Restart 1-12
  Restarting on a Different System 1-12
  User ID and Group ID of Restarted Processes 1-12
    Restoring the Target Process uid, gid, and Group Access Lists 1-12
    File Permission and Communication Problems after Changed uid or gid 1-13
  Parent Process ID (ppid) of Target Process 1-13
  Current Working Directory 1-13
  Restarting Under Share 1-13
  Job Control 1-14
    Control Terminals 1-14
    Process Groups and Terminal Control 1-14
    Passing Signals Through 1-15

The Checkpoint and Restart Utilities 2-1

The chkpnt Command 2-1
chkpnt Parameter Summary 2-1
Using the chkpnt Command 2-2
Save Open Files 2-2
Specify Checkpoint File Name and Checkpoint Directory 2-3
Force Checkpointing 2-4
Checkpoint in Interactive Mode 2-4
Checkpoint Entire Process Hierarchy 2-5
Send Signal to Target 2-5
Diagnostics 2-5

Checkpoint Examples 2-7
Shell Command Line Mode 2-7
Interactive Mode 2-8
Invoking chkpnt in Interactive Mode 2-8
Interactive Mode Commands 2-9
Print Information about Processes 2-11
Checkpointing a Process Hierarchy in Interactive Mode 2-12
Creating and Using Checkpoint Log Files 2-13

The restart Command 2-15
restart Parameter Summary 2-15
Using the restart Command 2-15
Send Signals to Target 2-16
Restart in Interactive Mode 2-16
Copy Back Files 2-16
Force Restart 2-17
Wait/Don't Wait 2-17
Diagnostics 2-17

Restart Examples 2-18
Shell Command Line Mode 2-18
Interactive Mode 2-18
Invoking restart in Interactive Mode 2-18
Interactive restart Commands 2-19
Print Information About a Process 2-21
Restarting a Process Hierarchy in Interactive Mode 2-21

Checkpoint Restart Programming Interface 3-1

Checkpoint C Function 3-1
chkpnt() Format and Parameters 3-1
chkpnt() Return Values and Error Codes 3-3

Restart C Function 3-4
restart() Format and Parameters 3-4

Fortran CR Functions 3-6
Fortran Checkpoint Function 3-6
Format and Parameters 3-6
Fortran Restart Function 3-6
Format and Parameters 3-7

Programming Guidelines 3-8

Device Interface 3-9
CR Device Driver ioctls 3-9

ConvexOS Devices That Support Checkpoint Restart 3-9
Example of Custom Device Driver Code 3-10

C Programming Example 3-12
The Checkpoint Call 3-15
The Restart Call 3-15

Fortran Programming Example 3-16

Appendix A: Error Messages

Appendix B: Reporting Problems

Index

Using This Book

Purpose and Audience

The *ConvexOS Checkpoint Restart Guide* describes the Checkpoint Restart (CR) capability of ConvexOS. This guide includes the following chapters:

1. "Checkpoint Restart Overview"—Gives a general overview of what Checkpoint Restart is and how it works.
2. "The Checkpoint and Restart Utilities"—Describes the interactive ConvexOS utilities used to checkpoint and restart processes.
3. "Checkpoint Restart Programming Interface"—Describes the ConvexOS C and Fortran library functions that checkpoint and restart processes.

The intended audience of this guide includes:

- Users who employ CR to checkpoint and restart application processes.
- Developers who wish to incorporate CR library functions into their programs.
- System managers who want a general understanding of what CR is and how it works.

Technical Assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384.
- All other locations, contact the local CONVEX office.

Associated Documents

In addition to this guide, you may find the following related CONVEX publications helpful:

- *Convex UNIX Primer* (DSW-133). This book introduces new users to the ConvexOS operating system.
- *ConvexOS Programmer's Reference* (DSW-332). This book is the standard reference for the ConvexOS operating system.
- CONVEX FORTRAN Language Reference Manual (DSW-037). This book defines the basic components of FORTRAN, including CONVEX extensions to the language.
- CONVEX FORTRAN User's Guide (DSW-038). This book describes the functions and operations of CONVEX FORTRAN for users of the ConvexOS operating system.

Ordering Documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Include the order number or the exact title, as listed on the front cover.

Notational Conventions

The following notational conventions are used in this guide.

Command Syntax

Consider this example:

`command` *input_file* `[...]` `{a | b}` `[`*output_file*`]`
① ② ③ ④ ⑤

1. **`command`** (in bold constant-width font) must be typed as it appears.
2. *input_file* (italics) indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets `[...]` indicates that additional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. *output_file* indicates an optional file name.

General Conventions

In general, the following conventions are used in this guide:

- **`Bold constant-width font`** identifies what must be typed by the users (e.g. in examples of commands).
- *Italics*
  - Designate user-supplied variables in a command-line example.
  - Introduce new and important terms.
  - Identify variables in mathematical equations.
  - Indicate titles of documents.
- `Constant-width font` is used to designate input and output, including:
  - Command names and options.
  - System calls.
  - Data structures and types.
  - Directives, program statements, display examples, printout examples, and error messages returned.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipses show that lines of code have been left out of an example.

- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press **RETURN**.
- References to the *ConvexOS Programmer's Reference* appear in the form *adb*(1), where the name of the man page is followed by its section number enclosed in parentheses.

Caution

A Caution highlights procedures or information necessary to avoid damage to equipment, software, or data.

Reader Response

If you have comments or questions about the contents of this book, please notify the CONVEX documentation department by using the reader's comment form at the end of this document.

Checkpoint Restart Overview

1

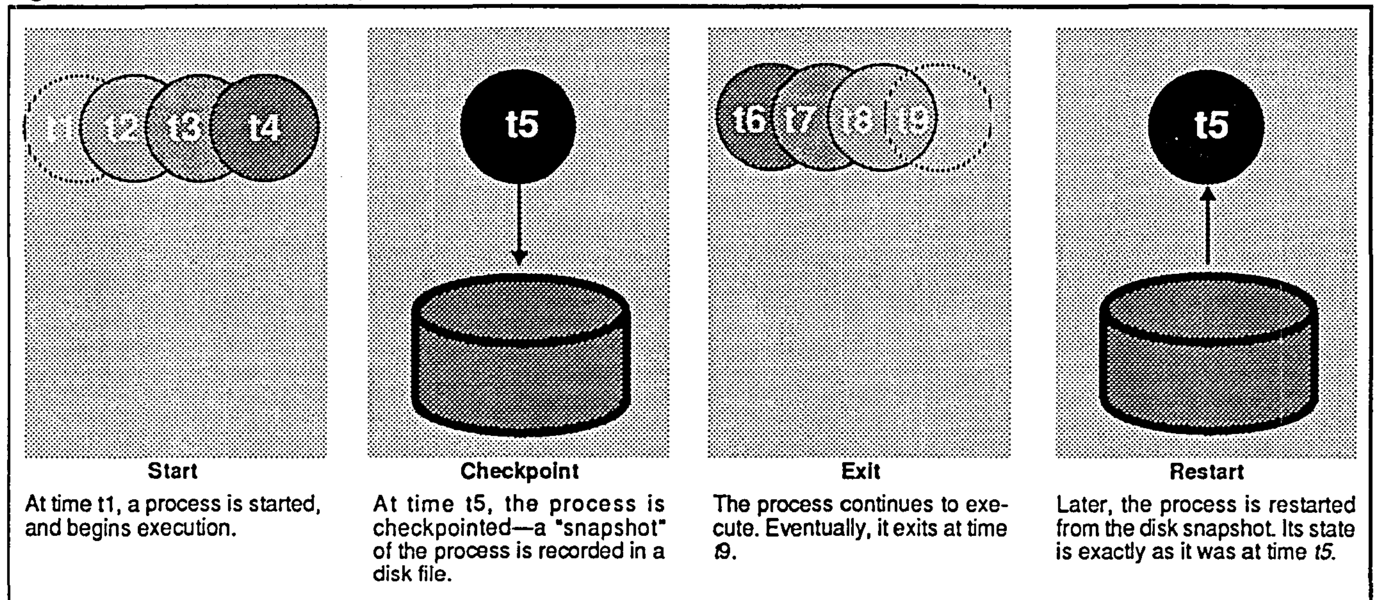
What is Checkpoint Restart?

Checkpoint Restart is a standard feature of the CONVEX Operating System, ConvexOS. This feature permits the state of selected processes or process hierarchies to be saved to disk files, and later to be restarted from the saved files.

Checkpoint Restart (CR) is especially useful for application programs that must run for a long time (e.g., complex simulations), and that—once halted—cannot be started again from the beginning without wasting significant time and resources. Such applications can be saved to files ("checkpointed") either by operator intervention or by a periodically executed script that includes CR commands. If the application is then halted (perhaps by a system crash, by a scheduled system shutdown, or by the operator because computing resources were needed for other tasks), it can—at a convenient time—be restarted as it was when it was last checkpointed. The process can be restarted from the same file as often as desired.

Figure 1-1 below shows the basic principles of checkpointing and restarting a process.

Figure 1-1: Basics of Checkpoint and Restart



Because related processes (processes linked through parent-child relationships) can be checkpointed automatically as a group, entire process hierarchies can be restored. This is useful in restoring all processes related to a certain job, or spawned by a particular shell.

The CR capability includes utilities and library routines that allow checkpointing and restarting processes either interactively, in shell scripts, through CXbatch, or in C or Fortran programs.

Utilities

Two utilities (shell commands) allow you to use the Checkpoint Restart capability by entering interactive commands at the ConvexOS prompt, or in ConvexOS shell scripts:

- `chkpnt`—checkpoints processes (saves their state to a "checkpoint" file).
- `restart`—restarts processes from checkpoint files.

Refer to Chapter 2, "The Checkpoint and Restart Utilities" and the *chkpnt*(1) and *restart*(1) man pages in the *ConvexOS Programmer's Reference* for information about these two utilities.

Library Routines

Four library routines provide the main programmatic interface to CR.

The following two C library routines checkpoint and restart processes:

- `chkpnt()` checkpoints processes.
- `restart()`—restarts processes.

Two Fortran functions of the same name are also provided, and do the same thing as the C functions.

Refer to Chapter 3, "Checkpoint Restart Programming Interface" for information about the CR library routines. These routines are also described in the *chkpnt*(3), *chkpnt(3f)*, *restart*(3) and *restart(3f)* pages in the *ConvexOS Programmer's Reference.*

Checkpoint Restart and CXbatch

The CXbatch system is a set of utilities that allow you to control scheduling, queueing and execution of processes in "batch" mode. Processes that are under control of CXbatch can be checkpointed and restarted by using CR features built into the CXbatch utilities. The following CXbatch utilities have CR capabilities:

- `qchkpnt`—This utility allows you to checkpoint any process being run by CXbatch. You may specify an interval at which this process will be periodically checkpointed.
- `qmgr`—The CXbatch queue management program provides commands that allow you to checkpoint processes being run through CXbatch and requeue them so that they will later be restarted.
- `qrestart`—This CXbatch utility allows the user or batch administrator to restart previously checkpointed CXbatch requests.

For more information about the CR capabilities of CXbatch, refer to the chapters on Checkpoint Restart in *CXbatch System Manager's Guide* and *CXbatch User's Guide.*

Limitations

This section describes the limitations of the Checkpoint Restart capability.

Performance

Since it may take several seconds to checkpoint any given process (very large processes may take several minutes to checkpoint), CR will perform best if the intervals at which processes are checkpointed is measured in hours, not minutes or seconds.

Checkpoint Restart is not designed to save and restart the entire system (i.e., to "roll-out" and "roll-in" all processes running on a machine at any one time).

Uncheckpointable Processes

Not all types of processes or process hierarchies can be checkpointed and then restarted. Processes *cannot* be checkpointed and restarted if they:

- Have a socket connection other than a pipe.
- Are debugging another process or contain a process file descriptor (this includes debuggers such as `adb` or `csd`, as well as utilities such as `ps`, `pstat`, `syspic`, `uptime`, `w`, and `chkpnt` itself).
- Mapped shared memory segments into their address space using the MAP_DEVICE option of `mmap`.
- Use virtual memory addresses in the range from 0xB000000 to 0xBFFFFFFF.
- Have more than 250 open file descriptors.
- Are communicating with another process via a common file that has been removed with the `unlink()` system call.[1]
- Have more than 58 memory segments.
- Use any device other than tty, pty, tape, or /dev/null.
- Have used `vfork()` to spawn a child, and the child has not yet performed an `exec()`.
- Are using labeled tape I/O.
- Have unlinked a file that was mapped into memory with the `mmap` system call. (Refer to the *mmap(2)* man page in the *ConvexOS Programmer's Reference* for more information on this system call.)

In addition, process hierarchies with more than 250 members cannot be checkpointed.

Files Used by Checkpointed Process

Only file descriptors of files that are open to the target process when it is checkpointed are saved in the checkpoint file. If a process opens and closes a file before being checkpointed, no information about the closed file is saved in the checkpoint file.

By default, data contained in files open to the target process is *not* saved by the checkpoint process. However, the files can be copied to the checkpoint directory by using the -C option with the `chkpnt` utility.

1. *Single* processes that read or write to an unlinked file can be checkpointed and restarted. For example, a process might establish a temporary file that does not need to be "cleaned up" when the process exits by first opening the file and then using `unlink ()` to remove its directory entry. The process could then still read and write to the file (by using the file descriptor obtained from the `open()` call), but the file would not exist as far as the file system is concerned, and thus would not have to be removed when the process exits.

If -C is used, the files will be copied to the checkpoint directory with names having the form *com.pid.filename.fd*, where *com* is the command name of the target process that is being checkpointed, *pid* is the process ID of the target process, *filename* is the name of the file, and *fd* is the file descriptor that the target process obtained when it opened the file. (If the file descriptor references a file that has been unlinked, a hyphen (-) will be used instead of a file name.)

Files copied to the checkpoint directory with the -C option will not be replaced in their original directories or opened automatically when the target process is restarted. To use these files, you must copy them back to their original location before restarting the process.

pid Conflict

When a process is restarted, it is, by default, given the same pid as it had when it was checkpointed. Under certain rare circumstances, the restart of a process may fail because its process ID (pid) is the same as that of another process already on the system.

For example, suppose that a process with pid 4004 is checkpointed and then killed (e.g., because of a system shutdown). Subsequently, a new process is created in the system, and—by chance—this process is assigned a pid of 4004. Any attempt to restart the checkpointed process will now fail because two processes with the same pid cannot exist on the system at the same time.

You can force *restart* to ignore a failed pid assignment and restart the target process with the next available (unused) pid. However, additional problems may arise as a result; for example, interprocess communication may fail if other members of the restarted process hierarchy expect the process to have its old pid.

Compatibility

The following compatibility requirements must be considered when restarting processes in a hardware or operating system environment different from the one under which they were checkpointed:

- Restarting a process cannot be guaranteed if the process was checkpointed under a release of ConvexOS that is not the same as the release under which it is being restarted.
- If a process is to be restarted on a different machine than the one on which it was checkpointed, the hardware architecture of the two machines must be compatible.

  In general, if the executable is portable between the two architectures (i.e., it will run on both machines), then the process can be checkpointed on one and restarted on the other. These are some examples of restrictions that are imposed by hardware architecture:

  - A process that was checkpointed on a CONVEX C2 machine that contains C2-specific or parallel instructions cannot be successfully restarted on a CONVEX C1. (If a process checkpointed on a C2 is executable on a C1, then it can be restarted on a C1.)
  - An executable that requires IEEE floating-point format will not restart on a machine that does not have the appropriate hardware to support that format.

CR and Processes Using the Tape System

Processes that access tapes in unlabeled mode can be checkpointed and restarted. If a target process has a tape device open, the current position (file number and record number) of that device will be saved in the checkpoint file. When the process is re-

started, the same tape device is opened, and the device is asked to restore the tape to the saved position.

Several restrictions and cautions must be observed when checkpointing and restarting processes that access tape devices:

- Processes using labeled tapes cannot be checkpointed. If an attempt is made to checkpoint a process that has a tape open in labeled mode, the checkpoint will fail.
- When you restart a process that accesses a tape, you must first place the correct tape *on the same drive as before* (no warning will be given if the wrong tape or drive are used).
- Before a process that uses a tape device is restarted, the same tape device must again be mounted with a `tpmount` command.
- The `chkpnt` process will store the current position in the tape *as it is known to the tape driver*. If the tape is repositioned manually while the tape device is open, the driver will be ignorant of this fact. Consequently, if the process using the tape is checkpointed after the tape is moved manually, the true tape position will not be recorded in the checkpoint file. When the process is restarted, the tape will be positioned to the point last known to the tape driver.
- Only information about tapes on *open* devices is recorded in the checkpoint file. Consequently, if a process that issues tape movement commands is checkpointed after it has closed the tape device and before it opens it again for further access, no information about the position of the tape will be preserved in the checkpoint file. If the process is restarted and the tape has been removed or repositioned, the tape cannot be returned to the correct position. (Utilities such as `cp` or `cat` that write to the tape open and then close the tape device. Thus, a process that performs multiple transfers of files to tape via `cp` may fail to run properly if checkpointed and restarted.)
- If the target process has backspaced over an end-of-file mark on the tape and has written data, it cannot be checkpointed (the checkpoint will fail) until the process has written another tape mark or moved past a tape mark.

Accounting

Checkpointing and restarting processes does not affect the accuracy of statistics kept by the ConvexOS accounting system—these statistics will always reflect the actual resources consumed by each process. When a process exits, accurate information is written to the accounting files regardless of whether or not the process was checkpointed or restarted.

For example, if a process that normally runs for 10 CPU seconds is checkpointed after executing for 6 CPU seconds and then runs to completion, the accounting record will state that the process consumed 10 CPU seconds. If the process is later restarted from the checkpoint file and then runs to completion, the accounting record for the *restarted* process will state that it consumed 4 CPU seconds.

However, the process itself cannot tell whether or not it has been restarted by accessing accounting information. If the restarted process in the example above obtains resource usage information via a `getrusage()` call, the returned information will be the same as though the process had been executing normally from the beginning, instead of being restarted from a checkpoint file. Thus, if the process makes the call immediately upon being restarted, it will be "deceived" into thinking that it has consumed 6+ seconds of CPU time.

