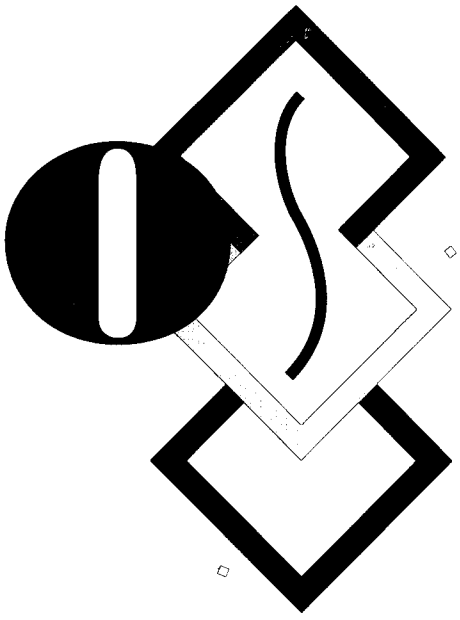


CONVEX

ConvexOS Extensions User's Guide

Second Edition

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214) 497-4000

ConvexOS Extensions User's Guide

Order No. DSW-053

Second Edition
March 1994

CONVEX Press
Richardson, Texas
United States of America

ConvexOS Extensions User's Guide

Order No. DSW-053

Printed in the United States of America

Revision Information for

ConvexOS Extensions User's Guide

Edition	Document No.	Description
Second	710-018330-001	Released with ConvexOS V11.0, March 1994.
First	710-018330-000	December 1991. Initial release.

Contents

Part 2 CONVEX Fair Share Scheduler

Part 3 POSIX Compliance

Figures

Tables

Preface

The *ConvexOS Extensions User's Guide* is a guide and reference for anyone who uses or has questions about the following ConvexOS-related topics:

- Working with large files
- Share scheduling
- POSIX compliance
- Checkpointing and restarting jobs

The commands described in this guide do not require superuser authorization.

Notational conventions

This section describes notational conventions used in this guide. These conventions include:

- Typographical conventions, or what various typefaces mean
- Command syntax and how to distinguish requirements for commands and examples
- Difference between "entering" and "typing" commands
- Identifying cautions and references

Typographical conventions

The following typefaces have special meaning and are used in this guide:

`Bold courier`	Identifies user input in examples.
`Courier`	Identifies input and output, including: • Command names • System calls • Data structures and types • Error messages
Italic	Identifies: • User-supplied variables in a command-line example • New and important terms • Titles of documents
KEYCAP	Indicates keyboard keys to be pressed. For example, **RETURN** refers to the carriage return key. Two **KEYCAP** terms separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-d** indicates that you must press the **d** key while holding down the **CTRL** key.

"Entering" and "typing" commands

In this guide there is a distinction between entering a command and typing a command. This distinction is:

type	Type in the command at the keyboard without pressing the **RETURN** key.
enter	Type in the command at the keyboard and follow by pressing the **RETURN** key.

Command syntax and example conventions

In the following example:

```
command     [options...]     {a | b}     filename[...] \
   ①            ②             ③            ④        ⑤
⑥ ⋮
```

① `command` must be typed as it appears.

② Brackets indicate a part of a command that is optional. Ellipsis indicate that the preceding term can be repeated. The brackets and ellipsis are not typed.

③ Braces indicate a part of a command that is necessary, but that there may be a choice in what is supplied. In this example, values for either variable *a* or *b* must be supplied, but not both. The braces and vertical line are not typed.

④ In this example, *filename* is a variable that must be supplied. The ellipsis enclosed in brackets indicates additional arguments of the same type can be supplied, but are not necessary. Again, neither the brackets nor the ellipsis is typed.

⑤ A backslash at the end of a line indicates that an example continues on the next line of text. If the example is to be entered, do not press **RETURN** until the entire example is typed.

⑥ A vertical ellipsis indicates that part of an example has been omitted. The vertical ellipsis is not typed.

Identifying notes and cautions

"Notes" and "Cautions" tags are used frequently throughout this guide. They are of these formats and have the following meanings:

Note

A Note highlights supplemental information.

Caution

A Caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Accessing associated documentation

Using this software may require information not specific to the tasks described in this document. This section describes where to find additional information and assistance.

Accessing man pages

For more information on ConvexOS, use the online man pages. To view a man page enter:

man *command*

where *command* is any valid ConvexOS command.