What Happens During Checkpointing

When a process is being checkpointed, its execution is first stopped. This is done with the `pattach` system call. (Refer to *pattach*(2) in the *ConvexOS Programmer's Reference* for more information about this call.) Using the information returned by `pattach` and other system calls, a checkpoint file is written to disk for each process that is checkpointed. (Refer to "The Checkpoint File" section on page 9 of this chapter for more information about this file.) This file contains all the information necessary to restart the process in the same condition as it was when it was checkpointed. The following information is recorded in the file for each checkpointed process:

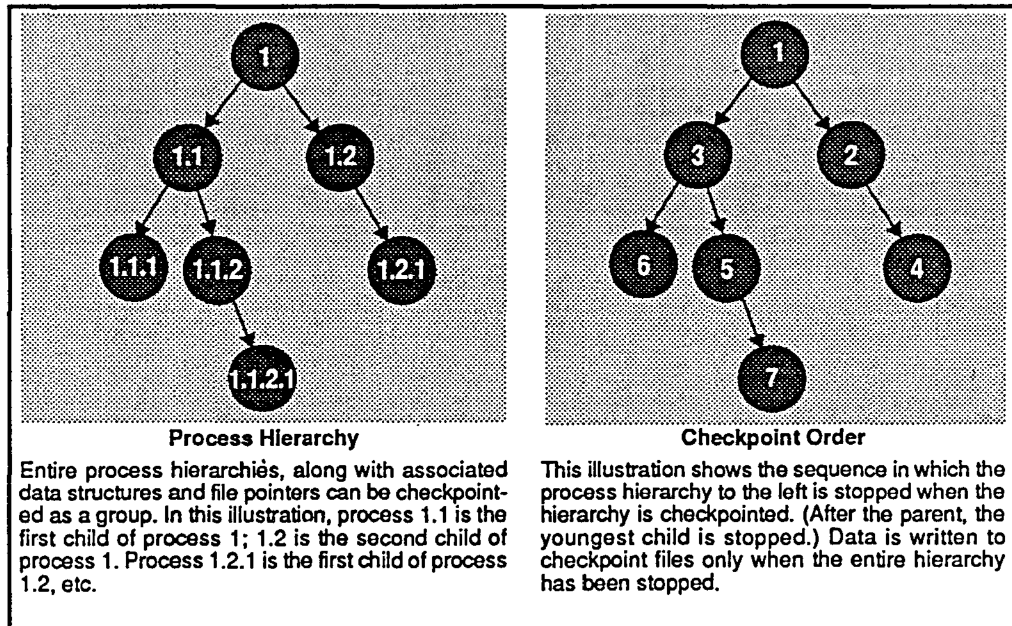
- Register contents.
- File descriptors (for files open to the process).
- Contents of private and shared memory regions used by the process.
- *proc* structure.
- *user* structure.
- *thread* structure.

Checkpointing Process Hierarchies

Instead of a single process, an entire process hierarchy (i.e., a group of processes having a common origin) may be checkpointed and subsequently restarted as a group. This cannot be done for unrelated processes—such processes must be checkpointed and restarted individually.

If a process hierarchy is checkpointed, every process in the hierarchy will be stopped by `pattach` before the first one is checkpointed. Processes are stopped by traversing the "family tree" of the hierarchy: the parent is stopped first, and then its children are stopped (youngest child first). After all the children are stopped, the children of the youngest child are stopped (again, youngest child first), etc. Figure 1-2 illustrates this relationship.

Figure 1-2: Checkpointing Process Hierarchies



Process Hierarchy

Entire process hierarchies, along with associated data structures and file pointers can be checkpointed as a group. In this illustration, process 1.1 is the first child of process 1; 1.2 is the second child of process 1. Process 1.2.1 is the first child of process 1.2, etc.

Checkpoint Order

This illustration shows the sequence in which the process hierarchy to the left is stopped when the hierarchy is checkpointed. (After the parent, the youngest child is stopped.) Data is written to checkpoint files only when the entire hierarchy has been stopped.

After all processes in the hierarchy have been stopped, a checkpoint file is written for each process.

Caution

The chkpnt utility or library function will not overwrite existing checkpoint files with new ones having the same name. If the same process must be checkpointed more than once, specify a new file name.

Any subtree (i.e., any set of branches beginning at a tree node) of a process hierarchy may be restarted independently. For example, in Figure 1-2 above, processes 1.1, 1.1.1, 1.1.2, and 1.1.2.1 are members of a subtree. Be sure that no members of an independently restarted subtree need resources outside the subtree. For instance, if process 1.1 uses a pipe to process 1.2, a restart of only the subtree beginning at 1.1 will fail. In this case, you should restart the entire process hierarchy, beginning with 1.

If the chkpnt process attempts to checkpoint itself (this can occur while checkpointing a process hierarchy that happens to include the chkpnt process), it will create a checkpoint file for itself that will be restarted as a "zombie" process with an exit code of zero. Thus, if the parent process of chkpnt is waiting for chkpnt to exit, the parent will not wait forever after it is restarted.

Refer to Chapter 2, "The Checkpoint and Restart Utilities" and Chapter 3, "Checkpoint Restart Overview" for specific information about the commands and options required to checkpoint processes and process hierarchies.

Files, Pipes, and Devices

Each target process has a file descriptor (unique identifier) for every file that has been opened by the process. Since all ConvexOS input and output is handled through files, these file descriptors refer not only to "regular" files, but also to entities such as pipes and devices (e.g., ttys or tape drives).

Because CR is designed to restore the state of every restarted process completely (if possible), information about the file descriptors of checkpointed processes is stored in the checkpoint file. Thus, information about any regular files, device files, pipes or named pipes that have been opened by the checkpointed process will be recorded, and can be restored when the process is restarted. Any data that is in transit through a pipe will be stored and replaced in the pipe when the hierarchy is restarted.

Caution

The contents of files open to a checkpointed process are not automatically saved (this can only be done by explicitly requesting this action with the chkpnt command). Ordinarily, only the file pathname and current position for each open file is saved in the checkpoint file—the file data is not saved.

If a checkpointed process reads or writes data to a regular file, that file must be accessible under the same path name when the process is restarted; in addition, the file should not have been changed since the process was checkpointed. If the modification timestamp of such a file indicates that the file has been changed since the process was checkpointed, a warning will be given (unless you deliberately override this safeguard with the -w option of `restart`).

Access Permissions

A number of restrictions are enforced to prevent the CR system from being used to bypass system security. These restrictions include the following:

- When a process is checkpointed, the chkpnt process must have permission to access the target process. This means that the chkpnt process (and, therefore, the user that starts it) must be running as root, or must have the same effective uid as the target process. (See also "User ID and Group ID of Restarted Processes" section on page 12.)

- If files that are being used by the target process are to be copied as part of the checkpoint operation, the `chkpnt` process must have "read" permission to those files.
- The target process must have "read" permissions for the executable file. (This prevents users from checkpointing processes and then extracting privileged information contained in the executable—e.g. passwords—from the checkpoint file.)
- The `chkpnt` process must be a member of the group having the current gid of the target process.

The Checkpoint File

When a process is checkpointed, all the information required to restart that process is stored in a checkpoint disk file.

Checkpoint File Name

You may either allow the `chkpnt` utility to assign a default name to the checkpoint files of target processes, or you may specify a name yourself.

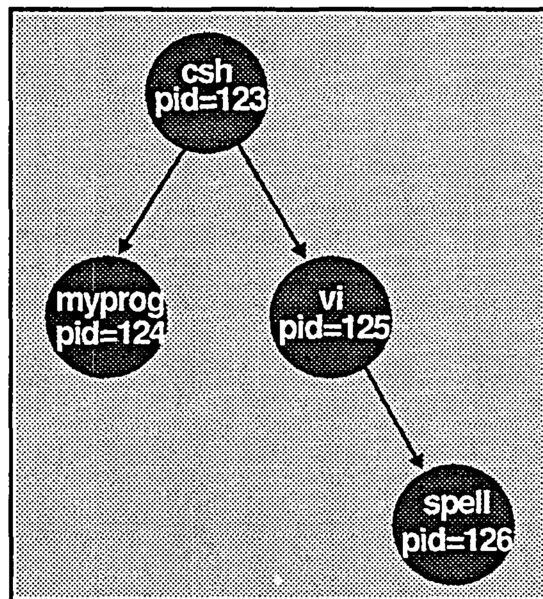
Default Checkpoint File Name

By default, the name of this file is composed of the command name (up to 16 characters) and the pid of the process. (As explained below, you may specify a name for the checkpoint file instead.) For example, the default checkpoint file name of a `csh` process with a pid of 14768 would be *csh.14768*.

If a process hierarchy is being checkpointed, the name of the checkpoint file for each member of the hierarchy will have the form *<lineage>.<program_name>.<pid>*, where *<lineage>* is a list of the program names of all the processes in the hierarchy beginning from the root process down to the parent of the checkpointed process; each member of the list is separated by a period. If *lineage* is so long that the length of the file name would exceed the maximum length permissible on the system, or the path name of the checkpoint file would exceed the maximum permissible length for directory names, then *lineage* will be abbreviated to the name of the root process followed by three periods (...).

To illustrate this, suppose that the process hierarchy in Figure 1-3 (below) is checkpointed. This hierarchy consists of four processes: a `csh` shell with a pid of 123 that has spawned two other processes: `myprog`, with a pid of 124 and `vi` with a pid of 125. The `vi` has, in turn, spawned a `spell` process with a pid of 126.

Figure 1-3 Checkpoint Hierarchy File Names



If you checkpoint this process hierarchy without specifying a checkpoint file name, the following four checkpoint files will be produced:

- csh.123
- csh.myprog.124
- csh.vi.125

- csh.vi.spell.126

If there were a very long chain of processes between `csh` and `spell`, then the checkpoint file would be called:

- csh...spell.126

Specifying a Checkpoint file Name

Instead of using the default described above, you may assign a checkpoint file name yourself by specifying it with the `chkpnt` utility or the `chkpnt()` function. If you are checkpointing a single process and you assign a file name, that name will be used without appending the pid of the target process.

If you are checkpointing a process hierarchy, the name you specify will be assigned to the root process. The remaining processes in the hierarchy will have names of the following form:

<name>.<lineage>.<program_name>.<pid>

<name> is the name you specified, *<lineage>* is a list of the program names of all the processes in the hierarchy up from the parent of the checkpointed process to the child of the root process, *<program_name>* is the program name of the checkpointed process, and *<pid>* is the pid of the checkpointed process. Each member of the list is separated by a period.

For example, if the name "abc" is specified as the checkpoint file name when the hierarchy in Figure 1-3 is checkpointed, then the following checkpoint files will be created:

- abc
- abc.myprog.124
- abc.vi.125
- abc.vi.spell.126

Checkpoint File Format

The checkpoint file conforms to the Standard Object File Format (SOFF), and is similar to a standard ConvexOS core file. (Refer to the *chkpnt*(5) and *core*(5) pages in the *ConvexOS Programmer's Reference* for more information about this format.)

Since the format of checkpoint files is similar to that of ConvexOS core files (though the checkpoint file contains information in addition to that normally found in core files), ConvexOS debugging utilities treat checkpoint files as though they were core files. For example, `adb` (the ConvexOS assembly language debugger) can read and alter the portions of the checkpoint file that correspond to core files. In addition, `sod` (displays SOFF files in human readable form) also works for checkpoint files.

Checkpoint File Size

The exact amount of disk space that will be consumed by the checkpoint file for any given process cannot be calculated easily in advance. The entire address range in use by the checkpointed process—including text, data, and stack—will be written to the file. As a rule of thumb, the checkpoint file will be *at least* as large as the virtual address space of the process. To see the size of a process' virtual address space, enter `ps -l` as in Figure 1-4.

Figure 1-4: Using *ps -l* to Show Process Size

```
% loop > loopy &
[1]     28921
% ps -l
      F  uid   PID  PPID CP    PRI  NI   SZ  RSS    WCHAN  STAT   TT  TIME COMMAND
 800009 1916 28909 28908  5 1.3e+02  0   76   36    31e43c  S     p6  0:01 -ksh
      9 1916 28921 28909 62 1.5e+02  0   60   36            R     p6  0:03 loop
      9 1916 28922 28909 17 1.3e+02  0 5128  352            R     p6  0:00 ps -l
%
```

Size in kilobytes of the address space of each process is shown under "SZ" (indicated by shaded rectangle).

In this example, a process named "loop" is started in background mode. The `ps -l` command shows the size of each process in kilobytes under the "SZ" heading. (surrounded by a gray rectangle in Figure 1-4). For example, the amount of virtual address space used by the `loop` process is 60 kilobytes. (Refer to the *ps*(1) page of the *ConvexOS Programmer's Reference* for more information about this utility.)

To see the size of the checkpoint file you could use the `ls -l` or `ls -s` commands after checkpointing the process, as shown in Figure 1-5.

Figure 1-5: Using ls to Show Checkpoint File Size

```
% chkpnt 28921
% ls -l
total 266
drwxrwxr-x  2 cash          512 Jun  2 16:25 chkdir
-rw-rw-r--  1 cash       176166 Jun  2 16:27 loop.272
-rw-------  1 cash       127035 Jul 13 11:04 loop.28921
-rw-------  1 cash            0 Jul 13 11:04 loopy
drwxr-xr-x  2 cash          512 Jun 28 14:50 out
drwxrwxr-x  2 cash          512 Jun  2 15:13 scripts
% ls -s
total 266
   2 chkdir        126 loop.28921        2 out
 126 loop.272       30 loopy             2 scripts
%
```

In Figure 1-5, `ls -l` shows the checkpoint file (loop.28921) to contain 127,035 bytes; `ls -s` shows that it actually occupies 126 kilobytes (126 kilobytes x 1024 = 129,024 bytes) of physical disk space.[1]

If you elect to save files used by checkpointed processes (this is an option of the `chkpnt` utility), then these files will, of course, use additional disk space.

1. The two sizes may be different because the `chkpnt` utility will not write blocks of zeroes to disk, providing that the zeroes occupy an entire filesystem block. (Such blocks might be written if the process has large uninitialized arrays.) Instead of writing such blocks of zeroes, `chkpnt` will "seek" ahead an appropriate number of blocks (using the `lseek` system call). To find out how much disk space a checkpoint file really occupies, use the `du` or `ls -s` commands. The difference between the file size returned by the `du` or `ls -s` commands and the size obtained by `ls -l` or `wc -c` is the amount of physical disk space that has been saved by "seeking over" blocks of zeroes in the process data structure.

What Happens During Restart

Restarting is the reverse of checkpointing: beginning with the root process, the checkpoint files for all members of checkpointed process hierarchies are read and the processes "resurrected." The condition of all processes in the hierarchy is restored to what it was when it was stopped by the checkpointing operation. This means that the pid of the process, its threads (including thread ID), registers, file descriptors (including pipes and the data in them), virtual memory regions, text, and data are restored.

When a restart operation is begun, a `restart` process is created that does the work of restarting the target process or process hierarchy. When the restart operation is complete for the entire hierarchy, control is restored to the process that held it when the hierarchy was checkpointed. If any member of the hierarchy cannot be restarted, the restart of the whole hierarchy fails.

File Access During Restart

The `restart` process must have permission to access files that are needed by the target process. To ensure this, it may be necessary to run `restart` as root or with the same uid as the target.

The `restart` process needs file access because—when a process is restarted–`restart` opens the files that were held open by the target process when it was checkpointed, and then hands the file descriptors to the (restarted) target process. Because it has the file descriptors, the target can read and write the files as before.

If the restarted target process had file descriptors open to unlinked files or directories when it was checkpointed, these file descriptors will point to unlinked files in /tmp after restart.

Restarting on a Different System

Processes or process hierarchies can be checkpointed on one CONVEX system, then restarted on a different system. Resources needed by the process must be present on the new system. For example, if the process requires access to certain files, those files must have been copied to the new system. (Since CR identifies files by pathname, the complete pathname of the copied file must be the same as its pathname on the original machine.)

As mentioned in the "Compatibility" section on page 4, the hardware architecture and operating systems of the two machines must be compatible.

User ID and Group ID of Restarted Processes

Restoring the Target Process uid, gid, and Group Access Lists

If `restart` is running with the effective uid of root, the target process will be restored to the same effective and real uid and gid values as it had when it was checkpointed. The supplemental group access list of the process is also restored. However, the saved setuid is *not* restored—instead, it is set to the same value as the effective uid of the target.

If `restart` is not running as root, then the uid, gid and group access list will be "inherited" from the process that invoked `restart`.

Caution

Since checkpoint files may be altered by users who have write permissions to them, caution should be exercised in restarting processes while running as root. For example, a checkpoint file may be altered by an unauthorized user so that the restarted process runs as root if it is restarted by root, and thus compromises the security of the system.

File Permission and Communication Problems after Changed uid or gid

If the uid and gid values of the target are not successfully restored to what they were when the target was checkpointed, file access and communication problems may result. If the target process is restarted with different uid or gid values from the ones it had when it was checkpointed, then it may not be able to access files owned by the original user or group. Also, interprocess communication (e.g., signals) may fail if other processes in the restored hierarchy expect the target process to have the same uid as before. If such failures occur, restart the target process as root or as the owner of the target.

Problems may also arise if the target process changed its effective uid or gid (via a `setuid` or `setgid` system call) at some time before it was checkpointed. For example, the target process may have begun its life as root, opened some files that are accessible only to root, and then changed its uid to that of the user who started the process, (e.g., user "smith").

If such a process is checkpointed after it changed its uid or gid, then it may not be possible to open the files that the target needs unless `restart` is running as root. If restart is running with smith's uid, then permission to open files that can be accessed only by root will be denied.

Parent Process ID (ppid) of Target Process

As its name implies, the parent process ID (ppid) of a process is the pid of its parent. For all members of a restarted hierarchy except the root process, the ppid is the same as it was when they were checkpointed. Because the parent of the root process of a restarted hierarchy is the `restart` process, the ppid of the root does not remain what it was when the process was checkpointed; instead, the ppid of such processes is always the same as the pid the `restart` process. (Since a single restarted process is logically the same as the root of a hierarchy that has only one member, this rule applies to the ppid of such single processes also.)

Current Working Directory

The current working directory (cwd) of the restarted target is the same as it was when the target was checkpointed, providing that this directory still exists when the process is restarted. If the cwd of the target no longer exists, then the restart will usually fail. However, restart will not fail if the current working directory of the target was removed *before* it was checkpointed. Such processes are restarted with an unlinked cwd in /tmp; note that this may cause any `fstat` function call issued by the target to return unexpected values after restart.

Restarting Under Share

If you are restarting a process as root on a machine running the Share scheduler,[1] the restarted process will run using root's shares. If you want to run the process using another share account, use `slxqt`. (Refer to the *slxqt(1)* man page in the *ConvexOS Programmer's Reference* for information on how to use this utility.)

1. Share is an optional CONVEX product.

Job Control

Control Terminals

Most interactive processes have a *control terminal*—a terminal from which they expect input and to which they send output. (This is usually the login terminal of the user who started the process). When you restart such a process from a terminal, the target process will not attempt to use its old control terminal. Instead, it will "talk" to you via the terminal you are now using. You can send the process input from this terminal, and output will be displayed there.

To provide this functionality, target process file descriptors that refer to a control terminal are treated differently from other file descriptors. If the `restart` process has a control terminal (as it would if you invoke it from a terminal), the file descriptor of that terminal is used as the control terminal of the target process. Terminal settings and special characters are restored to what they were when the process was checkpointed. Exceptions to this are the baud rate and window size—the values for these two are left as they were for the control terminal before the target process was restarted. (If the current window size is different from the checkpointed window size, a SIGWINCH signal will be sent to the target process.)

Caution

When restarting processes that have a control terminal from an environment where there is no control terminal, you must use the -F (force) option of restart. For example, if a process that has a control terminal (e.g., was started interactively by a user) is checkpointed and then later restarted via CXbatch, it will fail unless -F is used.

This special handling does not apply if the target process sends output to any terminal other than its control terminal. File descriptors that point to terminals other than the control terminal are restored as they were when the target process was checkpointed.

Process Groups and Terminal Control

The file descriptor of the control terminal determines which terminal the target expects to be the source of commands and the destination of output. However, for a process to receive input from a terminal, it is not enough for it to be the control terminal of that process.

Each process belongs to a *process group*, and the kernel allocates control of terminal devices on the basis of this process group ID. The process group that currently has access to the terminal is called the *terminal group*, or the *foreground group*. When you are using a terminal interactively, the process group of the shell is usually the terminal group. This means that the shell automatically receives any commands that you enter at your terminal. To make another process the recipient of these commands, you must start it from the shell command line, and run it in the foreground. When you do this (e.g. by entering an `ls` command), the process group ID of that process (in this case, `ls`) becomes the controlling group. Thus, if you enter a break character (CONTROL-C), the `ls` process—and not the shell—receives the command, and exits. When the second process exits, the shell resumes control of the terminal.

The situation is a bit more complicated if you restart a process in the foreground. For example, suppose that you enter the following:

```
% restart procl
```

After you have entered this command, a `restart` process is started, and the process group ID of `restart`—and not the process group ID of your shell—is now the controlling group. When `restart` starts the target process or process hierarchy (`procl`), it causes the process group ID of `procl` to become the terminal group. Thus, any in-

put from your terminal will go directly to `proc1`. If you enter commands at your terminal, `proc1` will receive them.

When `proc1` dies, the process is reversed—the `restart` process makes its own group the terminal group, and then exits. (If it did not do this, the shell would generate spurious error messages.)

Passing Signals Through

When the `restart` process receives a signal that is likely to be intended for the target, and not `restart` itself, it passes that signal through to the target process. The following signals are "passed through" in this manner by `restart`: SIGINT, SIGQUIT, SIGSTP, SIGHUP, SIGTERM, SIGWINCH, SIGUSR1, and SIGUSR2.

The example in Figure 1-6 illustrates "passing through" signals:

Figure 1-6 Passing Signals Through

Restart process from file "example.1" →
put job in background →
a jobs command shows active jobs →
Kill job 1 (the restart process →
Another jobs command shows no jobs →

```
% restart example.1
^Z
% jobs
[1] + Stopped restart example.1
% kill -TERM%1
% jobs
%
```

In this example, the `restart` job was begun in the foreground, but then stopped with a **CONTROL-Z** command. When this command was given, the `restart` process passed it on the target, which then went "to sleep"; since the controlling process was stopped, the shell was given control of the terminal again. The `kill` command then caused a SIGTERM signal to be sent to the `restart` process, and this signal was passed on to the target process, thus killing it.

The situation would be the same if you started the `restart` job in the background (by ending the command with an ampersand). Any signal in the above list sent with a `kill` command would be passed through to the target process by `restart`.

The Checkpoint and Restart Utilities

2

Two ConvexOS command-line utilities provide checkpoint and restart services: chkpnt and restart. These utilities are described in this chapter. General information about checkpointing and restarting processes can be found in Chapter 1, "Checkpoint Restart Overview". The *chkpnt*(1) and *restart*(1) man pages in the *ConvexOS Programmer's Reference* also contain relevant information.

The chkpnt Command

To checkpoint processes, enter the chkpnt command at the ConvexOS shell prompt. This command has the following format:

chkpnt [-CFinqvX] [-r|-j|-p] [-d *checkpoint_directory*]
 [-f *checkpoint_file*] [-I *logfile*] [-L *logfile*]
 [-k *signo*|-K *signo*] {-P *fd* | *pid*}

chkpnt Parameter Summary

The meaning of the chkpnt parameters and option flags is summarized here, and described in greater detail in the "Using the chkpnt Command" section on page 2.

The following options can be specified singly or in combination with other parameters or options:

Option	Meaning
-C	Copy all open files to the checkpoint directory.
-F	Force checkpointing despite error conditions.
-i	Invoke interactive mode for chkpnt command.
-n	Perform checkpointability tests on target, but do not checkpoint.
-q	Quiet mode; suppress warning messages—error messages not suppressed.
-v	Verbose output.
-X	Print debugging output.
-r	Recursive checkpoint; checkpoint entire process hierarchy beginning at *pid*.
-j	Recursive checkpoint (same as -r).
-p	Do not perform recursive checkpoint (the default).

The following parameters can be specified alone or in combination with other parameters or options:

Parameter	Meaning
-d *checkpoint_directory*	Create checkpoint file in specified directory; default is current working directory.
-f *checkpoint_file*	Use this name for the checkpoint file (may be full pathname).
-I *log_file*	Use a log file (created with -L parameter) as command input for chkpnt.

`-k` *signo* — Send target process a signal (specified by *signo*) after checkpointing of the target process or process hierarchy has been completed.

`-K` *signo* — Like `-k`, but send signal (specified by *signo*) to every process in the hierarchy.

`-L` *logfile* — Invoke interactive mode of `chkpnt` and record actions in a file having the name *logfile*.

`-P` *fd* — File descriptor of target process (obtained by making a call to `pattach`; this is normally used only for debugging the target process.

pid — The process identifier (*pid*) of the target process; this may be either an individual process, or the root of a process hierarchy to be checkpointed). Either *fd* or *pid* is required.

Using the chkpnt Command

Though a number of other parameters can be given, you can checkpoint a process by simply specifying the process identifier (pid) with the `chkpnt` command. (The pid can be determined with the `ps` command. Refer to the *ps*(1) page in the *ConvexOS Programmer's Reference* for more information.) To checkpoint a process with pid 7365, this would be the simplest form of the command:

```
% chkpnt 7365
```

Checkpointing a process creates a "checkpoint" file. This file can later be used to restart the checkpointed process (also known as the *target* process) with the `restart` command. Though the target can be killed with the `chkpnt` command (using the `-K` *signo* parameter), the default operation of this command permits the process to continue executing once checkpointing has been completed. Refer to Chapter 1, "Checkpoint Restart Overview" (especially the sections called "Limitations" and "What Happens During Checkpointing") for general information about checkpointing processes.

Caution

Normally, chkpnt will not overwrite existing checkpoint files. If the same process must be checkpointed several times and you want to preserve the old checkpoint files, specify a new file name with the -f parameter, or use -d to put the file in a different directory. Be careful of the force (-F) option; if you use it, chkpnt will overwrite checkpoint files.

In addition to this simple form of the command, a number of parameters and option flags can be specified with `chkpnt`. The effect of these parameters and flags is described below.

Save Open Files

`-c` — By default, `chkpnt` does not save the data contained in regular files that were open to the target process—only the file descriptors are saved. When the process is restarted, these file descriptors are used to reopen the files. Consequently, the files cannot have been changed since the process was checkpointed. (The `restart` utility checks for this; if the file was changed after the checkpoint, then a warning will be given. The consequences of overriding the warning are unpredictable.)

The `-c` flag can be used with `chkpnt` to cause all regular files open to the target process to be copied to the checkpoint di-

rectory. (You can then copy the files to their original location before you restart the process.)

The following is an example of a chkpnt that uses the -C flag to store all open regular files in the current working directory:

% chkpnt -C 7365

Specify Checkpoint File Name and Checkpoint Directory

-d *checkpoint_directory* The name of the directory in which the checkpoint file is to be created; this directory must already exist. Either a relative or absolute pathname may be specified. (Relative pathnames are relative to the current working directory.) In the following example, an absolute pathname is given—the checkpoint file will be created in the /mnt/checkpoint directory:

```
% chkpnt -d /mnt/checkpoint 7365
```

-f *checkpoint_file* The name of the checkpoint file can be specified with this parameter. If no checkpoint file name is specified, the default name will be used. Refer to "Checkpoint File Name" on page 9 of Chapter 1 for information about the naming conventions. If a process hierarchy is being checkpointed, and if -f is specified, the given name will be used as the first element of the checkpoint file name for each process.

Since chkpnt will not overwrite existing checkpoint files, -f can be used to specify a new file name for previously checkpointed processes.

If the name specified with -f contains no slashes (i.e., is only a file name), and if -d is not given, the checkpoint file will be placed in the current working directory. The following example will create a checkpoint file called "simul1" in the current working directory:

```
% chkpnt -f simul1 7365
```

If a checkpoint directory is also specified with -d, the file will be placed in that directory. The following example places the checkpoint file in /mnt/chk/simul1:

```
% chkpnt -d /mnt/chk -f simul1 7365
```

It is possible to specify the directory in the -f parameter, thus making -d redundant. This example would cause the same result as the preceding one:

```
% chkpnt -f /mnt/chk/simul1 7365
```

Either an absolute or a relative pathname may be specified with -f. The following example places the checkpoint file in a subdirectory of the current working directory; the subdirectory is called "save":

```
% chkpnt -f save/simul1 7365
```

If the name given with -f contains slashes, the -d parameter is redundant. If -d is specified along with -f, the path spec-

ified by both must be the same, or chkpnt will fail. For example, the following is wrong, and will cause failure:

```
% chkpnt -d /mnt/chk -f chk/simul1 7365 &
chkpnt: checkpoint directory "/mnt/chk"
conflicts with checkpoint file path "chk/
simul1"
```

Force Checkpointing

`-F`

Force checkpointing despite error conditions. If this flag is specified, the utility will attempt to complete checkpointing even though error conditions occur that would otherwise cause checkpointing to be aborted. A checkpoint file created with this flag may not restart correctly. This flag should be used only when the error condition reported by chkpnt is not essential to correct execution of the target process (e.g., an unused pipe to a process outside of the restarted hierarchy).

Caution

Using -F overwrites any existing checkpoint files of the same name. If you previously checkpointed the same process and want to preserve the old checkpoint file, specify a new file name or directory.

Checkpoint in Interactive Mode

`-i`

Interactive mode; allows making decisions about each open file descriptor and each process of a hierarchy being checkpointed. In particular, this mode allows you to choose which open files to copy to the checkpoint directory (as opposed to the -C flag, which automatically copies all regular files open to the target process to the checkpoint directory).

Refer to the "Interactive Mode" section on page 8 for more information about -i.

`-L` *logfile*

Invoke the interactive mode of chkpnt (refer to the description of -i above), and record the session in *logfile.* The session can later be "played back" by using the -I parameter; all the checkpoint options chosen during the session using -L will be repeated when chkpnt is later invoked with -I. Either a relative or an absolute pathname may be given; if a relative pathname is specified, the file will be created in the checkpoint directory (refer to -d above). (Refer to the "Interactive Mode" section on page 8 for more information about using log files.)

The following example begins an interactive checkpoint session of process 3448, and records the session in a file called "mylog" in the checkpoint directory:

```
% chkpnt -L mylog 3448
```

`-I` *logfile*

Use the log file created by a previous interactive chkpnt session that used the -L flag (refer to the description of -L above). The selections that were made during the previous session will be used for the current one; no further interactive input will be accepted. In the following example, the log file called "mylog" is used to control the checkpoint of process 3448:

```
% chkpnt -I mylog 3448
```

Checkpoint Entire Process Hierarchy

`-j`
: Checkpoint the entire process (job) hierarchy beginning with the process specified by pid. This is the same as the `-r` option.

`-r`
: Recursive checkpoint; the entire process hierarchy rooted at the process specified by pid is checkpointed (same as `-j`). If neither `-r` nor `-j` are specified, only the process specified by pid will be checkpointed. The following example recursively checkpoints all processes descended from process 1232:

```
% chkpnt -r 1232
```

`-p`
: Checkpoint only the process specified by pid; since this is the default if neither `-j` or `-r` is given, it is not necessary to specify -p.

Send Signal to Target

`-k` *signo*
: Send the signal specified by *signo* to the target process (i.e., the process being checkpointed) when the checkpoint has been successfully completed. If the checkpoint fails, no signal is sent. Either the signal name (e.g., KILL or SIGKILL) or the signal number (e.g., 9) may be specified.

The signal is sent only to the root process, even if a process hierarchy is being checkpointed. (To send a signal to every member of the hierarchy, use `-K`.)

The following example sends a "HUP" signal to process 9778 after it has been checkpointed:

```
% chkpnt -k HUP 9778
```

`-K` *signo*
: This flag works just like `-k`, except that if a process hierarchy is being checkpointed, a signal is sent to every member of the hierarchy once the checkpoint has been completed successfully. (No signal is sent if the checkpointing of any process in the hierarchy fails.) The following example sends a SIGUSR1 signal to every member of the process hierarchy beginning with process 9778:

```
% chkpnt -r -K SIGUSR1 9778
```

Diagnostics

`-n`
: No checkpoint—perform all the checkpointability tests on the target (for diagnostic purposes or to write an interactive log file with `-L`), but do not actually write a checkpoint file. The following example tests the checkpointability of process 345, invokes the interactive mode of `chkpnt`, and writes a checkpoint file called "checkit":

```
% chkpnt -nL checkit 345
```

`-p` *fd*
: File descriptor (obtained from `pattach`); can be used instead of pid when debugging the target process.

`-q`
: Quiet mode (for use with `-F`); suppresses warning messages. Fatal error messages are still generated. The following example checkpoints process 7519 in quiet mode:

```
% chkpnt -F -q 7519
```

-v
: Verbose output; gives additional information during checkpointing, such as file descriptors of files open by the target process, and other diagnostics.

```
% chkpnt -v 1232
```

-x
: Print debugging output.

Checkpoint Examples

There are two ways to use the chkpnt utility: as a single command issued from the shell command prompt, or interactively. The following examples illustrate various parameters and options of chkpnt. The first section following this one—"Shell Command Line Mode"—gives an example of the non-interactive (shell command line) mode of chkpnt. "Interactive Mode" section on page 8 gives examples of interactive checkpointing.

Shell Command Line Mode

To use chkpnt from the shell command line, enter the command name (chkpnt), followed by any desired parameters, and then the pid of the process you want to checkpoint. No further input will be allowed by chkpnt, and the shell prompt will return after checkpointing is complete.

Figure 2-1: Checkpointing from the Shell Command Line

Show what is in the current working directory with an *ls -F* →
There are 3 subdirectories, no files →
A *ps* shows currently running processes →
The target process (*pid* 491)→
Checkpoint the target →
ls -F shows the new checkpoint file (ksh.491) →

```
% ls -F
chkdir/   out/      scripts/
% ps
 PID TT  STAT   TIME COMMAND
  282 p0  S      0:04 -ksh[cash]
  491 p0  S      0:00 -ksh[cash]
  497 p0  S      0:00 sleep 10
  400 p1  I      0:01 -ksh
  496 pa  R      0:02 ps
% chkpnt 491
% ls -F
chkdir/   ksh.491   out/      scripts/
%
```

Figure 2-1 shows a simple checkpointing example. An ls -F command shows that there are no files (only subdirectories) in the current working directory. The ps command shows the processes that are owned by the user; the one with pid 491 is the target process. After checkpointing has been completed, the checkpoint file ksh.491 has been created in the current working directory.

The checkpoint file was created in the current working directory because no checkpoint directory was given on the command line. The examples in Figure 2-2 show how you can specify a checkpoint directory and a file name.

Figure 2-2: Specifying Checkpoint Directory and File Name

```
Checkpoint, put file in chkdir directory →                   % chkpnt -d chkdir 491
List contents of chkdir →                                    % ls -F chkdir
The checkpoint file is there →                               ksh.491
Checkpoint, put file in chkdir and call it "test1" →         % chkpnt -f chkdir/test1 -k HUP 491
The new checkpoint file is there →                           % ls -F chkdir
                                                             ksh.491  test1
Check active processes →                                     % ps
Because of the -k flag, the target was sent a HUP              PID TT  STAT   TIME COMMAND
signal, and was killed →                                       491 p1  Z      0:00 <defunct>
                                                               282 p0  I      0:04 -ksh[cash]
                                                               400 p1  I      0:01 -ksh
                                                               522 pa  R      0:00 ps
```

The last chkpnt command in Figure 2-2 included a -k parameter that caused a HUP signal to the be sent to the target process when checkpointing was completed. This signal caused the process to exit.

Interactive Mode

This mode is invoked by using the -i flag with chkpnt. Normally, all checkpoint parameters must be entered on the command line with chkpnt. In interactive mode, you will be prompted to make decisions such as which processes should be checkpointed (in process hierarchies), which file descriptors should be saved in the checkpoint file, and which files should be copied to the checkpoint directory. (This is the only way you can copy only a portion of the regular files open by the target process; the -c flag automatically copies *all* open regular files.)

Invoking chkpnt in Interactive Mode

In Figure 2-3, a ps command is used to show the processes being run by the user. One of these processes—the process "loop"—will be checkpointed; its pid is 5120. The chkpnt command is given with the -i option, and the interactive chkpnt prompt appears.

Figure 2-3: Invoking *chkpnt* Interactive Mode

The *ps* command shows running processes and their *pid* →

The chkpnt command with *-i* invokes interactive mode →

This line prompts for action on the item being displayed (the item here is the process "loop") →

```
% ps
PID TT STAT TIME COMMAND
 5026 p4 S 0:01 -ksh[cash]
 5032 p4 S 0:02 emacs -nw log
 5120 p4 R 0:11 loop
 4619 p6 S 0:01 -ksh
 5068 p6 R 0:03 ps
% chkpnt -i 5120
Chkpnt interactive mode. Type '?' for help.
loop - pid = 5120 pgrp = 5120 ppid = 5026 [cip]:
```

The last line in the screen above is the prompt; it always ends in a colon. You can give chkpnt single character commands by entering them after the prompt. (A list of legal commands is shown in Figure 2-4, below.) Some (but not necessarily all) of the commands that are legal for each prompt are shown in square brackets just before the colon.

The prompt shows information about the process or file descriptor that can be acted upon by entering a command. This process or file descriptor is called the "current item." In this example, the current item is the process "loop"; its pid, process group identifier (pgrp), and parent process ID (ppid) are shown in the prompt. After you take action on the current item by responding to the prompt, the next item is shown. When you have taken action on the last item, the interactive session ends, and chkpnt writes the checkpoint files. (If you quit before this by responding "Q" to any prompt, no checkpoint files are written.) After chkpnt has executed, the shell prompt returns.

Interactive Mode Commands

To display a list of all legal commands when the current item is a process, enter a question mark after the prompt, as shown in Figure 2-4.

Figure 2-4: Interactive *chkpnt* Commands for Processes

To see a list of commands applicable to the current item, enter a question mark (?) →

After the list, the prompt is redisplayed →

```
loop - pid = 5120 pgrp = 5120 ppid = 5026 [cip]:?

(Q)uit - exit without performing any checkpointing
(C)heckpoint - exit interactive mode and proceed with checkpointing
(G)oto - make a specified pid the current process
(P)rint - display information about all processes
(?) - display help message
(c)heckpoint - make the current process eligible for checkpointing
(i)gnore - make the current process ineligible for checkpointing
(p)rint - print information about the current process

loop - pid = 5120 pgrp = 5120 ppid = 5026 [cip]:
```

As shown in Figure 2-4, the following commands are available in the interactive mode of chkpnt when the current item is a process:

Q To quit the interactive chkpnt session without performing any checkpointing, enter Q; this cancels everything that you have done during this interactive session. This command can be used in response to any prompt at any time during the session.

C To exit the interactive session and proceed with checkpointing, enter C. The selections that you have made up to this point will be acted upon.

G To go directly to a certain process in a hierarchy (instead of going through them all in order), enter G, followed by a space and the pid of the desired target process. The specified process becomes the current item. If you enter G without a pid, the next process in the hierarchy will become the current item.