To print a man page, enter:

man *command* > *filename*

lpr -P<*printer*> *filename*

References made to man pages throughout this document are in the form:

cat(1)

where the man page's section number, enclosed in parentheses, follows.

Accessing paper documentation

For more in-depth information on ConvexOS, you can order these books from CONVEX Computer Corporation:

- *ConvexOS Primer* (DSW-133), an introduction to ConvexOS for new users
- *Managing ConvexOS: Configuration Guide* (DSW-030), a guide for configuring ConvexOS
- *Managing ConvexOS: Operations Guide* (DSW-031), a guide for ConvexOS maintenance and operations

Ordering additional paper 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

If possible, please include the order number (DSW number) or the exact title of the document you are ordering.

Acquiring technical assistance

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

- 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 can also use the `contact` utility, if you would like to report any problems you may have with ConvexOS or its associated documentation. For more information refer to the contact(1) man page or the appendix "Using `contact`" in the *ConvexOS Primer* or *Managing ConvexOS: Operations Guide.*

Part 1 Large files

Large files

1

ConvexOS allows you to create and manipulate files up to one terabyte minus 512 bytes in length. (One terabyte is 2^{40} bytes.) However, because not all utilities and applications handle files larger than two gigabytes, you must be aware of the conditions for working with such files.

This chapter uses the term *large files* to mean files larger than two gigabytes. The material in this chapter is presented in three sections. The first, "File systems and large files," explains factors of your file system environment that are important when working with large files.

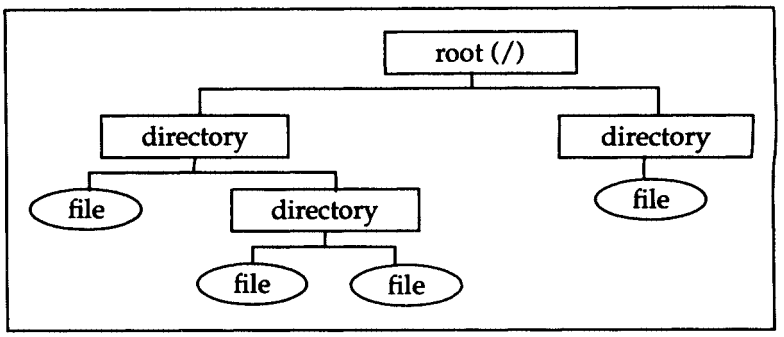
The second, "ConvexOS utilities and large files," explains conditions that apply to utilities in a large files environment. This section includes operator-oriented utilities such as `dump` and `tar`.

The third section, "Programming with large files," covers strategies for programmers who wish to write applications to create or manipulate large files. This section includes descriptions of special system calls for use with large files.

File systems and large files

File systems are structures that computers use to organize and store information. In ConvexOS, the file system is a hierarchical tree as illustrated in Figure 1.

Figure 1 File system hierarchical tree



Each file is connected or related to another starting with the root (/) directory file and continuing downward through a virtually unlimited number of files and directories.

The term file system can refer to the entire file hierarchy or to a subsection of the file tree. In this chapter the term file system refers to a subsection of the file tree, a collection of files, directories, and file management structures, that is assigned by the system manager to a certain portion of the disk system.

File system characteristics

To find out what file system you are working in:

Step 1 Enter `df .`

Figure 2 shows sample output from the `df` command.

Figure 2 Sample `df` output

```
% df .
File system kbytes used avail capacity Mounted on/dev/st3414015229809287
```

Step 2 Look at the far right column of the `df` output for the name of your file system.

If the far left column of the `df` output starts with *machinename*:, instead of `/dev/xxx`, then you are working in a file system that is NFS-mounted from remote host *machinename*. If this is the case, you cannot create or manipulate large files in this file system.

The same large file restrictions apply to local file systems mounted with the `nolf` (no large files) option. Before creating or manipulating a large file, make sure you will not place it in a file system that cannot hold it. Nonlarge-file compatible file system types and the restrictions they impose are described in the following sections.

NFS and `nolf` file systems

The Network File System (NFS) allows file systems from other machines to be remote mounted on your machine; you access normal files in NFS-mounted file systems just like you access files that really live on your home machine. However, the NFS protocol limits the size of files that can be accessed via NFS to two gigabytes or less.