P Show information about all processes in the hierarchy (information includes the identifiers in the prompt line and all file descriptors open to the process).

? Display this list.

c To checkpoint the process shown in the prompt, enter c (the interactive session will continue and the next prompt is displayed, unless you have responded to all prompts). The actual checkpoint processing will be done only when the interactive session has ended.

i If you enter i, the process shown in the prompt will be ignored, and not checkpointed; you may still elect to checkpoint other processes in the hierarchy. Selectively checkpointing and restarting members of a process hierarchy (i.e., restarting some and not others) has unpredictable consequences. Some restarted applications may not work properly if this is done.

p This is like P, except that information will be shown only for the process displayed in the prompt.

When the current item is a file descriptor, the command menu is different from the menu that is available when the current item is a process. By entering ? in response to a file descriptor prompt, you will see a list of legal commands for file descriptors:

Figure 2-5: Interactive chkpnt Commands for File Descriptors

To see a list of commands applicable to the current item, enter a question mark (?) →

```
fd 3 -rw-r--r-- root Apr 11 16:39:17 1990 6872/0 /etc/ttys [is]:?

(Q)uit - exit without performing any checkpointing
(C)heckpoint - exit interactive mode and proceed with checkpointing
(G)oto - make a specified pid the current process
(P)rint - display information about all processes
(?) - display this help message
(s)ave - store data about the file in the checkpoint file.
(i)gnore - no information about the file will be stored
```

After the list, the prompt is redisplayed →

```
fd 3 -rw-r--r-- root Apr 11 16:39:17 1990 6872/0 /etc/ttys [cis]:?
```

As shown in Figure 2-5, the following commands are available in the interactive mode of chkpnt for file descriptors:

Q Exit interactive mode; do not write any checkpoint files.

C Exit interactive mode without allowing further input; proceed with checkpointing the process or process hierarchy.

G To go directly to a certain process in the hierarchy (instead of going through them all in order), enter G, followed by a space and the pid of the desired target process.

P Show information about all processes in the hierarchy (information includes the identifiers in the prompt line and all file descriptors open to the process).

? Displays this list

s Store data about the file in the checkpoint file (only file descriptor information—and not data—is saved).

c Copy the entire file to the checkpoint directory. (This command is not available if the current item is not a regular file (e.g., if it is a device).

i Ignore - no information about the file will be stored.

Print Information about Processes

When a process is the current item, you can see a list of open file descriptors for that particular process by entering p as in Figure 2-6.

Figure 2-6: Print Information About a Process

Enter "p" at the prompt to see descriptors of all files open to the process →
First file descriptor (standard in) →
Second file descriptor (standard out) →
Third file descriptor (standard error) →
Fourth file descriptor →

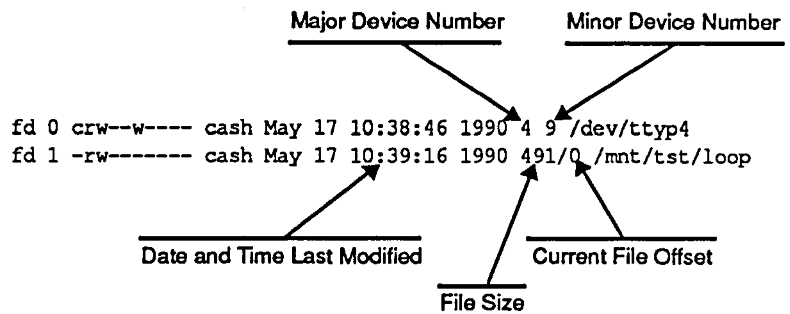
```
loop - pid = 5120 pgrp = 5120 ppid = 5026 [cip]:p
loop - pid = 5120 pgrp = 5120 ppid = 5026
fd 0 crw--w---- cash May 17 10:38:46 1990 4 9 /dev/ttyp4
fd 1 -rw------- cash May 17 10:39:16 1990 491/0 /mnt/tst/loop
fd 2 crw--w---- cash May 17 10:38:46 1990 4 9 /dev/ttyp4
fd 3 -rw-r--r-- root Apr 11 16:39:17 1990 6872/0 /etc/ttys

loop - pid = 5120 pgrp = 5120 ppid = 5026 [cip]:
```

If you enter an upper case P instead of a lower case p, and if a process hierarchy is being checkpointed, all processes and their file descriptors in the hierarchy will be shown.

In addition to the file descriptor number, access privileges, owner, and file name, the file descriptor entries also show such information as the major and minor device numbers (for device files), the date and time last modified, the file size (in blocks), and the current offset into the file. Figure 2-7 shows the information that is contained in each file descriptor entry.

Figure 2-7:File Descriptor Information



Checkpointing a Process Hierarchy in Interactive Mode

The following figures show an example of checkpointing a process hierarchy in interactive mode.

Figure 2-8: What is Running?

A *ps* -l command shows running processes→

This is the root process of the hierarchy that will be checkpointed.→

```
% ps -l
... UID    PID    PPID  ...    STAT  TT   TIME COMMAND
... 1916  15633  14498 ...    R     pl   0:03 ps
... 1916  15614  14498 ...    S N   pe   0:00 ksh
... 1916  15615  15614 ...    S N   pe   0:00 ksh
... 1916  15616  15615 ...    S N   pe   0:00 ksh
... 1916  15630  15614 ...    S N   pe   0:00 sleep 10
... 1916  15631  15615 ...    S N   pe   0:00 sleep 10
... 1916  15632  15616 ...    S N   pe   0:00 sleep 10
... 1916  14498  14497 ...    I     pf   0:01 -ksh[cash]
%
```

This `ps -l` command shows a number of processes owned by the user. (Not all the fields usually shown by this command appear in the example. Omissions are indicated by the ellipsis marks.) The checkpoint target is a process hierarchy rooted at the `kshv` that has a pid of 15614. To checkpoint this hierarchy interactively, you must use the `-r` (to checkpoint the whole hierarchy recursively) and `-i` flags. As shown in Figure 2-9, a `P` command will display information about the whole hierarchy.

Figure 2-9: List the Target Hierarchy

Invoke *chkpnt* with *-ri* →
Enter "P" to see information about all processes →
The root of the process hierarchy →
Standard in file descriptor →
Standard out file descriptor →
Standard error file descriptor →
File descriptor for process program file →
File descriptor for shell history file →
Next process in hierarchy →
Subsequent processes and file descriptors omitted for brevity

```
% chkpnt -ri 15614
Chkpnt interactive mode.  Type '?' for help.
ksh - pid = 15614 pgrp = 15614 ppid = 14205 [cip]:P

ksh - pid = 15614 pgrp = 15614 ppid = 14205
fd 0 crw--w---- cash May 23 14:03:04 1990 14 9 /dev/ttype
fd 1 crw--w---- cash May 23 14:03:04 1990 14 9 /dev/ttype
fd 2 crw--w---- cash May 23 14:03:04 1990 14 9 /dev/ttype
fd 30 -rwx---r-- cash May 15 15:52:19 1990 383/383 /mnt/test/ex
fd 31 -rw------- cash May 23 14:03:04 1990 10452/10452 /.khistory
sleep - pid = 15703 pgrp = 15614 ppid = 15614
.
.
.
ksh - pid = 15614 pgrp = 15614 ppid = 14205 [cip]:
```

As shown in Figure 2-9, the root process (15614) has five open file descriptors. (The other members of the process hierarchy are not shown.) In Figure 2-10, the process is checkpointed. Information about all file descriptors is saved in the checkpoint file; in addition, the /mnt/test/ex file is copied into the checkpoint directory.

Figure 2-10: Checkpoint the Process

Checkpoint the process by responding with "c" to the prompt →
Save file descriptor for fd0 by entering "s" →
Save fd1 →
Save fd2 →
Copy contents of file to checkpoint directory →
Save file descriptor →
Exit interactive *chkpnt*; finish checkpointing →

```
ksh - pid = 15614 pgrp = 15614 ppid = 14205 [cip]:c
fd 0 crw--w---- cash May 23 14:03:04 1990 14 9 /dev/ttype [is]:s
fd 1 crw--w---- cash May 23 14:03:04 1990 14 9 /dev/ttype [is]:s
fd 2 crw--w---- cash May 23 14:03:04 1990 14 9 /dev/ttype [is]:s
fd 30 -rwx---r- cash May 15 15:52:19 1990 383/383 /mnt/test/ex
[cis]:c
fd 31 -rw------ cash May 23 14:03:04 1990 10452/10452 /mnt/.khistory
[cis]:s
sleep - pid = 15703 pgrp = 15614 ppid = 15614 [cip]:C
%
```

After you enter C, chkpnt will complete the checkpointing operation noninteractively. When chkpnt is done, the shell prompt returns.

Creating and Using Checkpoint Log Files

If you want to checkpoint the same process multiple times, you need only checkpoint it interactively once if you use the -L parameter to create a log file. This file contains information about the decisions that you made during the interactive session; you can "play back" this session by using this log file as input by specifying the -I parameter with a command-line mode chkpnt command, as shown in Figure 2-11.

Figure 2-11: Creating a Checkpoint Log File

Checkpoint the process interactively and create log file called "mylog" →

Examine contents of log file →

```
% chkpnt -L mylog 650
Chkpnt interactive mode.  Type '?' for help.
ksh - pid = 650 pgrp = 650 ppid = 644 [cip]:c
fd 0 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 [is]:s
fd 1 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 [is]:s
fd 2 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 [is]:s
fd 3 -rw-r--r-- root Apr 11 16:39:17 1990 6872/0 /etc/ttys [cis]:s
fd 30 -rwx------ cash May 27 15:17:19 1990 383/383 /mnt/ex3 [cis]:c
% cat mylog
650 ksh checkpoint
0 :/dev/ttyp0: path
1 :/dev/ttyp0: path
2 :/dev/ttyp0: path
3 :/etc/ttys: path
30 :/mnt/cash/bin/victim: copy
31 :/mnt/cash/.khistory: path
%
```

The log file mylog contains information that can be used to cause subsequent invocations of chkpnt to take the same actions as the first time. Figure 2-12 shows how subsequent chkpnt commands can use the log file.

Figure 2-12: Using a Checkpoint Log File

Use the log file to control *chkpnt* →
chkpnt shows the actions taken →

```
% chkpnt -I mylog 650
ksh - pid = 650 pgrp = 650 ppid = 644 checkpoint
fd 0 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 pathname
fd 1 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 pathname
fd 2 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 pathname
fd 3 -rw-r--r-- root Apr 11 16:39:17 1990 6872/0 /etc/ttys pathname
fd 30 -rwx------ cash May 27 15:17:19 1990 383/383 /mnt/ex3 copy
%
```

Caution

Do not use log files for processes using files that have names containing colons. The checkpoint log file format uses colons as delimiters; file names containing colons will cause errors.

The restart Command

To restart processes from a checkpoint file, enter the `restart` command at the ConvexOS shell prompt. This command has the following format:

`restart [-CFiqtvwWXz] [-k` *signo* `|-K` *signo*`]` *checkpoint_file*

restart Parameter Summary

The meaning of the `restart` parameters and option flags is summarized here, and described in greater detail in "Using the restart Command", below.

The following options can be specified singly or in combination with other parameters or flags:

Option	Meaning
`-C`	Copy files back into same directories as at checkpoint.
`-F`	Force restart despite error conditions.
`-i`	Invoke interactive mode of `restart`.
`-q`	Quiet mode—suppress warning messages (for use with `-F`).
`-t`	Restart in traced state (for debuggers).
`-v`	Verbose output.
`-W`	Do not wait for restarted process to complete.
`-w`	Wait for restarted process to complete (this is the default behavior; the flag is supplied for compatibility with other implementations).
`-X`	Print debugging output.
`-z`	Restart the process in a stopped state.

The following parameters can be specified by themselves or in combination with other parameters or options:

Parameter	Meaning
checkpoint_file	Restart the process from the specified checkpoint file (required).
`-k` *signo*	Send signal specified by *signo* to target process when restart is complete.
`-K` *signo*	Like `-k`, but signal is sent the every member of the process hierarchy rooted at the target process.

Using the restart Command

Although a number of other parameters can be given, you can restart a process by simply specifying the checkpoint file name with the `restart` command. To restart a process that was checkpointed in a file called "/mnt/checkpoint/ex3", this would be the simplest form of the command:

```
% restart /mnt/checkpoint/ex3
```

The process about which information was stored in this file will now be "resurrected" in the state that it was in when it was checkpointed. If this process was the root of a process hierarchy, and if the hierarchy was checkpointed recursively (with the `-r` flag of `chkpnt`), the entire process hierarchy will be restarted. Unless the `restart` command was issued in background mode (by ending the command with a &), the shell from which the command was issued will be suspended until the restarted processes have exited.

The process can be killed and restarted from the same checkpoint file as often as desired.

Refer to Chapter 1, "Checkpoint Restart Overview" (especially the sections called "Limitations" and "What Happens During Restart") for general information about restarting processes.

In addition to this simple form of the command, a number of parameters and option flags can be specified with `restart`. The effect of these parameters and flags is described below.

Send Signals to Target

`-k` *signo*

Send the signal specified by *signo* to the target process (i.e., the process being restarted) when the restart is complete. By default, a SIGCONT signal is sent to the target process when restart is complete; `-k` can be used to override this default. The signal can be specified by either signal name (e.g., CONT) or by number. If the specified signal is zero (`-k` 0), no signal will be sent.

If a process hierarchy is being restarted, the entire hierarchy must be restarted before the signal is sent. Furthermore, the signal is sent only to the root process; to send a signal to every process in the hierarchy, use `-K`.

The following example sends a SIGUSR1 signal to the target process:

```
% restart -k SIGUSR1 /mnt/checkpoint/ex3
```

`-K` *signo*

Like `-k`, but send the signal specified by *signo* to every member of the process hierarchy being restarted.

Restart in Interactive Mode

`-i`

Interactive `restart` mode—like its counterpart in `chkpnt`—permits you to make item-by-item decisions about processes and file descriptors when you are restarting processes. For example, you can choose to restart only some processes in a hierarchy, or you can cause certain file descriptors to be ignored or changed. See "Interactive Mode" section on page 18 for more information about how to use this mode.

The following example restarts the process stored in /mnt/checkpoint/ex3 interactively:

```
% restart -i /mnt/checkpoint/ex3
```

Copy Back Files

`-C`

If the files open to the target process were copied to the checkpoint directory (by specifying `-C` with the `chkpnt` command), then they can be copied back from the checkpoint directory to their original location by specifying `-C` with `restart`.

Caution

Using -C will copy the files that were saved to the checkpoint directory back to their original locations; the files in those locations will be overwritten, and any changes made after checkpointing will be lost.

Force Restart

`-F`
Force restarting despite error conditions. If this flag is specified, then the utility will attempt to restart the target process even though error conditions occur that would ordinarily cause the restart to be aborted. Processes restarted in this manner may not execute properly.

The following example forces restart of the process stored in /mnt/checkpoint/ex3:

```
% restart -F /mnt/checkpoint/ex3
```

Wait/Don't Wait

`-w`
Wait for target process; because this is the default, the `-w` flag need not be specified (it is provided for compatibility with other interfaces). If the default is in effect, the `restart` process forks the target process (and any other processes in the hierarchy), and waits for it to exit. (Unless `restart` is run in the background, the shell is suspended until `restart` exits.)

`-W`
Do not wait for target process. If this flag is specified, the restart process does not fork the target—instead, it *becomes* the target process (or the root process of the hierarchy being restarted). If this happens, the pid of `restart` is changed to that of the target process. This flag is for noninteractive invocation of `restart` only, and should never be specified from an interactive shell. Because `restart` changes its pid when `-W` is used, the shell will become "confused," and remain suspended even after the target process exits.

Diagnostics

`-q`
Quiet mode; suppress warning messages (for use with `-F`).

`-t`
Restart in traced state (similar to the `exect` system call); used for restarting under control of a debugger.

`-v`
Verbose output; gives additional information (such as file descriptors used) during restart.

The following example restarts the process stored in /mnt/checkpoint/ex3 and gives verbose output:

```
% restart -v /mnt/checkpoint/ex3
```

`-z`
Restart the process in a stopped state.

The following example restarts the process stored in /mnt/checkpoint/ex3 in a stopped state:

```
% restart -z /mnt/checkpoint/ex3
```

The effect of `-z` is the same as using `-K SIGSTOP`; `-z` cannot be used with `-k` or `-K`.

Restart Examples

There are two ways to use restart: as a single command issued from the shell prompt, or interactively. The following examples illustrate the use of various parameters and options of restart. The first section following this one—"Shell Command Line Mode"—gives an example of the noninteractive (shell command line) mode of restart. "Interactive Mode" (below) shows examples of interactive mode.

Shell Command Line Mode

To use restart from the shell command line, enter the command name (restart) followed by the name of the checkpoint file that you wish to restart, and any other parameters that you want to specify. No further input will be allowed by restart. Unless you issue the restart command in background mode (by terminating it with an &), the shell prompt will not return until the restarted process has exited.

Figure 2-13: Restarting from the Shell Command Line

A *ps* command shows the running processes→
An *ls -F* shows files in working directory→
loop2.14501 is a checkpoint file→
Restart *loop2* from checkpoint file (in background)→
Show running processes →
The *restart* process is waiting for *loop2* to exit→
loop2 is now running with its original *pid*→

```
% ps
  PID TT  STAT    TIME COMMAND
 3812 p9  I       0:00 -ksh
 3948 pc  R       0:00 ps
% ls -F
chkdir/       loopout      out/         typescript
loop2.14501   mylog        scripts/     victim*
%
% restart loop2.14501 &
% ps
  PID TT  STAT    TIME COMMAND
 3812 p9  I       0:00 -ksh
 3953 pb  S       0:00 restart loop2.14501
14501 pb  S       0:00 loop2
 3957 pc  R       0:00 ps
%
```

Figure 2-13 shows a simple restart example. The ps command shows the running processes to be a ksh and the ps command itself. An ls -F shows a checkpoint file called "loop2.14501" among the contents of the current working directory. (The name of the checkpointed process was loop2; its pid was 14501.) After the process is restarted, a ps shows loop2 running under its original pid. The restart process itself is waiting for loop2 to exit. (If the restart command had not been entered in background mode—by ending it with an amperasnd (&)—then the shell prompt would not have returned until loop2 exited.)

Interactive Mode

This mode is invoked by using the -i flag with restart. Normally, all parameters must be entered on the command line with restart. But in interactive mode, you will be prompted to make decisions such as which processes in a process hierarchy to restart, and which file descriptors to use, change, or ignore.

Invoking restart in Interactive Mode

Figure 2-14 shows an example of invoking restart in interactive mode. In that figure, a ps command shows the currently running processes, and an ls -F shows the

contents of the current working directory. The checkpoint file that is used for the restart is loop2.14501.

Figure 2-14: Invoking restart Interactive Mode

A *ps* command shows the running processes→
An *ls -F* shows files in working directory→
loop2.14501 is a checkpoint file→
Restart *loop2* in interactive mode)→
This is the interactive mode prompt→

```
% ps
  PID TT  STAT    TIME COMMAND
  445 p0  S       0:00 -ksh
  455 p8  R       0:02 ps
% ls -F
chkdir/       loopout       out/          typescript
loop2.14501  mylog          scripts/      victim*
% restart -i loop2.14501
Restart interactive mode.  Type '?' for help.
loop2 - pid = 14501 pgrp = 14501 ppid = 14498 [ipr]:
```

As in interactive chkpnt mode, the last line in Figure 2-14 is a prompt, and shows the "current item" (i.e., the item about which you are being asked to make a decision). You can give single-character commands to `restart` by entering them after the colon in the prompt. (A list of legal commands is shown in Figure 2-15, below.) Some (but not all) commands that are legal for each prompt are shown in square brackets just before the colon.