The system manager may disallow large files in a given file system using the `nolf` option of `mount`. (For more information, see the chapter "Setting Up the Disk System" in *Managing ConvexOS: Configuration Guide* or the mount(8) man page.)

Large files that exist in a file system before it is mounted `nolf` are not harmed. However, only the first two gigabytes of the file are accessible after the `nolf` mount. Attempts to place new large files in a `nolf` file system will fail.

The same restrictions imposed by `nolf` apply to NFS-mounted file systems. Before creating, copying, or moving a large file, check the `mount` options on the destination file system using the following command:

```
mount | grep fsname
```

where *fsname* is the name of the destination file system. The output of this command shows the options with which file system *fsname* was mounted. If `nfs` appears as the file system type or `nolf` appears in the list of options displayed by the `mount` command, do not attempt to place a large file in file system *fsname*.

Figure 3 shows sample output from the `mount` command.

Figure 3 Sample `mount` output

```
% mount
/dev/st3on/testtype4.2(rw)
/dev/dd0b on /exprot/swap2 type 4.2 (rw)
/dev/dd0h on /usr/adm type 4.2 (rw, nolf)
newhost: /usr/spoo/news on /rmt/newshost/news type nfs (rw, bg, intr)
   .
   .
   .
```

In this example, /usr/adm is mounted `nolf` and /usr/spool/news is NFS mounted; large files cannot be created in either of these file systems.

ConvexOS utilities and large files

Some ConvexOS utilities work on large files in the same manner as they do regular files. Others place restrictions on large file operations. The following sections explain how some of the ConvexOS utilities interact with large files.

General use utilities

Not all utilities are practical for use with large files. For example, you are not likely to try to examine a large file using `more`. The ConvexOS utilities most usable with large files have been modified to ensure their usefulness. These utilities are

- `cat`
- `cp`
- `dd`
- `find`
- `ftp`
- `ls`
- `mv`
- `rcp`
- `tail`
- `test`

Utility use is subject to the file system restrictions described in the section "File Systems and Large Files" above. Using a utility to place a large file in a file system where it is not allowed may result in truncation of the file.

For example, suppose you want to move large file test1 from file system /foo to file system /bar using `mv`. If /bar cannot hold large files, `mv /foo/test1 /bar` returns a write failed message. /bar/test1 contains only the first two gigabytes of test1.

Utilities like `ls` also restrict file systems when used with large files. If a large file exists in a file system which is later mounted with the `nolf` option or via NFS, `ls` is unable to display valid size information for that file. No matter how large the file truly is, `ls -l` displays a file size of two gigabytes due to file system restrictions.

Shells and pipes

The `sh` and `csh` shells and their variants cannot manipulate large files. This means that functions such as shell redirection do not work for large files.

Pipes do work with large files. Using pipes in combination with various utilities, you can replicate many shell functions. For instance:

```
cat fname1 > fname2
```

does not work because it relies on the shell to write into fname2. However,

```
cat fname1 | dd of=fname2
```

has the same result and is valid for large files.

You can also use pipes to feed large files to some utilities that otherwise could not handle them. For instance,

```
cat fname | grep pattern
```

works, even though

```
grep pattern fname
```

does not.

File system utilities

Several utilities used for creating and maintaining file systems work with large files. Many of these are used by the system manager and require root permission. See the *Managing ConvexOS* document set for more information on those utilities.

`dump`, `restore`, and `xdump`, used by operators and users to backup and restore file systems, also understand large files. These utilities handle large files just as they would a smaller file.

Three other file system utilities have limited usefulness with large files. `tar`, `pax`, and `cpio` have a limit of 11 octal digits built into their format; they can be used to back up files up to eight gigabytes in size. If any of these utilities encounters a file larger than eight gigabytes, it issues a warning and skips that file.

Holes in large files

A hole is a region of a file that has not been written to but has been bypassed with the seek system call. The presence of the hole is recorded, but its content is not stored on disk, and it does not occupy any disk space.

A large file containing a hole may fit on a particular file system, while a file of the same size without the hole may not. cp and mv can fill in holes under some circumstances. The following examples explain how you can avoid this.

Both cp and mv have a -z option. This options preserves holes in files when they are copied or moved. Without the -z option, both cp and mv fill holes with zeros.

For example, the file myfile has a size of 10 megabytes as reported by ls, but is only taking up 32 kilobytes of actual disk space, according to du:

```
% ls -l myfile
-rw------- 1 joe 10485760 Sep 10 15:26 myfile
% du -a myfile
32      myfile
```

This discrepancy occurs because myfile contains a hole. If myfile is copied or moved to another directory, the holes are filled with zeros:

```
% cp myfile bigfile
% ls -l bigfile
-rw------- 1 joe 10485760 Sep 10 15:30 bigfile
% du -a bigfile
10304   bigfile
```

If myfile is copied with the -z option, the hole is preserved and the copy takes up only 32 kilobytes of actual disk space:

```
% cp -z myfile bigfile
% ls -l bigfile
-rw------- 1 joe 10485760 Sep 10 15:34 bigfile
% du -a bigfile
32      bigfile
```

Programming with large files

Three things allow a program to read or write beyond the two gigabyte boundary in a file.

- Setting the `O_LARGEFILE` flag when a file is opened. To do so using the `open()` system call or the `fopen()` system call,
  ```
  fd=open("largefilename", O_RDWR | O_LARGEFILE);
  ```
  or
  ```
  FD=fopen("largefilename", "rwl");
  ```
- Setting the same flag with the `fcntl()` system call.
  ```
  fd=open("largefilename", O_RDWR);
  flag=fcntl(fd, F_GETFL, 0);
  fcntl(fd, F_SETFL, flag!FLARGEFILE);
  ```
- Setting the large files flag using the `lseek64()` system call.
  ```
  fd=open("largefilename", O_RDWR);
  lseek64(fd, 12345678900, SEEK_SET);
  ```

Note that you must use `fcntl` properly when working with large files. Shortcut methods may lose the large file flag. For example, setting Async I/O on a large file with a call structure such as the following

```
fcntl(fd, F_SETFL, FASIO)
```

loses the O_LARGEFILE flag. However,

```
flag=fcntl(fd, E_GETFL, 0);
flag|=FASIO
fcntl(fd, F_SETFL, flag);
```

keeps the setting of O_LARGEFILE.

The following sections explain how programmers can manipulate large files. Later sections explain some restrictions when operating on large files.

FORTRAN support for large files

Transparent support for large files is available with CONVEX FORTRAN V7.0. Large formatted and unformatted files may be read and written. The FORTRAN I/O interface extends transparently to large files.

There is one limit on the use of large files, namely that direct access files are limited to 2**31-1 RECORDS. The actual size of such a file is then limited by the size of individual records. This restriction is due to the use of the `INTEGER*4` data type for the REC= specifier in READ and WRITE statements for direct access files, and the NEXTREC= specifier in INQUIRE statements.

FORTRAN programs must be re-linked with V7.0 in order to read and write large files. All FORTRAN programs linked with the V7.0 library have large file access.

System calls for use with large files

Many system calls take a file offset as an argument. 64-bit versions of many of these system calls are available for use with large files. Two stdio routines, `fseek64()` and `ftell64()`, also are available for use with large files.

Each of these functions has a 32-bit counterpart whose name is the same except for the ending 64 (`lseek64()`, `lseek()`; `fseek64()`, `fseek()`; etc.). Except as noted, the 64-bit function has the same functionality as its 32-bit counterpart, but takes a 64-bit offset.

The new system calls live in libc.a and its variants. They are not available in libc_old.a.

- `lseek64()` — seek on a file descriptor using 64 bit offsets. A successful call to `lseek64()` sets the `O_LARGEFILE` bit.
- `fseek64()` — seek on a stream using 64 bit offsets. Implemented with `lseek64()`.
- `stat64()` — perform a `stat()` on a large file, returning a structure with 64 bit offsets. Uses the `stat64_t` structure; declares the `st_size` field as an `off64_t`.
- `fstat64()` — perform an `fstat()` on a file descriptor, returning a structure with 64 bit offsets
- `lstat64()` — `lstat()` with 64 bit offsets
- `ftell64()` — stream position as a 64 bit offset
- `truncate64()` — `truncate()` using 64 bit offsets
- `ftruncate64()` — `ftruncate()` using 64 bit offsets

The existing versions of `read()` and `write()` work with large files, but must have the `O_LARGEFILE` bit set in order to function past two gigabytes into a file.