Interactive restart Commands

To display a list of all legal commands for the current item, enter a question mark after the prompt. Figure 2-15 shows the legal commands for processes.

Figure 2-15: Interactive restart Commands for Processes

Enter "?" to see a list of commands→
After the list, the prompt is redisplayed→

```
loop2 - pid = 14501 pgrp = 14501 ppid = 14498 [ipr]:?

(Q)uit - exit without performing the restart
(R)estart - exit interactive mode and proceed with restart
(G)oto - make a specified pid the current process
(P)rint - display information about all processes
(?) - display this help message
(r)estart - make the current process eligible for restart
(i)gnore - make the current process ineligible for restart
(p)rint - print information about the current process and its chil-
dren

loop2 - pid = 14501 pgrp = 14501 ppid = 14498 [ipr]:
```

As shown in Figure 2-15, the following commands are available in the interactive mode of `restart` when the current item is a process:

Q To quit the interactive `restart` session without restarting anything, enter Q; this cancels everything that you have done during this interactive session. This command can be issued at any time in response to any prompt.

R To exit the interactive session and proceed with restarting, enter R. The selections that you have made up to this point will be acted upon.

G To go directly to a certain process in a hierarchy (instead of going through them all in order), enter G, followed by a space and the pid of the desired target process. The specified process will become the current item.

P Show information about all processes in the hierarchy (including file descriptors that were open to the process).

? Displays this list

r To restart the process shown in the prompt, enter r (the interactive session will continue and the next prompt is displayed, unless you have responded to all prompts). Restarting will be done only when the interactive session has ended.

i If you enter i, the process shown in the prompt will not be restarted; you may still elect to restart other processes in the hierarchy. Selectively restarting members of a process hierarchy (i.e., restarting some and not others) has unpredictable consequences. Some restarted applications may not work properly if this is done.

p This is like P, except that information will only be shown for the process displayed in the prompt.

When the current item is a file descriptor (and not a process), there are additional commands, some of the commands have a different meaning, and some are not available. By entering ? in response to a file descriptor prompt, you will see a list of legal commands for file descriptors, as in Figure 2-16.

Figure 2-16: Interactive *restart* Commands for File Descriptors

Restart the process by entering "r" →
"?" shows legal commands for file descriptor →
After the list, the prompt is redisplayed→

```
loop2 - pid = 14501 pgrp = 14501 ppid = 14498 [ipr]:r
fd 0 crw--w---- cash May 29 17:01:13 1990 4 9 /dev/ttyp4 [ium]:?

(Q)uit - exit without performing restart
(R)estart - exit interactive mode and proceed with restart
(G)oto - make a specified pid the current process
(P)rint - display information about all processes
(?) - display this help message
(i)gnore - ignore the file descriptor
(u)se - use the file descriptor
(m)odify - change the pathname of a file descriptor

fd 0 crw--w---- cash May 29 17:01:13 1990 4 9 /dev/ttyp4 [ium]:
```

As shown in Figure 2-16, the following commands are available for file descriptors in interactive `restart`:

Q Exit interactive mode; do not restart any processes.

R Exit interactive mode without allowing further input; proceed with restarting the process or process hierarchy.

G To go directly to a certain process in the hierarchy (instead of going through them all in order), enter G, followed by a space and the pid of the desired target process.

P Show information about all processes in the hierarchy (information includes the identifiers in the prompt line and all file descriptors open to the process).

? Display this list.

i Ignore (do not use) this file descriptor when restarting the process. (If the restarted process later tries to use the ignored file descriptor, file access will fail, and an EBADF error message will be returned to the process.)

u Use this file descriptor as it is.

m Modify this file descriptor to point to a different file; to modify, enter m followed by the pathname of the file to use. For example, if a file descriptor points to a file called /mnt/user/data1, you can use the file called /mnt/user/old.data1 by entering **m /mnt/user/old.data1**. (Note that changing the path referenced by a descriptor will affect any shared file descriptors created with a dup system call or shared across an exec system call.)

Print Information About a Process

When a process is the current item, you can see a list of open file descriptors for that particular process by entering lower case p. If you enter upper case P, and if a process hierarchy is being restarted, all the processes and their file descriptors in the hierarchy will be shown. This works like the corresponding features in interactive chkpnt (refer to "Print Information about Processes" section on page 11).

Restarting a Process Hierarchy in Interactive Mode

Interactive mode lets you select which processes to restart, and which file descriptors to use or change. Figure 2-17 shows an example of interactively restarting only the root of a hierarchy containing three processes.

Figure 2-17: Interactive *restart*

Show contents of current working directory →
loops.6934 contains root of process hierarchy →
Restart root process (*pid* is 6934) →
Ignore (do not restart) child process →
Ignore (do not restart) child process →
After restarted processes exit, prompt returns →

```
% ls -F
chkdir/          loops.loops.6935  out/           ttys*
loops.6934       loops.loops.6936  scripts/
% restart -i loops.6934
Restart interactive mode.  Type '?' for help.
loops - pid = 6934 pgrp = 6934 ppid = 6817 [ipr]:r
fd 0 crw--w---- cash Jun  2 14:16:27 1990 0 9 /dev/ttyp0 [ium]:u
fd 1 crw--w---- cash Jun  2 14:16:27 1990 0 9 /dev/ttyp0 [ium]:u
fd 2 crw--w---- cash Jun  2 14:16:27 1990 0 9 /dev/ttyp0 [ium]:u
fd 3 -rw-r--r-- root May 30 19:04:38 1990 6872/0 /etc/ttys [ium]:u
loops - pid = 6936 pgrp = 6934 ppid = 6934 [ipr]:i
loops - pid = 6935 pgrp = 6934 ppid = 6934 [ipr]:i
%
```

In this example, the first (root) process is restarted by responding with r to the prompt. The file descriptors for that process are then displayed; because the response to each prompt was u, they are all used. Because the response was i to the prompt for

each of the two child processes, these processes are not restarted. After last prompt shown in the example, the process is restarted. When it exits, the shell prompt returns.

Note that selectively checkpointing and restarting members of a process hierarchy (i.e., restarting some and not others) has unpredictable consequences. Some restarted applications may not work properly if this is done.

Checkpoint Restart Programming Interface

3

This chapter describes the Checkpoint Restart C and Fortran library functions that can be used to build CR capability into applications, discusses related programming considerations, and gives some code samples.

For additional information about the library functions, refer to the appropriate page in the *ConvexOS Programmer's Reference*. General information about checkpointing and restarting processes can be found in Chapter 1, "Checkpoint Restart Overview".

Checkpoint C Function

The C library function used to checkpoint processes is `chkpnt()`. The usage and parameters of this function are described below. The function is also described in the *chkpnt*(3) man page in the *ConvexOS Programmer's Reference.*

The `chkpnt()` function invokes the ConvexOS `chkpnt` utility at run time; the utility is then used to do the actual checkpointing. (Refer to Chapter 2, "The Checkpoint and Restart Utilities" for more information.) The utility need not be in the default search path of the `chkpnt()` process when it is invoked—the utility will automatically be sought in its default directory.

Caution

The chkpnt utility must reside in the default directory in which it was placed by the installation script (/usr/convex/chkpnt), or the chkpnt() function will not work. If chroot() is used to change the root directory, /usr/convex/chkpnt must exist at the new root.

The chkpnt() function cannot be called from within a multithreaded region of a program.

chkpnt() Format and Parameters

The `chkpnt()` function has the following format:

```
chkpnt (class, pid, dir, name, options, signo)
```

The following information and option can be specified in the parameters:

- int *class*
  If CHKPNT_FAMILY is specified for this parameter, the entire process hierarchy (beginning with the process having the identifier specified in the *pid* parameter) is checkpointed. If CHKPNT_PROC or 0 (zero) is specified, only the process indicated by *pid* will be checkpointed.
- int *pid*
  process identifier—If a single process is being checkpointed, this is the unique identifier (pid) of the target process. If a process hierarchy is being checkpointed, the *pid* is that of the parent process (also called the *root* process) from which the en-

tire hierarchy is descended. If the pid is zero, the process making the `chkpnt` call is itself checkpointed.

- char **name*
  checkpoint file name—A relative or absolute pathname must be specified; it will be the pathname of the checkpoint file of the target process. If a process hierarchy is being checkpointed, this name will be the checkpoint file of the root process. The maximum length of *name* is 16 characters, in addition to the null terminator. (Refer to the "The Checkpoint File" section on page 9 of Chapter 1 for more information about checkpoint file naming conventions.)

- int *options*
  options—you may specify one or more of the options listed below. The CHKPNT_SIGFAMILY and CHKPNT_SIGROOT options are mutually exclusive. All other combinations of options may be specified if they are separated by an *or bar* (|).

  CHKPNT_FORCE
  proceed despite error conditions—You may force checkpointing to proceed even if error conditions are encountered that would ordinarily cause checkpointing to be aborted. A checkpoint file created with this option may not restart correctly.

  For example, checkpointing a process that has an open socket will ordinarily fail. However, the process can be checkpointed using the CHKPNT_FORCE option. When the process is restarted from this file, it may run correctly if it does not try to access the socket.

  CHKPNT_KILL
  Kill the checkpointed processes when checkpointing has been successfully completed. If checkpointing fails for any reason for any process in the hierarchy, the target processes continue normal execution.

  CHKPNT_PFD
  interpret *pid* parameter as file descriptor—Ordinarily, `chkpnt()` calls `pattach()` to stop the target process and to return a file descriptor. (Refer to the *pattach(2)* page in the *ConvexOS Programmer's Reference* for more information about this function.) If the process that is calling `chkpnt()` has itself already invoked `pattach()` using the exclusive option (e.g. to debug the target), `chkpnt()` will use the pid as though it were a file descriptor returned by `pattach()`, and will not again call `pattach()`.

  CHKPNT_SIGFAMILY
  send signal to process family—Send all processes in the checkpointed hierarchy a signal when checkpointing of the hierarchy has been completed (no signal will be sent if checkpointing of any process fails). The signal must be specified in the *signo* parameter.

  CHKPNT_SIGROOT
  send signal to target process—Send a signal only to the root process of the hierarchy being checkpointed when checkpointing of the entire hierarchy has been completed (no signal will be sent if checkpointing fails). The signal must be specified in the *signo* parameter.

- int *signo*
  signal to send—The signal specified in this parameter is sent if the CHKPNT_SIGFAMLY or CHKPNT_SIGROOT options are specified in the *options* parameter. If either of these options is given, then *signo* must be specified. (Refer to the *signal*(3C) or *sigvec(2)* pages in the *ConvexOS Programmer's Reference* for a list of legal signals.)

chkpnt() Return Values and Error Codes

If checkpointing of the specified process or process hierarchy is successful, `chkpnt()` returns 0 (zero); if checkpointing fails for any target process, -1 is returned to indicate failure.

In case of failure, the external variable `errno` is set to an error code that indicates the reason for the failure. The following error codes may be returned (they are defined in /usr/include/sys/errno.h)[1]:

Code	Meaning
EACCESS	Search permission denied on a component of the path specified by *name* parameter (change the permissions or change the name).
EEXIST	The checkpoint file already exists (delete the file or specify another name with the *name* parameter).
EFBIG	The checkpoint file would have been too large if it had been written (e.g., because there is not enough room in the file system).
EINTR	The checkpointing operation was interrupted by a signal.
EINVAL	An invalid parameter was specified with the `chkpnt()` function call.
ENAMETOOLONG	The string specified for *name* is too long (i.e., longer than PATH_MAX).
ENOSPC	No free space on the device that contains *name*.
EPERM	The process making the `chkpnt()` call does not have permission to checkpoint one or more of the target processes.
EPIPE	One or more of the target processes has a pipe that ends at a process that is not in the target hierarchy.
EROFS	The file specified by *name* resides on a read-only file system.
ESRCH	No process with the specified pid has been found.

1. Typically, the perror() library function is used to handle error conditions. Refer to the perror(3) man page in the *ConvexOS Programmer's Reference* for more information about this function.

Restart C Function

The `restart()` C library function can be used to restart processes from checkpoint files. This function is also described on the *restart*(3) page in the *ConvexOS Programmer's Reference.*

The `restart()` function invokes the `restart` utility at run time; the utility is then used to do the actual restarting. (Refer to Chapter 2, "The Checkpoint and Restart Utilities" for more information.) The utility need not be in the default search path of the `restart` process when it is invoked—the utility will automatically be sought in the directory in which it was placed by the installation script.

Caution

The restart utility must reside in the default directory in which it was placed by the installation script (/usr/convex/restart), or the restart() function will not work.

The restart() function cannot be called from within a multithreaded region of a program.

If the target process was checkpointed as a member of a process hierarchy, all processes below the target process in that hierarchy will be restarted along with the target process.

After all processes in the hierarchy have been restarted, the root process will, by default, be sent a SIGCONT signal. This default can be modified with the *flags* parameter (described below).

restart() Format and Parameters

The `restart()` function has the following format:

```
restart (path, flags, signo)
```

The following information and options can be specified in the parameters:

- char **path*
  pathname of checkpoint file—the pathname of the checkpoint file from which the target process will be restarted. If a process hierarchy is being restarted, this is the checkpoint file of the root process. (Each process in the hierarchy will have its own checkpoint file.)
- int *flags*
  flags—you may specify one or more of the options listed below. The RESTART_SIGFAMILY and RESTART_SIGROOT options are mutually exclusive. All other combinations of options may be specified if they are separated by an *or bar* (|).

  RESTART_SIGFAMILY
  send signal to process family—Send all processes in the restarted hierarchy a signal when the hierarchy has been completely restarted (no signal will be sent if restart of any process fails). The signal must be specified in the *signo* parameter.

  RESTART_SIGROOT
  send signal to target process—Send a signal only to the root process of the hierarchy being restarted when the entire hierarchy is completely restarted (no signal will be sent if restarting fails). The signal must be specified in the *signo* parameter.

If neither RESTART_SIGFAMILY nor RESTART_SIGROOT are specified, the root process will be sent a signal; if, in addition, no signal is specified in the *signo* parameter, a SIGCONT signal will be sent to that process.

RESTART_FORCE
proceed despite error conditions—This option will cause `restart()` to attempt to restart a process even though it detects error conditions. Processes restarted with this option may not run.

For example, if another process on the system already has the pid of the process that is being restarted, the restart will ordinarily fail. If RESTART_FORCE is specified, the process will be restarted, but its pid will be different from what it was when the process was checkpointed.

Caution

Restarting a process with a pid other than the one it had when it was checkpointed has unpredictable results. The process may not execute correctly.

Another example would be a process that had a file open when it was checkpointed. If the file is not available when the process is restarted, restart will fail unless RESTART_FORCE is used. If this option is used, the process may run correctly—if it does not try to access the unavailable file.

RESTART_DEBUG
restart in debug state—The process will be started in debug state as though `exect()` had been called (refer to the *exect*(3) man page in the *ConvexOS Programmer's Reference* for more information). A process restarted with this option can easily be placed under control of a debugger.

RESTART_SUSPEND
restart in suspended state—the process will be restarted, but left in a "stopped" state as though it had been sent a SIGSTOP signal.

- int *signo*
  signal to send—the signal specified in this parameter is sent if the RESTART_SIGFAMILY or RESTART_SIGROOT options are specified in the *options* parameter. If either of these options is specified, *signo* must also be specified. (Refer to the *sigvec(2)* man page in the *ConvexOS Programmer's Reference* for a list of legal signals.)

If restart of the specified process or process hierarchy is successful, `restart()` returns the pid of the restarted process (or the root process if a hierarchy is being restarted). If restarting fails for any process in the hierarchy, -1 is returned. In case of failure, the external variable `errno` is set to one of the following error codes:[1]

Code	Meaning
EACCESS	Search permission denied on a component of the path specified by *name* parameter (change the permissions or change the name).
EAGAIN	The pid or process group ID (gid) of one or more of the target processes is already in use on the system.
EINTR	The restart operation was interrupted by a signal.
EINVAL	An invalid parameter was specified with the `restart()` function call.
ENAMETOOLONG	The string specified for *name* is too long (i.e., longer than PATH_MAX).
ENOENT	The checkpoint file *name* does not exist.

1. The perror() library function is typically used to handle error conditions. Refer to the perror(3) page in the *ConvexOS Programmer's Reference* for more information about this function.

Fortran CR Functions

Two Fortran library functions that perform checkpointing and restarting are provided with CR. These functions, chkpnt and restart, work like their C library analogues, and are described below.

No "header" file analogous to /usr/include/chkpnt.h is provided for the CR Fortran library calls. You must, therefore, make your own provision for defining constants for your Fortran program that are defined for C programs in the chkpnt.h file. (Refer to this file for a complete list of constants.) For example, if your program uses CHKPNT_KILL as a parameter, you must first declare it and define it with a Fortran parameter statement:

```
.
.
.
integer CHKPNT_KILL
parameter (CHKPNT_KILL = '0010'x)
.
.
.
```

Refer to the "Fortran Programming Example" section on page 16 for more information.

Fortran Checkpoint Function

The Fortran chkpnt function works like the corresponding C library function. Like its C counterpart, the Fortran function invokes /usr/convex/chkpnt at run time.

Caution

The chkpnt utility must reside in the default directory in which it was placed by the installation script (/usr/convex/chkpnt), or the Fortran chkpnt function will not work.

Format and Parameters

The Fortran chkpnt function has the following format:

```
integer function chkpnt (class, pid, name, options, signo)
```

This function has the same parameters as its C counterpart. They are summarized below; refer to the "chkpnt() Format and Parameters" section on page 1 for a complete description.

- integer *class*
- integer *pid*
- character*(*) name
- integer *options*
- integer *signo*

Fortran Restart Function

The restart Fortran function works like the corresponding C library function. Like its C counterpart, the Fortran function invokes /usr/convex/chkpnt at run time.

Caution

The restart utility must reside in the default directory in which it was placed by the installation script (/usr/convex/restart), or the Fortran restart function will not work.

Format and Parameters

The Fortran `restart` function has the following paramters:

```
integer function restart (path, flags, signo)
```

This function has the same parameters as its C counterpart. They are summarized below; refer to the "restart() Format and Parameters" section on page 4 for a complete description.

- character*(*) path
- integer flags
- integer signo

Programming Guidelines

If you are writing an application that may be checkpointed, you should adhere to the following guidelines:

- Do not use any resources not supported by CR. For example, do not use sockets, file or memory locks, unsupported devices, or MAP_DEVICE mapped memory segments.
- Do not store the current terminal name (as returned by the `ttyname` function) and rely on it to correspond to the current terminal over the entire lifetime of the application. The terminal name may be different if the application is restarted from a different terminal.
- Do not save the current time (obtained by the `time` or `gettimeofday` functions), and expect elapsed time to be accurate or meaningful later. The restart may occur long after the `time` or `gettimeofday` calls.
- Do not use the `setpid` system call. This call will fail if it is issued after the application is restarted, since `restart` calls `setpid` to restore the target's pid, and a process may call `setpid` only once.
- Do not fork children related to the application and then exit without waiting for the children. This makes it difficult for `chkpnt` to locate all the processes of the hierarchy, since no top-level parent will exist.

Device Interface

This section describes the interface between Checkpoint Restart and ConvexOS devices.

Not all ConvexOS devices support CR. If you attempt to checkpoint a device that does not support CR, the checkpoint will fail, and an error message like the following will be given:

```
/dev/somedevice: device does not support checkpoint
  capability
```

CR Device Driver ioctls

If your application must use a device that does not support CR, you can a customize a device driver to handle two new ioctls—IOCCHKPNT and IOCRESTART. Processes using such customized devices can be checkpointed.

IOCCHKPNT and IOCRESTART are defined in <sys/ioctl.h> and pass an argument of type chkpnt_devbuf_t, as defined in <convex/chkpnt.h>. The definition looks like this:

```
/*
 * Generic device chkpnt buffer for IOCCHKPNT and IOCRESTART ioctls.
 */
typedef struct {
    char chkpnt_devbuf[256];
} chkpnt_devbuf_t;
```

The chkpnt utility issues the IOCCHKPNT ioctl once for each unique file descriptor (not shared via a dup or exec call) that the target process holds open for a device. The device driver stores information describing the current state of the device into the chkpnt_devbuf_t buffer. This information is written to the checkpoint file to be used later by restart.

The restart utility opens the device file with the same "open" flags as were used when the target process opened the original device before it was checkpointed. The restart utility then issues an IOCRESTART ioctl with a pointer to the chkpnt_devbuf_t stored in the checkpoint file by chkpnt. The driver then uses the information stored in the buffer to restore the device to the state at the time of checkpoint. Neither chkpnt nor restart interpret the data stored in chkpnt_devbuf_t in any way.

ConvexOS Devices That Support Checkpoint Restart

This section describes the standard ConvexOS devices that support checkpoint restart and gives a brief summary of the device state that is saved and restored during checkpoint and restart. This information is intended to provide guidance for writing custom device drivers.

The standard ConvexOS devices that support checkpoint restart are:

- all normal terminal devices.
- the control terminal device /dev/tty.
- the slave side of pseudo-terminals.
- raw tape devices.

- the null device /dev/null.

During checkpointing, the terminal devices save the current settings for all special control characters (e.g. erase character and interrupt character), the window size, the settings for control flags (e.g. ECHO and ICANON), and the line discipline. The raw tape devices save the current tape position (tape file number and record number within the file) and settings for the user-selectable options (e.g., retry on errors and ignore EOT). The null device does not have any device state to save; it returns success for the IOCCHKPNT and IOCRESTART ioctls.

Example of Custom Device Driver Code

The following code fragment is sample user-level code to checkpoint and restore device state in much the same way that the chkpnt and restart utilities do:

```
#include <sys/ioctl.h>
#include <convex/chkpnt.h>
int fd;
chkpnt_devbuf_t cb;

fd = open(DEVPATH, O_RDONLY);
if (ioctl(fd, IOCCHKPNT, &cb) < 0) { /* invoke device driver */
    perror("IOCCHKPNT");
    exit(1);
}
printf("device state is checkpointed\n");
if (ioctl(fd, IOCRESTART, &cb) < 0) { /* invoke device driver */
    perror("IOCRESTART");
    exit(1);
}
printf("device state is restored\n");
```

The device is free to interpret the chkpnt_devbuf_t ioctl buffer in any convenient form. For example, the ConvexOS raw tape driver overlays this buffer with a struct tapechkpnt of the form:

```
struct tapechkpnt {
    int         t_version;          /* tape checkpoint version */
    daddr_t     t_file;             /* current file on tape */
    daddr_t     t_rec;              /* current record on tape */
    int         t_retry;            /* current retry status */
    int         t_eot;              /* current EOT behavior */
};
```

The driver handles the IOCCHKPNT ioctl like this:

```
case IOCCHKPNT:
    /*
     * Save off the checkpoint information.
     */
    chkpt = (struct tapechkpnt *)pdata;
    if (ucb->un.tp.lost == TRUE)
        return EAGAIN;

    chkpt->t_version = ucb->un.tp.chkpt_info.t_version;
    chkpt->t_file    = ucb->un.tp.chkpt_info.t_file;
    chkpt->t_rec     = ucb->un.tp.chkpt_info.t_rec;
    chkpt->t_retry   = ucb->un.tp.chkpt_info.t_retry;
    chkpt->t_eot     = ucb->un.tp.chkpt_info.t_eot;
    break;
```

The IOCRESTART ioctl is handled like this:

```
case IOCRESTART:
    /*
     * Restore the Checkpoint information.  Make recursive calls to
     * taioctl to reposition the tape to the checkpointed location.
     */
    chkpt = (struct tapechkpnt *)pdata;

    /* verify checkpoint version */
    if (chkpt->t_version != TAPECHKPNT_VERSION)
        return EINVAL;

    /* restore retry attribute */
    if ((retval = taioctl(dev, (chkpt->t_retry == RETRY_NORMAL) ?
                          MTIOCRTY : MTIOCNRTY, NULL, flags)) !=0)
        return retval;

    /* restore eot attribute */
    if ((retval = taioctl(dev, chkpt->t_eot ? MTIOCNEOT : MTIOCEOT,
                                              NULL, flags)) !=0)
        return retval;

    /* rewind the tape */
    mtcommand.mt_op = MTREW;
    mtcommand.mt_count = 1;
    if ((retval = taioctl(dev, MTIOCTOP, &mtcommand, flags)) != 0)
        return retval;

    /* advance to the correct file */
    mtcommand.mt_op = MTFSF;
    mtcommand.mt_count = chkpt->t_file;
    if (chkpt->t_file && ((retval = taioctl(dev, MTIOCTOP, &mtcommand,
                                                  flags)) != 0))
        return retval;

    /* skip backwards if record count is negative */
    if (chkpt->t_rec < 0 && chkpt->t_file > 0) {
       mtcommand.mt_op = MTBSR;
       mtcommand.mt_count = 1;
       taioctl(dev, MTIOCTOP, &mtcommand, flags);

       if (chkpt->t_rec == -1)
           break;

       mtcommand.mt_op = MTBSR;
       mtcommand.mt_count = -(chkpt->t_rec + 1);
       if (chkpt->t_rec && ((retval =
                     taioctl(dev, MTIOCTOP, &mtcommand, flags)) != 0))
           return retval;

    }
    /* otherwise skip forward to the correct record */
    else {
          mtcommand.mt_op = MTFSR;
          mtcommand.mt_count = chkpt->t_rec;
          if (chkpt->t_rec && ((retval =
              taioctl(dev, MTIOCTOP, &mtcommand, flags)) != 0))
          return retval;
    }
    break;
```

C Programming Example

The programming example in Figure 3-1 below demonstrates a call to `chkpnt()` and to `restart()`. In this example, the process forks to create a child. The child is then checkpointed and restarted by the parent.

The program contains four functions: `main()`, `do_child()`, `do_parent()`, and `sighandler()`. This is what these functions do:

- main()
  - Makes a `sigmask()` call to block delivery of the SIGCONT signal that is sent by `do_parent()` to the child process. (This is done to prevent race conditions that could occur if the signal is received before the child issues the `sigpause()` call.)
  - Forks a new instance of itself.
  - If the `fork()` returns a negative value, it failed; the program then prints an error message and exits.
  - If the `fork()` returns 0, the process is the child; it then calls `do_child()`.
  - If the `fork()` returns a nonzero, nonnegative value, this is the pid of the child (and the process is the parent); it then calls `do_parent()`.
- do_child()
  - The call to `signal()` establishes SIGCONT as the signal for `sigpause()` to wait for; when this signal is received, `sighandler()` will be called.
  - Prints a message: "Before checkpoint."
  - Makes a `sigpause()` call to suspend the process; while it is paused, the process will be checkpointed, killed, and restarted by its parent.
  - Once the process is restarted, it is released from its paused state by a SIGCONT signal sent by its parent. (The `sighandler()` function will also be called.)
  - It then prints a message: "After restart." and exits.
- do_parent()
  - Unblocks the SIGCONT signal
  - Checkpoints the child process and then kills it (the child is also killed if the checkpoint fails).
  - Restarts the child, and waits for it to exit.

The `chkpnt()` and `restart()` calls in the example are discussed after the listing.

Figure 3-1: C Programming Example

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <chkpnt.h>
#include <sys/wait.h>

main ()
{
        pid_t pid;                              /* Process-id of forked child */

        void do_child();
        void do_parent();

        sigblock (sigmask (SIGCONT));/* Block delivery of SIGCONT until */
                                        /* the sigpause call in do_child(). */
                                        /* The child would wait forever if */
                                        /* the signal is delivered before */
                                        /* the call to sigpause. */

        pid = fork ();                          /* Create a child process */
        if (pid < 0) {
                perror ("unable to fork child process");
                exit (1);
        }

        if (pid == 0)
                do_child ();
        else
                do_parent (pid);
}

/***
 *      void do_child():
 *
 *      Wait for the parent to checkpoint and restart.
 */
void do_child ()
{
        void sighandler ();             /* Signal handler for SIGCONT */

        signal (SIGCONT, sighandler);
        printf ("Before checkpoint.\n");
        sigpause (0);                           /* Wait for restart */
        printf ("After restart.\n");
        exit (0);
}
```

Figure 3-1 (continued):

```
/***
 *       void do_parent():
 *
 *       Checkpoint and restart child process.
 */
void do_parent (kid)
pid_t kid;                                      /* The pid of forked child */
{
        pid_t rpid;                             /* Pid of restarted proc */

        sigunblock (sigmask (SIGCONT));

        if (chkpnt (CHKPNT_PROC, kid, "ex1", CHKPNT_SIGROOT|CHKPNT_FORCE, SIGKILL) < 0) {
                fprintf (stderr, "chkpnt failed: %s\n", strerror (errno));
                /* Kill the child because the checkpoint failed */
                kill (kid, SIGKILL);
                exit (1);
        }

        /*
         * We just killed child with the chkpnt().  We must wait
         * for it to exit.
         */
        if (waitpid (kid, (int *) 0 , 0) < 0) {
                fprintf (stderr, "waitpid failed: %s\n", strerror (errno));
                exit (1);
        }

        /*
         * Restart child from the checkpoint file.  A SIGCONT will
         * be sent.  This will end the child's sigpause call and cause
         * it to exit.
         */
        if ((rpid = restart ("ex1", 0)) < 0) {
                fprintf (stderr, "restart failed: %s\n", strerror (errno));
                exit (1);
        }

        /*
         * Wait for the restarted child to exit.
         */
        if (waitpid (rpid, (int *) 0 , 0) < 0) {
                fprintf (stderr, "waitpid failed: %s\n", strerror (errno));
                exit (1);
        }

        exit (0);
}

/***
 *       void sighandler():
 *
 *       Signal handler for SIGCONT.
 */
void sighandler (s)
int s;
{
    printf ("Got signal %d.\n", s);
}
```

The Checkpoint Call

In Figure 3-1, the following call to `chkpnt()` in `do_parent()` checkpoints the child process:

```
(chkpnt (CHKPNT_PROC, kid, "ex1",
  CHKPNT_SIGROOT|CHKPNT_FORCE, SIGKILL)
```

These are the arguments of the call:

- CHKPNT_PROC—Checkpoint only the target process specified by pid.
- kid—a variable that contains the pid of the child process that is to be checkpointed.
- ex1—the name of the checkpoint file (since this is a relative and not an absolute pathname, a file called "ex1" will be created in the current working directory).
- CHKPNT_SIGROOT | CHKPNT_FORCE—The first option causes a signal (specified in the last parameter) to be sent to the root process after checkpointing is complete; the second option causes checkpointing to be performed even if error conditions are encountered.
- SIGKILL—Signal to send to root process (since this is a "kill" signal, the target is killed).

The Restart Call

In Figure 3-1, the following call to `restart()` restarts the child process:

```
restart ("ex1",0)
```

These are the arguments of the call:

- ex1—the checkpoint file for the process to be restarted is in the current directory, and is called "ex1".
- 0—Since the `flags` parameter is zero, no options will be used.

Fortran Programming Example

The following is an example of a Fortran program that uses the Fortran chkpnt library call.

Figure 3-2 Fortran Programming Example

```
C---------------------------------------------------------------------C
C A Fortran example of how to have a program checkpoint itself        C
C---------------------------------------------------------------------C
       implicit none    ! No implicitly defined variables !

C---------------------------------------------------------------------C
C This section sets some constants for checkpoint                      C
C The values of these are found in /usr/include/chkpnt.h               C
C---------------------------------------------------------------------C
       integer CHKPNT_PROC
       parameter (CHKPNT_PROC = '0001'x)
       integer CHKPNT_KILL
       parameter (CHKPNT_KILL = '0010'x)

C---------------------------------------------------------------------C
C Functions used in this code                                          C
C---------------------------------------------------------------------C
       integer getpid
       integer chkpnt

C---------------------------------------------------------------------C
C Variables used in this code                                          C
C---------------------------------------------------------------------C
       integer  ret_val
       integer  pid

C---------------------------------------------------------------------C
C This section will checkpoint itself into a file named ex1            C
C If the system crashes it will be possible to use restart(1) to       C
C restart from the file ex1.                                           C
C---------------------------------------------------------------------C
       pid = getpid()
       ret_val = chkpnt(CHKPNT_PROC, pid, 'ex1', 0)
       if (0 .NE. ret_val) then
         write(*,*)'chkpnt 1 failed'
         if (-1 .EQ. ret_val) then
           call perror("chkpnt_1")
           call exit(1)
         else
           write(*,*)'ret_val = ',ret_val
           call exit(2)
         endif
       endif
```

Figure 3-2 (continued):

```
C----------------------------------------------------------------------C
C This section will checkpoint itself into a file named ex2 with the   C
C option that causes this program to be killed when the checkpoint    C
C is completed. After the checkpoint executes the file ex2 will exist  C
C but the caller will notice that this code has been killed. It will   C
C then be available for restarting using restart(1).                  C
C----------------------------------------------------------------------C
      pid = getpid()
      ret_val = chkpnt(CHKPNT_PROC, pid, 'ex2', CHKPNT_KILL)
      if (0 .NE. ret_val) then
        write(*,*)'chkpnt 2 failed'
        if (-1 .EQ. ret_val) then
          call perror("chkpnt_2")
          call exit(1)
        else
          write(*,*)'ret_val = ',ret_val
          call exit(2)
        endif
      endif