Limited system calls

The `mmap()`, `getrlimit()`, and `setrlimit()` system calls require 32-bit file offsets and do not understand large files. They do not adversely affect large files, but their usefulness with respect to large files is limited.

`mmap()` only maps the first two gigabytes of a file. `setrlimit()` can limit file size with byte granularity up to two gigabytes. It may, as always, set the file size limit to `RLIM_INFINITY` to allow access to any size file. `setrlimit()` cannot set, and `getrlimit()` cannot understand, limits above two gigabytes.

The `flock` structure used by `fcntl()` also contains file offsets and does not understand large files; `flock` is used with NFS, which cannot handle large files. Because `flock` uses a 32-bit offset, file locking is limited to the first two gigabytes of a file.

Include file values

The following include files contain definitions useful in programming with large files:

- <sys/types.h> contains the typedef `off64_t`, which defines the possible offsets, and thus the maximum size, of a file. It is defined as a long long.
- <sys/stat.h> contains the structure, `struct cvxstat`, used by `stat64()`. It is defined as type `stat64_t`. A structure `stat`, defined as type `stat_t`, is defined for regular file offsets.
- The `open()` `flag O_LARGEFILE` and `fcntl()` flag `FLARGEFILE` are defined in <sys/fcntl.h>. Both `open()` and `fcntl()` previously required this file.

File system considerations

Although the stated limit for large files is one terabyte minus 512 bytes, some combinations of block size and number of free blocks can make it impossible to write a file that large. On file systems with block sizes less than 32K, you must use triple indirect block pointers to write a large file of maximum size. In such a file system (block size < 32K), the availability of block pointers determines the maximum file size you may write.

Existing programs and large files

Compiling C code with the `-pcc` option forces use of backward-compatible system libraries. Thus, compiling with the `-pcc` option disables large file support for the program being compiled. Versions 4.2 and later of the CONVEX C compiler contain support for large files.

The behavior of existing programs with large files is determined by the behavior of the system calls those programs use. Recompiling an old executable in a large file environment has no effect because it still uses the same system calls.

The danger in using an existing program with a large file is to the file, not the program. A program that does not understand large files sees only the first two gigabytes of a large file and could truncate the file at the two-gigabyte boundary or write data in the wrong place. The following paragraphs explain how several system calls behave if they encounter a large file.

`lseek()` fails on a large file if the file offset is already greater than two gigabytes or if a successful seek would cause the offset to be greater than two gigabytes. It ignores the `O_LARGEFILE` bit.

`stat()` All variants of `stat()` return the largest 32-bit integer (i.e. 0x7fffffff) in the `st_size` field of the stat structure if the file size being reported is greater than 0x7fffffff. `stat()` gives no indication that the returned value is not the actual size of the file.

`truncate()` `truncate()` and `ftruncate()` both will truncate a large file at the two-gigabyte boundary.

Remember that these limitations also apply if a program that understands large files calls another program that does not.

POSIX issues

ConvexOS large file support does not comply with the POSIX 1003.1 standard. Therefore, strictly conforming POSIX applications do not have access to large files beyond the two gigabyte boundary. Conforming applications receive an EFBIG error if they attempt to write more than two gigabytes.

Programs using the new system calls provided for large files support are using extensions to the POSIX standard. The new system calls are only available in the extended (default) mode of the CONVEX C compiler.

Part 2 CONVEX Fair Share Scheduler

Share scheduling

2

Share is an optional product that allows machine resources to be divided between groups of users. This chapter explains

- How Share scheduling differs from standard ConvexOS scheduling
- How to determine if Share is running on your machine
- How to determine what proportion of resources are available to you (how many *shares* you have)
- How to determine what your *usage* is
- How to determine the current *charge* for resources
- Strategies that will help you get the most out of your share

User-level man pages available online for Share are

- charges(1)
- dis(1)
- idle(1)
- pl(1)
- pwintf(1)
- rates(1)
- sl(1)

What is Share scheduling?

A *scheduler* is the part of the operating system that keeps track of processes that are waiting to be executed and decides which process executes next. (A *process* is a program that has been started but has not yet completed.)

On a time-sharing system, there are usually many process running at the same time. The *central processing unit* (CPU) is a part of a computer that executes processes. Each CPU can execute only one process at a time, so processes must take turns using it.