      end
```

Error Messages

This appendix lists the error messages that can be produced by the Checkpoint Restart utilities. Messages are listed separately for chkpnt and restart.

Because the content of many error messages is variable, the messages are listed here as printf format strings, where "%d" stands for a numeric (decimal) element in the actual error message, and "%s" stands for a character string. In addition, "%m" is the content of an error message string that corresponds to the value of the errno variable when the message is sent; this variable usually contains specific information about what went wrong.

Typically, these messages start with either a process pid (represented as %d) or a checkpoint file name (represented as %s). The combination %s(%d) refers to the command name (%s) and pid (%d).

The error messages are listed in alphabetical order sorted by the first significant word (i.e., not a format specifier) in each message. The portions of the error messages not composed of format specifiers are printed in bold face.

Checkpoint Error Messages

The following messages may be given if an error condition occurs while running chkpnt.

%d: %m

The pid (%d) specified on the command line cannot be checkpointed for the reason given in %m.

%s: %m

A stat of the checkpoint directory (%s) specified on the command line has failed. Check to make sure that the directory exists and is readable.

%s(%d) **fd** %d **(dev** %d **ino** %d **size** %lld**): fdpath failed:** %m

chkpnt is unable to determine the path of the regular file referenced by a file descriptor (fd) held by the process.

An NFS bug may cause this to happen for file descriptors that reference a remote file mounted over NFS. A work-around for this is to wait, then try the checkpoint again.

%s(%d): %s **(fd** %d**):** %s **file lock held**

An exclusive or shared file lock (obtained via flock) is held on the file which this file descriptor (fd) references.

%s(%d): %s **(fd** %d**): record lock held beginning at byte** %ld

A record lock obtained via fcntl is held by the process on the file which this file descriptor references.

`%s(%d) fd %d: attempt to attach to pipe failed: %m`

The "write" end of a pipe does not appear within the process hierarchy, and does exist outside the hierarchy.

`can't get fd flags for fd %d of %s(%d): %m`

The fcntl F_GETFD has failed for this file descriptor.

`can't get info for fd %d of %s(%d): %m`

chkpnt is unable to get file descriptor information for the indicated file descriptor.

`%s(%d): can't get map file path for region %05x: %m`

chkpnt is unable to determine the pathname of a mapped file.

`%s(%d): can't get syscall context (tid %d)`

chkpnt is unable to retrieve the current register state for a thread of this process. This message will follow a more specific message indicating the error that occurred.

`%s(%d): can't get vector register state (tid %d)`

chkpnt is unable to retrieve the current vector register state for a thread of this process. This message will follow a more specific message indicating the error that occurred.

`%s(%d): cannot checkpoint a process created with vfork`

The indicated process has been created by its parent using the `vfork` system call and the parent is also being checkpointed. This situation cannot be checkpointed.

`%s(%d): cannot checkpoint prepaged or non-swappable process`

The indicated process was created from an executable that was marked as prepaged or non-swappable (refer to the *ld*(1) man page in the *ConvexOS Programmer's Reference*). This type of process cannot be checkpointed correctly. A checkpoint of this process can be forced with the "-F" option, but when restarted, the process will no longer be prepaged or non-swappable.

`checkpoint directory %s conflicts with checkpoint file path %s`

The checkpoint directory specified with the "-d" command line option and the path to the checkpoint file specified with the "-f" option point to different directories. If both options are given the directory must be identical. Specify only the file name and not the directory path with the "-f" option when using the "-d" option to give a checkpoint directory.

`checkpoint file path exceeds max of %d`

The sum of the lengths of the checkpoint directory path and the checkpoint file name exceeds the system maximum length for file paths. Specify a shorter checkpoint directory path or a shorter checkpoint file name.

`%s(%d) chkpnt region failed (tid = %d)`

chkpnt is unable to read a memory region for the process.

`%s(%d) close %d (%d): %m`

chkpnt is unable to close the file descriptor obtained from the inferior process.

`%s(%d) fd %d %s copy to %s: %m`

An attempt to copy a file used by the process failed. The file was selected to be copied with the -C command line option, the interactive mode copy command, or because it was a file that was unlinked at the time of the checkpoint.

`%s: device does not support checkpoint capability`

The process being checkpointed has a file descriptor open to a device that does not support checkpointing.

%s(%d) fd %d %s: device position cannot be determined: %m

The process being checkpointed has a file descriptor open to a tape device, and that device is unable to determine the current tape position. This can happen if the application has backspaced over a tape mark, since the tape driver can then no longer determine the current record within the file.

%s(%d) fd %d %s: drain fifo close: %m
%s(%d) fd %d %s: drain fifo open: %m
%s(%d) fd %d %s: drain fifo: %m

These messages mean that an error occurred while attempting to read data from a named pipe used by the process being checkpointed.

%s(%d) fd %d %s failed IOCCHKPNT ioctl: %m

The process being checkpointed has a file descriptor open to a device that does not support checkpointing.

%s(%d) fd %d: failed to reattach pipe from %s(%d) fd %d

`chkpnt` failed to get a copy of a file descriptor that referenced the "write" end of the pipe in the checkpointed hierarchy.

%s: file exists

The checkpoint file already exists. `chkpnt` will overwrite existing checkpoint files if the "-F" command line option is used.

%s(%d) fd %d fstat: %m

The `fstat` system call failed on the indicated file descriptor of the indicated process.

%s(%d): getpattr: %m

`chkpnt` cannot get the process attributes for the indicated process.

invalid process file descriptor: fd %d: %m

The file descriptor number given with the `-P` command line option or with the CHKPNT_PFD option to `chkpnt(3)` does not reference a process (i.e. returned from `pattach`), or the process has not been attached using the O_EXCL option to `pattach`, or the process is not currently in a stopped state.

(%s)%d fd %d is a symbolic link

`fstat` for the indicated file descriptor has returned a file type indicating the file descriptor references a symbolic link.

%s(%d) is checkpointable

The message is printed when the -v option is used with the -n option to verify that a process is checkpointable.

%s(%d) fd %d is socket

The indicated file descriptor references a socket. This file descriptor cannot be checkpointed.

logfile read failed

An error was encountered while attempting to read the logfile. This could be caused by I/O errors on the file, or a syntax error was discovered. If you have modified your logfile, verify that the changes are correct.

logfile write failed

An error was encountered while attempting to write the logfile.

%s(%d) fd %d: lseek failed

`chkpnt` cannot `lseek` on the indicated file descriptor to determine the current file seek position.

`%s(%d): map file %s: %m`

chkpnt is unable to `stat` the mapped file. This could happen if the target process has unlinked the file after mapping it into its address space.

`%s(%d): %d memory regions exceeds max of %d`

The process to be checkpointed has more than the maximum number of memory regions allowed by chkpnt. This process will not checkpoint or restart correctly.

`missing process id specification`

No process ID was given on the command line.

`mregion %d: %m`

The `mregion` system call has failed for the indicated process.

`%s(%d): %d open fds exceeds max of %d`

The process to be checkpointed has more than the maximum supported number of file descriptors open.

`%s(%d) fd %d pipe close: %m`

An error was encountered closing a pipe file descriptor.

`%s(%d) fd %d: pipe destination is outside of selected processes`

The indicated file descriptor of the process is the "write" end of a pipe and no reference to the "read" end has been found within the process hierarchy. Check that all the processes of the application belong to the process hierarchy specified to be checkpointed.

`%s(%d) fd %d pipe drain: %m`

chkpnt has found a pipe file descriptor that has buffered data (i.e. written by the producer, but not yet read by the consumer) and has encountered an error while trying to read the data from the pipe.

`%s(%d) fd %d: pipe source is outside of selected processes`

The indicated file descriptor of the process is the "read" end of a pipe and no reference to the "write" end has been found within the process hierarchy. Check that all the processes of the application belong to the process hierarchy specified to be checkpointed.

`%s(%d) fd %d: process file descriptor not checkpointable`

The target process has a process file descriptor obtained via `pattach`. This makes the process non-checkpointable.

`%s(%d) process seek va 0x%x: %m`

While reading the process's virtual address space, an error was encountered setting the current seek offset to the indicated process virtual address.

`read %s(%d) va 0x%x c %d: %m`

An error occurred while reading the indicated process virtual address.

`%s(%d): region 0x%x overlaps with restart address space`

This process contains valid virtual addresses in the range 0xb0000000 to 0xbfffffff. This address range will conflict with the address space of `restart` so this process will not restart correctly.

`%s(%d) fd %d restore seek pos %lld: %m`

chkpnt encountered an error while restoring the file seek pointer for the indicated file descriptor. The regular file referenced by this descriptor has been unlinked by the target process so no file name is available.

`%s: seek: %m`

While writing the checkpoint file, chkpnt encountered an error while attempting to seek in the checkpoint file for a block that contains only zeroes.

`unable to close logfile %s: %m`

An error occurred during close while attempting to read or write the logfile.

`unable to open logfile %s: %m`

An error occurred during open while attempting to read or write the logfile.

`unknown file type 0x%x for fd %d of %s(%d)`

chkpnt has encountered a file descriptor for which the file type is unknown. fstat on the file descriptor may have returned an unknown value in the st_mode field.

`%s: write (string table): %m`

An error occurred while writing the string table to the checkpoint file.

`%s: write proc region: %m`

An error occurred while writing a virtual address region of the target process to the checkpoint file.

Restart Error Messages

The following error messages may be given if error conditions are encountered while running `restart`.

`%s: %m`

The checkpoint file path (`%s`) specified on the command contains a slash and `restart` has encountered an error with `stat` on the directory portion of the path. Check to see that the checkpoint file path has been specified correctly and that the user has permission to search the directory.

`%s: can't find S_CONTEXT section`

`restart` is unable to find a section header with a type of S_CONTEXT in the checkpoint file. The checkpoint file has been corrupted or truncated when written by `chkpnt`. This may happen if the file system became full during the checkpoint.

`%s(%d): can't restore pending signal %d: kill: %m`

The specified signal was pending for the target process at the time of checkpoint. `restart` has failed to deliver the signal to the current process via the `kill` system call.

`%s(%d): chdir %s: %m`

`restart` has failed to restore the current working directory of the target process. Check to see that the specified directory exists and directory search permissions are allowed.

`%s(%d): chdir tmpcwd %s: %m`

The target process had an unlinked current working directory at the time of checkpoint. In this case, `restart` makes a temporary directory using the pattern "/tmp/restartdirXXXXXX", restores the process's cwd to this directory and removes it. `restart` has failed to restore the cwd to this temporary directory. Check that the current umask for the process that invoked `restart` allows owner search permission for newly created directories. Refer to the *umask*(2) man page in the *ConvexOS Programmer's Reference.*

`%s: checkpoint cputype (%d) different from current machine cputype (%d)`

This message is printed if the -v option is used with `restart`. It is an informational message notifying the user that the CPU type of the machine on which this checkpoint file was created does not match the current machine's cputype. Refer to the *getsysinfo*(2) man page of the *ConvexOS Programmer's Reference.*

`%s(%d): chroot %s: %m`

The target process has a different root directory (refer to the *chroot*(2) man page of the *ConvexOS Programmer's Reference*) and `restart` was unable to restore it. Verify that the specified directory exists. Note that `restart` will only attempt to change the root directory if invoked by the super user.

`close fifo %s: %m`

`restart` could not close the specified named pipe after restoring the checkpointed data

`%s: contains instructions unsupported by this machine's architecture`

The checkpoint file came from process that uses intrinsic or parallel instructions and the current machine does not have support for the instruction types.

current gid %d differs from checkpoint file gid %d
current uid %d differs from checkpoint file uid %d

These warnings are printed if a process is checkpointed with one uid (gid) and restarted with another. As indicated by this warning message, the restarted process has different permissions than it did before it was checkpointed. It may not be able to open certain data files or send signals to certain processes.

%s(%d): dup %d failed: %m
dup %d failed: %m
%s(%d): dup2 %d %d failed: %m

These messages indicate that, in an attempt to restore the original file descriptors, the dup system call failed.

error: close %s %d failed: %m

`restart` has failed in an attempt to close the indicated file descriptor.

%s(%d): fcntl %d 0x%x 0x%x: %m

`restart` uses the F_SETFL and F_SETFD options of the `fcntl`(2) system call to restore file descriptor attributes. If the system call fails, this message is printed. The first hexadecimal field is the value of the flags for F_SETFL, the second is the value of F_SETFD flags. Refer to the *fcntl*(2) man page in the *ConvexOS Programmer's Reference*.

fifo open %s: %m

`restart` has failed to open the indicated named pipe. Be sure the named file exists and check the permissions on this file. Refer to the *mknod*(1) man page of the *ConvexOS Programmer's Reference*.

%s(%d): file copy from %s to %s failed: %m

`restart` failed to copy a regular file back to the original path. Check that the destination directory exists and that you have permission to create files in that directory. Check that you have permission to write to the destination file, and that the file system has sufficient free space.

fork: %m

`restart` was unable to fork a child to become the head of the new process hierarchy.

fork: failed for %s(%d): %m

A process in the restart hierarchy was unable to fork a child to become the indicated process.

%s(%d): internal error %d: fds curfd %d dstfd %d

`restart`'s internal file descriptor table was found in an inconsistent state.

internal error - double fd close %s fd %d
internal error - double fd open %s fd %d

These messages indicate that `restart`'s internal file descriptor table was found in an inconsistent state. A new file descriptor was opened for which an entry exists.

%s(%d): internal error: anon region %05x not in region table

The indicated region is a MAP_ANON region that is shared with other processes. This region was not found in `restart`'s internal region table.

internal error: link file type unexpected!

A file descriptor was found with type S_IFLNK indicating a symbolic link. Symbolic links should always appear as the linked-to file, never as a symbolic link.

%s: internal region mismatch

The number of regions described in the checkpoint file does not match the number listed in the checkpoint header. The checkpoint file has been corrupted or truncated

when written by chkpnt. This may happen if the file system filled during the checkpoint.

%s: invalid MAP_TYPE (0x%x)

A region was found that was not one of MAP_FILE, MAP_ANON, MAP_THREAD, or MAP_DEVICE.

invalid option combination: -t option specified without -W

restart was invoked with the -t option but without the -W option. This option combination will leave a restarted process hung since without -W restart will be the parent of a traced child but will not do any tracing.

ioctl IOCRESTART %s (fd %d): %m

The IOCRESTART ioctl has failed on the file descriptor for the indicated device. This may be caused by restarting an application that uses a device that does support the IOCCHKPNT ioctl but does not support the IOCRESTART ioctl.

%s(%d): lseek %s: %m

restart has encountered an error while opening or setting the lseek file position. Most likely is the file (%s) used by the process has been removed or is no longer readable.

%s(%d): map file fd %d close: %m

restart was unable to close a file descriptor open to map in a region of type MAP_FILE | MAP_SHARED.

%s(%d): MAP_ANON mapped file %s: %m

restart creates and opens files with the pattern /tmp/restartmemXXXXXX to restore shared memory regions of type MAP_ANON. The open for one of these files failed for the indicated reason. This error may be generated if the files were removed from /tmp between the time they were created by restart and the time they were opened for use.

MAP_DEVICE memory unsupported

A memory region of type MAP_DEVICE was found in the checkpoint file for the process. MAP_DEVICE memory regions cannot be restored.

%s(%d): mapped file %s: %m

The MAP_FILE | MAP_SHARED region was mapped from this specified file. restart was unable to open this file for the reason given.

%s: missing S_CHKPNT section

The section header containing checkpoint information was not found in the given checkpoint file. The checkpoint file has been corrupted or truncated when written by chkpnt. This may happen if the file system became full during the checkpoint.

%s(%d): mkdir tmpcwd %s: %m

The checkpointed process had an unlinked current working directory at the time it was checkpointed and restart was unable to create the temporary directory in /tmp to use a the restarted process's current working directory.

mmap 0x%x 0x%x 0x%x: %m

restart was unable to map in the indicated memory region. Be sure that the checkpointed process did not contain memory regions in the range 0xb0000000-0xbfffffff since these overlap with restart's own memory regions.

mregion: %m

restart was unable to obtain memory region information about itself.

mremap 0x%x 0x%x 0x%x: %m

restart was unable to remap the indicated memory region to set the regions limit parameter.

mremap vl %x vs %x pr %x sh %x: %m

restart was unable to remap the indicated memory region.

munmap: %m

restart was unable to unmap its original stack region.

new stack mmap: %m

restart was unable to map its new stack.

%s**: not a directory**

This warning is printed when the checkpoint file path contains a "/" and the directory component of this path is not a directory.

%s**: not checkpoint file format**

The specified file exists but is not a checkpoint file. Check that the file specified on the command line is correct and is a valid checkpoint file (this can be tested with the file utility; refer to the *file*(1) page in the *ConvexOS Programmer's Reference*).

%s**: open failed:** %m

The checkpoint file cannot be opened. Check that the file exists and that you have read permission for the file (refer to the *chmod*(1) man page of the *ConvexOS Programmer's Reference*).

%s**: open:** %m

restart was unable to reopen the checkpoint file. Be sure that the checkpoint file exists and that you have read permission for the file (refer to the *chmod*(1) man page of the *ConvexOS Programmer's Reference*). One reason for this message is the checkpoint file was removed while restart was attempting to restart the process.

%s(%d)**: open** %s**:** %m

restart has encountered an error while opening or setting the lseek file position. Most likely is the file (%s) used by the process has been removed or is no longer readable.

pid %d **for** %s **in use, retrying**

A process with the process identifier needed by restart is currently active in the system. This verbose message is printed once a second for 10 seconds while restart waits for the process to exit, making the process ID available.

pipe close failed: %m

restart was unable to close a pipe file descriptor.

pipe create failed: %m

restart was unable to create a pipe file descriptor for the process being restarted.

read failed: unable to fill region from chkpnt file: %m

restart was unable to read a memory section from the checkpoint file. The checkpoint file has been corrupted or truncated when written by chkpnt. This may happen if the file system became full during the checkpoint.

%s**: requires IEEE floating point hardware unavailable on this machine**

This process uses IEEE floating point and was checkpointed on a machine that supported IEEE floating point mode, but the machine on which the process is being restarted does not have IEEE floating point support. Move this checkpoint file to a machine that has the necessary IEEE hardware support (refer to the *getsysinfo*(1) man page of the *ConvexOS Programmer's Reference*) to restart the process.

`restart comm area init failed`

`restart` was unable to map in the shared memory used to communicate with other processes of the restart hierarchy.

`%s: restarting prepaged or non-swappable process as normal process`

This process was prepaged or non-swappable, but a checkpointed was forced. The process cannot be restarted as prepaged or non-swappable, but is restarted as a normal demand-paged process. This verbose message is printed if the -v option is specified.

`%s(%d): restoring current baud rate: %s`

This verbose message is printed when the baud rate stored in the checkpoint file does not match the current baud rate and the input baud rate matches the output baud. In this case, `restart` uses the baud rate of the user's terminal and prints this message if the -v option is specified.

`%s(%d): rmdir tmpcwd %s: %m`

The checkpointed process had an unlinked current working directory at the time it was checkpointed and `restart` was unable to remove the temporary directory in /tmp created for the process's current working directory.

`seek failed: unable to fill region from chkpnt file: fd %d off %llx: %m`

`restart` was unable to seek to the proper location within the checkpoint file to read the contents of a memory region. The checkpoint file has been corrupted or truncated when written by chkpnt. This may happen if the file system became full during the checkpoint.

`%s(%d): setaid(%d): %m`

`restart` failed to restore the saved activity ID.

`%s(%d): setgroups: %m`

`restart` failed to restore the group access list.

`%s(%d) setitimer(%d): %m`

`restart` failed to restore an interval timer. The values 0, 1, or 2 indicate which timer failed: real, virtual, or profiling, respectively.

`%s(%d): setpattr: %m`

`restart` has failed to restore the process attributes.

`%s(%d): setpgrp %d: %m`

This `restart` process has failed to join a process group that should have been created by another process in the restarted hierarchy. This will happen if the process group leader is unable to get its required process ID and thus is unable to become the process group leader.

`%s(%d): setpgrp to %d: %m`

`restart` was unable to restore the process group. This will happen if `restart` could not get its required process ID.

`setpid: pid %d for %s in use`

`restart` was unable to restore the process ID. This will happen if another process with this pid is running in the system.

`%s(%d): setpriority %d: %m`

`restart` has failed to restore the process priority.

`%s(%d): setregid(%d, %d): %m`

`restart` has failed to restore the process real and effective group ID.

`%s(%d): setreuid(%d, %d): %m`

restart has failed to restore the process real and effective ID.

`%s(%d): shared mem tmp file create failed: %m`

restart has failed to create a temporary shared memory file using the pattern "/tmp/restartmemXXXXXX".

`sigstack: %m`

restart has failed to restore the process signal stack.

`sigvec (%d): %m`

restart has failed to restore the signal vector state for the indicated signal.

`socket file descriptor cannot be restored`

A file descriptor for a socket was found in the checkpoint file and cannot be restored.

`%s(%d): tcsetattr: %m`

The baud rate or window size for the checkpointed process is different from the current baud rate or window size and restart was unable to set these terminal attributes to the current values.

`tcsetpgrp %s(%d) fd %d pgrp %d: %m`

The restart process restoring the root process in a restart hierarchy was unable to set the terminal process group to the indicated value. This will happen if the process group leader is unable to get its required process ID and thus is unable to become the process group leader.

`unable to restore fifo data buffer: %m`

restart has failed to restore checkpointed data for a named pipe.

`unable to restore pipe data buffer: %m`

restart has failed to restore checkpointed data for a pipe.

`unknown file type 0x%x for %s`

The file type for a file descriptor for the indicated file is unknown.

`%s(%d): using current baud rates: ispeed %s ospeed %s`

This verbose message is printed when the baud rate stored in the checkpoint file does not match the current baud rate. In this case, restart uses the baud rate of the users terminal and prints this message if the -v option is specified.

`%s(%d): using current window size: %2d rows %2d cols`

This verbose message is printed when the window size stored in the checkpoint file does not match the current window size. In this case, restart uses the window size of the user's terminal and prints this message if the -v option is specified.

`warning: inherited open file descriptor %d`

This warning is printed if restart is invoked with open file descriptors other than 0, 1, and 2. This will happen if restart is invoked from within a process that has not closed its file descriptors.

`%s(%d): warning: only superuser may lower priority to %d`

The checkpointed process was running at a lower scheduling priority (refer to the *setpriority*(2) man page of the *ConvexOS Programmer's Reference*) but has been restarted by a non-root user. The non-root user cannot restore the lower priority. Restart the process as root or have a superuser lower the priority of the running process after it has been restarted.

%s(%d): warning: only superuser may restore root directory %s

The checkpointed process had changed its root directory (refer to the *chroot*(2) man page of the *ConvexOS Programmer's Reference*) but has been restarted by a non-root user. The non-root user can not restore the proper root directory. Restart the process as the superuser.

Reporting Problems

B

This appendix describes procedures for reporting problems to the CONVEX Technical Assistance Center (TAC) and the contact utility. Special procedures for reporting Checkpoint Restart problems are described under , "Reporting Checkpoint Restart Problems" below.

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation question, contact the TAC. This group stands ready to solve such problems.

To reach the TAC via telephone:

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384.
- All other locations, contact the local CONVEX office.

You may also use the online contact utility to report problems to the TAC. This utility is described under "The contact Utility" section on page 2 of this appendix.

Reporting Checkpoint Restart Problems

This section describes procedures specific to CR that will help you to provide the TAC with information needed to diagnose and solve your problem.

In order to analyze a bug in chkpnt or restart, all related checkpoint files and data files should be submitted to the the TAC. To determine which files are needed, follow one of the two methods described below.

Reproducible Problems

If the problem is reproducible (e.g. every checkpoint of an application fails to restart properly), invoke chkpnt with the -C option to copy all regular files to the checkpoint directory, and then copy the contents of the entire checkpoint directory to the tape. This command creates a new checkpoint directory:

```
% mkdir chkpntdir   # make a clean directory
```

Next, checkpoint the target process (substitute the pid of the process for *pid* in the example below). If a process hierarchy is to be checkpointed, include the *-r* option.

```
% chkpnt -C -d chkpntdir pid
```

Mount the tape drive:

```
% tpmount
```

Write the data to tape:

```
% tar c chkpntdir
```

Unmount the tape drive:

```
% tpunmount
```

Unreproducible Problems

If the problem cannot be consistently reproduced, but one particular checkpoint file will not restart correctly, use the "-v" option of `restart` to determine which files are used by the processes being restarted. You can then copy these files to the checkpoint directory.

First, go to the checkpoint directory:

```
% cd chkpntdir
```

Then invoke `restart` with the `-v` option on the problematic file (substitute the name of the checkpoint file that will not restart for *badchkpnt*:

```
% restart -v badchkpnt
```

The verbose output will indicate which files are needed by the process that you are attempting to restart. Copy these files to the checkpoint directory (the current working directory):

```
% cp file1 file2 file3 .
```

(Replace *file1...* with the appropriate file names.)

Then mount the tape, copy the contents of the current directory to it, and unmount the tape:

```
% tpmount
% tar c .
% tpunmount
```

Once the tape has been written, send it to the TAC along with a description of the problem.

The contact Utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it electronically to the TAC. The TAC notifies you within 48 hours that your report has been received.

To use `contact` requires:

- UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- Full path name of the program or utility in question
- Version number of the program or utility in question

UUCP Connection

Before using `contact`, ask your system administrator if your site has a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX-based system to another. The uucp (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

Finding the Program Path Name

To determine the full path name of the program or utility in question, use the `which` command. Figure B-1 illustrates use of the `which` command to find the full path name of the loader (`ld`) utility.

Figure B-1
Using the `which` command

```
> which ld
/bin/ld
>
```

In this example, the full path name of the loader is /bin/ld.

If you use the C shell (`csh`), you can also use the `whence` command to find the program path name. The `whence` command works like `which`, but faster.

For more information on the `which` command, refer to the *which*(1) man page. You can also use the `info` online information system by entering **`info which`** at the system prompt.

Finding the Program Version Number

To determine the version number of the program or utility in question, use the *vers* command. Figure B-2 illustrates use of the *vers* command to find the version number of the loader (`ld`) utility. Enter **`vers`**, then the path name of the program or utility.

Figure B-2
Using the `vers` command

```
> vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader version number is 7.0.

For more information on the *vers* command, refer to the vers(1) man page. You can also use the *info* online information system by entering **`info vers`** at the system prompt.

Using contact

The `contact` utility prompts for the following information:

- Your name, title, phone number, and corporate name
- Name and version of the product
- One-line summary of the problem
- Detailed description of the problem

- Priority of the problem
- Instructions on how to reproduce the problem
- Comments about the problem
- Comments about the documentation relating to the problem
- Files to include in the contact report

Following is a step-by-step discussion of these prompts.

Step 1a

To invoke the contact utility, enter **contact** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software, or documentation question. Figure B-3 illustrates use of the contact command and the resulting system response.

Figure B-3 Beginning a contact session

```
> contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

Step 1b

If there is a .contact file in your home directory, contact skips the first prompt. (Refer to "Using a .contact File" section on page 7 for more information.) Figure B-4 illustrates the contact command and the system response when you have a .contact file in your home directory.

Figure B-4 Beginning a contact session with a .contact file

```
> contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

Step 2

The contact utility prompts for the version number of the product. If you do not know the version number, press **CTRL-Z** to suspend the session.

Use the *which* (or *whence* if you use csh) and *vers* commands to find the version number of the product. Use the fg command to return to the session, and enter the version number in the form X.X or X.X.X.X.

Step 3

The `contact` utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Please make this summary as descriptive as possible in one line.

Step 4

The `contact` utility prompts for a detailed description of the problem. Please make this description as complete as possible. Include source code and a stack backtrace when possible. (Refer to the adb(1) or csd(1) man page for information on obtaining a stack backtrace.) The more information you provide, the quicker the TAC can isolate and solve the problem.

Step 5

The `contact` utility prompts for the priority of the problem. Figure B-5 illustrates this prompt and priority levels from which to choose. You must enter a priority number.

Figure B-5 Specifying the priority of a problem

```
Enter a problem priority, based on the following:
1) Critical-work cannot proceed until the problem is resolved.
2) Serious-work can proceed around the problem, with difficulty.
3) Necessary-problem has to be fixed.
4) Annoying-problem is bothersome.
5) Enhancement-requested enhancement.
6) Informative-for informational purposes only.
>
```

Step 6

The `contact` utility prompts for an explanation of how to reproduce the problem. Please include the command syntax and options you used and anything else you did to make the program run.

Step 7

The `contact` utility prompts for any other pertinent comments. Please include all relevant information.

Step 8

The `contact` utility prompts for suggestions regarding documentation supporting the product. Indicate whether the documentation could be revised to address the problem.

Step 9

The `contact` utility prompts for names of files necessary to reproduce the problem. Figure B-6 illustrates this prompt and sample user response.

Figure B-6 Including files in a contact report

```
Are there any files that should be included in this report (yes | no)?
> yes
Please enter the names of the files, one to a line (^D to terminate)
> test.f
> ~/subroutines/sub.f
>
```

If files specified are small text files, they are automatically included in the contact report. If the files are too large to be included in this report, contact gives further instructions on how to submit these files.

To specify a directory, combine directory files into a single file using the *tar* command (refer to the tar(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

Note — **Tilde-escape sequences (described on page 8) are not recognized in responses to this prompt. In *contact*, a tilde in this section indicates your home directory. This convention is based on use of the tilde for expanding file names in *csh*.**

Step 10

The contact utility prompts you to review, edit, submit, or abort the report. Figure B-2 illustrates this prompt.

Figure B-7
Prompt to review, edit, submit, or abort report

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review	review the text of the contact report. You are then prompted again to select an option.
Edit	edit the text of the contact report. If you choose to edit the report, contact opens your default text editor.
Submit	sends the report to the CONVEX TAC. The TAC notifies you within 48 hours that your report has been received. Choosing this option exits the contact utility and returns you to the shell.
Abort	saves the text of the report in a file named ~/dead.report. Choosing this option exits contact and returns you to the shell.

Tips for Using contact

The contact utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to:

- Use a .contact file
- Abort a contact session

- Resubmit an aborted report
- Suspend a contact session
- Move within contact from one prompt to another
- Use tilde-escape sequences in the contact utility

Using a .contact File

When you invoke contact, it first prompts for your name, title, phone number, and company name. You can, however, create a .contact file to skip this first prompt.

Follow these steps to create a .contact file.

1. Create a .contact file in your home directory.
2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke contact, it automatically includes the .contact file as input for the first prompt and proceeds to the next prompt.

Aborting the Report

To abort a contact report, either press the interrupt key (usually **CTRL-C**) or choose the abort option when prompted by the contact utility. Using **CTRL-C** to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named ~/dead.report.

Submitting the dead.report File

After you abort a contact session, the contact utility saves the report in a file named ~/dead.report. Using the contact command with the -r option automatically merges the contents of the ~/dead.report file into the new contact session. Enter

```
contact -r
```

and contact finds the ~/dead.report file and merges it into the contact report. You can then edit the report. When you end the editing session, contact resumes at the final prompt, which asks you to review, edit, submit, or abort the report.

Suspending a Report

Sometimes it is necessary to stop in the middle of a contact report and return to the shell (for instance, to suspend the contact session to find the program path name or version number). To suspend the contact session, press **CTRL-Z**.

To return to the contact session, press **fg**. Using **CTRL-Z** and the *fg* (foreground) command, you can toggle back and forth between the contact utility and the shell. You cannot, however, use **CTRL-Z** and *fg* to toggle back and forth in the Bourne shell (sh).

Ending a Response

The contact utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the

next prompt, press **RETURN**. Other prompts require more than a one-line response; to move to the next prompt, press **CTRL-D**.

Tilde-Escape Sequences

The `contact` utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. You can use the following tilde sequences within `contact`:

`~e`	start the text editor (defined in the EDITOR environment variable)
`~h`	display a list of available tilde-escape sequences
`~p`	print the `contact` report to the terminal screen
`~r filename`	read the contents of `filename` as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence works only for prompts that allow more than a one-line response.
`~~`	insert a single tilde as the first character in the line

Index

A

access permissions to processes and files 1-7

accounting system, and Checkpoint Restart 1-5

assistance v

associated documents v

audience v

C

-C option of chkpnt 1-3

-C option, of restart will overwrite files 2-16

caution
- -C option of restart will overwrite files 2-16
- cannot call restart() from within multithreaded region 3-4
- checkpoint files not overwritten 1-7, 2-2, 2-4
- chkpnt must be in default directory for Fortran chkpnt() function 3-6
- chkpnt utility must reside in default directory 3-1, 3-5
- contents of open files not automatically saved 1-7
- file names containing colons 2-14
- pid of restarted process must be the same 3-5
- restart utility must reside in default directory 3-4, 3-7
- restarting process with pid other than original 3-5
- restarting processes while running as root 1-12

checkpoint
- C library function 3-1
- C library function, parameters 3-1
- checkpointing chkpnt process itself 1-7
- directory, specifying 2-3
- file, specifying 2-3
- force, in C library function 3-2
- forced 2-4
- interactive mode 2-4, 2-8
- interactive mode, commands 2-9, 2-10
- interactive mode, using log files 2-13
- interactive mode, with process hierarchies 2-12
- job hierarchy 2-5
- kill target process, with C library function 3-2
- log file 2-4
- print debugging output 2-6
- process hierarchies 1-6
- recursive 2-5
- saving open files 2-2
- send signal to target 2-8
- send signal to target, with C function 3-2
- shell command line mode 2-7
- signal to hierarchy with C function 3-2
- signal, send to target hierarchy 2-5
- signal, sending to target 2-5
- single process only 2-5
- suppress warning mesages 2-5
- test for checkpointability 2-5
- using log file 2-4
- verbose output 2-6
- what happens during 1-6

Checkpoint and Restart Utilities 2-1

checkpoint directory, specifying 2-3

checkpoint file 1-6
- assigning a name 1-10
- default name 1-9
- file descriptors saved in 1-3
- format 1-10
- name 1-7, 1-9, 1-10
- name for process hierarchies 1-9
- size 1-10

checkpointability, and
- debugging 1-3
- file descriptors, maximum number of 1-3
- I/O to unlinked files 1-3
- labeled tape I/O 1-3
- memory segments limit 1-3
- mmap system call 1-3
- shared memory 1-3
- socket connections 1-3
- test with -n flag 2-5
- vfork 1-3
- virtual memory addresses 1-3

chkpnt utility
- and chkpnt() C library function 3-1
- command format 2-1

error messages A-1
interactive mode of 2-11
interactive mode, commands 2-10
parameters 2-1
-C parameter 2-2
-d checkpoint_directory parameter 2-3
-f checkpoint_file parameter 2-3
-F parameter 2-4
-I *logfile* parameter 2-4
-i parameter 2-4, 2-8
-i parameter with -L 2-13
-i parameter, interactive mode commands 2-9
-i parameter, with -r 2-12
-j parameter 2-5
-k parameter 2-8
-K *signo* parameter 2-5
-k *signo* parameter 2-5
-L *logfile* parameter 2-4
-n option 2-5
-p *fd* 2-5
-p parameter 2-5
-q option 2-5
-r parameter 2-5
summary 2-1
-v option 2-6
-X option 2-6
shell command line mode 2-7
showing file descriptors 2-11
using the command 2-2

chkpnt() C library function
error codes 3-3
example 3-12
format 3-1
options
CHKPNT_FORCE 3-2
CHKPNT_KILL option 3-2
CHKPNT_PFD 3-2
CHKPNT_SIGFAMILY option 3-2
CHKPNT_SIGROOT 3-2
parameters 3-1
class 3-1
name 3-2
options 3-2
pid 3-1
signo 3-2
uses chkpnt utility 3-1

chkpnt() Fortran library function 3-6
example 3-16
parameters 3-6

colons, in checkpoint log file names 2-14

compatibility, hardware 1-4

contact utility B-2
using B-3

control terminal, and restarted processes 1-14

ConvexOS release compatibility and Checkpoint Restart 1-4

CXbatch and Checkpoint Restart 1-2

D

debugging output, printing during chkpnt 2-6

debugging, and checkpointability 1-3

devices, and checkpointability 1-3

E

error codes
chkpnt() C function 3-3

Error Messages A-1

error messages
checkpoint A-1
restart A-6

F

file
specifying checkpoint file 2-3

file descriptors
control terminal file descriptor 1-14
for devices and pipes 1-7
for files open to checkpointed processes 1-7
information contained in 2-12
interactive chkpntcommands for 2-10
maximum number of 1-3
show open in chkpnt interactive mode 2-11
with interactive restart 2-20

file locks 3-8

files
access during restart 1-12
contents of open files not saved 1-7
copying from checkpoint directory on restart 2-16
copying to checkpoint directory 1-3
file names containing colons, caution 2-14
modified by target process after checkpoint 1-7
names of files copied to checkpoint directory 1-4
pointers to unlinked files after restart 1-12
saved during checkpointing 1-3
saving to checkpoint directory 2-2

Finding version numbers B-3

force checkpointing 2-4

force restart 2-17

fork, wait before exiting after 3-8

Fortran checkpoint and restart functions 3-6

G

gid
  restoring upon restart 1-12

group access list
  restoring upon restart 1-12

H

hardware architecture, and checkpoint restart 1-4

Help v

hierarchies
  checkpoint in interactive mode 2-12

I

interactive checkpointing 2-4

interactive mode
  of chkpnt 2-8
    commands 2-9
    creating log files 2-13
    show open file descriptors 2-11
    with process hierarchies 2-12
  of restart 2-16, 2-18
    commands 2-19
    commands for file descriptors 2-20
    example 2-21

J

job hierarchy, checkpointing 2-5

L

labeled tape I/O, and checkpointability 1-3

labeled tape, and Checkpoint Restart 1-5

library routines, for Checkpoint Restart 1-2

limitations
  accounting 1-5
  compatibility of hardware and software 1-4
  labeled tape 1-5
  non-checkpointable processes 1-3
  of Checkpoint Restart 1-3
  performance 1-3
  pid conflict 1-4
  tape system 1-4

locks, file or memory 3-8

log files
  creating 2-4, 2-13
  names containing colons, caution 2-14
  using 2-4

M

MAP_DEVICE mapped memory segments 3-8

memory locks 3-8

memory segments, maximum number of 1-3

mmap, and checkpointability 1-3

multithreaded region, cannot call restart() within 3-4

N

name, assigning to checkpoint file 1-10

name, of checkpoint file 1-9

O

ordering documents vi

P

pattach() system call, used during checkpointing 1-6

performance limitations of Checkpoint Restart 1-3

pid conflict on restart 1-4

pipes, data in transit through during checkpointing 1-7

portability of checkpoint files 1-4

process hierarchies, checkpointing 1-6

programming guidelines 3-8

purpose of document v

Q

qchkpnt 1-2

qmgr 1-2

qrestart 1-2

quiet option
  restart 2-17

quiet option, of chkpnt 2-5

R

recursive checkpointing 2-5

restart
  and current working directory of target process 1-13
  C library function 3-4
  communication problems after 1-13
  copy back files from checkpoint directory 2-16
  file access during 1-12

file access problems after 1-13
file pointers to unlinked files 1-12
force 2-17
in stopped state 2-17
interactive mode, invoking 2-16
ppid (parent process ID) of target process 1-13
process hierarchies 1-7
quiet option 2-17
recursive, with `restart()` C function 3-4, 3-5
restarting processes on a different system 1-12
restoring target process uid,gid, and group access list 1-12
send signal to target, with `restart()` C function 3-4
sending signal to target 2-16
sending signal to target hierarchy 2-16
`setgid` 1-13
`setuid` 1-13
traced mode 2-17
under Share 1-13
verbose mode 2-17
wait/don't wait 2-17
what happens during restart 1-12

`restart` utility
command format 2-15
error messages A-6
interactive mode 2-18
interactive mode commands 2-19
interactive mode commands, for file descriptors 2-20
interactive mode, example 2-21
parameters
-C parameter 2-16
-F option 2-17
-i option, example 2-21
-i parameter 2-16
-K *signo* parameter 2-16
-k *signo* parameter 2-16
-q option 2-17
-t option 2-17
-v option 2-17
-W option 2-17
-w option 2-17
-z option 2-17
shell command line mode 2-18
using the `restart` command 2-15

`restart()` C library function
error codes 3-5
example 3-12
format 3-4
options
RESTART_DEBUG 3-5
RESTART_SIGFAMILY 3-4, 3-5
RESTART_SIGROOT 3-4
RESTART_SUSPEND 3-5
parameters 3-4
`flags` 3-4
`path` 3-4
`signo` 3-5
`restart` utility must be in default directory 3-4

`restart()` Fortran function 3-7
parameters 3-7

S

security, and Checkpoint Restart 1-7

setpid
and restart 3-8

Share
restarting processes under 1-13

shared memory, and checkpointability 1-3

shell command line mode, of `restart` 2-18

signal
handling, by `restart` 1-15
restarted process sent SIGCONT by default 3-4
send signal to hierarchy, with C function 3-2
send to checkpoint target 2-5
send to checkpoint target hierarchy 2-5
send to restart target 2-16
send to restart target hierarchy 2-16
send with `restart()` 3-5

size of checkpoint file 1-10

socket connections, and checkpointability 1-3

stopped state, restart in 2-17

T

TAC v, B-1

tape system, and checkpoint restart 1-4

Technical assistance v

Technical Assistance Center B-1

terminal
and restart 3-8

terminal name, current 3-8

time, current 3-8

typographic conventions vi

U

uid
restoring upon restart 1-12

unlinked files, I/O to, and checkpointability 1-3

Using this guide v

utilities
    for Checkpoint Restart 1-2

V

verbose mode
    restart 2-17

verbose option, of chkpnt 2-6

Version number, program, finding B-3, B-7

vfork, and checkpointability 1-3

virtual memory, and checkpointability 1-3

CONVEX Checkpoint Restart Guide
Document No. 710-006530-001

Reader's Comments

You are invited to submit your comments concerning the clarity and usefulness of this manual. Your comments are most welcome, and help us continue in our efforts to generate quality customer documentation. Feel free to mark up the documents and send copies of the pages in question. (Please list the page number below for questions and comments):

____

____

____

____

____

____

____

____

From:

Name ____________________ Title ____________________

Company ____________________ Date ____________________

Address ____________________ Phone (____)____________________

City ____________________ ____________________ State ZIP Code

FOR ADDITIONAL INFORMATION/DOCUMENTATION:

Location	Phone Number
Within the Continental U.S.	1(800)952-0379
From Canada	1(800)345-2384
Outside continental U.S.	Contact local CONVEX office

Send Mail Orders to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, Texas 75083-3851 USA

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX COMPUTER CORPORATION
PO BOX 833851
RICHARDSON TX 75083

(Fold Here Second)

(Tape Here)