Each process waiting for a CPU is given a *priority value* and placed in a scheduling queue based on that value. Processes with low priority values have a high execution priority and are placed at the front of the queue. Processes with high priority values have lower execution priorities and are placed farther back in the scheduling queue. Processes that are currently in the scheduling queue are *active processes*.

When a process reaches the front of the scheduling queue, it is executed for a brief period of time, called a *time slice*. If a process uses its entire time slice without completing, its priority is recalculated and it is placed back on the scheduling queue to wait for another time slice. The priority of a process may change frequently throughout its lifetime.

There are two types of schedulers available on CONVEX systems:

- The standard ConvexOS scheduler
- The CONVEX Share Scheduler, an optional product

The major difference between the two lies in the factors each takes into account when assigning priorities to active processes.

Standard ConvexOS scheduling

The standard ConvexOS scheduler takes the following factors into account when assigning priorities:

- The recent activity of the process. For example, when a process is executing, its priority falls. While a process is *blocked* (waiting for some other event such as a disk read or user input), its priority rises.
- The nice value of the process. A nice value is a user-specified parameter that allows you to voluntarily lower the priority of a process. (For more information on nice values, see "The nice command" section on page 24.)

- The amount of CPU time the process has already consumed. Processes that have already consumed a great deal of CPU time will have lower priorities. The system load may also affect the degree to which recent CPU utilization affects priority.

Using only these factors to calculate priority results in short-term, per-process scheduling. The standard ConvexOS scheduler considers only the recent activity of single processes. If there are 100 active processes, each process will receive approximately 1/100th of the machine.

As a result, a single user can get a large share of the machine by simply running many processes. Also, a user who has used the machine heavily for several hours can get an equal portion (or a larger portion) of resources as a user who has not used the machine for some time.

CONVEX Share Scheduler

Unlike the standard ConvexOS scheduler, Share is long-term, per-user scheduler. Users or groups of users are assigned a predetermined *share* of the machine by the system manager. Share tracks the amount of resources each active user or group is using, and prevents users from receiving more than their *intended share*.

Instead of the factors the standard ConvexOS scheduler uses to calculate priority, Share uses the following:

- The user's accumulated history of recent CPU usage, for all the user's active processes
- The number of active processes a user has
- The user's intended share

By limiting the rate at which a user can consume CPU resources, Share prevents greedy users from impacting other users. If a user has a high recent usage, is assigned a small share of the machine, or has many active processes, each of that user's active processes will receive a smaller portion of CPU resources.

All users are organized into scheduling *groups*. Shares are assigned both to groups and to users in the groups. The number of shares you are assigned is significant in relation to

- The number of shares assigned to other users in your group
- The number of shares assigned to other scheduling groups

Important terms

This section contains terms you should understand when using Share.

Shares

Shares are units assigned to you by your system manager. The number of shares you have is significant only in relation to the number of shares assigned to other users and groups.

Usage

Usage is an accumulated history of your recent CPU consumption. Your usage increases when you are logged in and running processes. It decreases when you are inactive.

Usage half-life

If your usage accumulated infinitely, eventually it would be high enough to ensure that all your processes are scheduled at a very low priority. The length of time for which your previous usage affects your priority is the *usage half-life.*

When you are not logged in, your usage decreases to half its initial value over the half-life period. When you are logged in your usage also decreases, but since you are accumulating usage (by running processes) at a much faster rate, the decrease isn't usually noticeable.

Intended share

Your *intended share* is the percentage of CPU resources you should receive, based only on the number of shares you have and the number of shares other active users have. When other users log in or out, your intended share will change.

Effective share

Your *effective share* is the percentage of CPU resources you should receive in the near future, based on your intended share and your usage.

Monitoring your share

Several utilities can be used to monitor your share and usage. This section describes how to run these utilities and evaluate their output. For information on using your share effectively, see the section titled "Tips for using Share" section on page 24.

Determining current charge and half-life

The `charges` program displays information about the way Share is configured on your system. Figure 4 shows sample output from the `charges` command.

Figure 4 Sample `charges` output

```
% charges
Scheduling flags = SHARE,ADJGROUPS,LIMSHARE, charging percentage = 100%,
usage decay rate = 0.99692410 (half-life 900.0 seconds),
Max group nesting = 4.

Charge: syscall 0%      bio 0%        tio 0%       tick 100%       click 0%
```

The fields highlighted in gray are described below. See the *CONVEX Share Scheduler System Manager's Guide* for information about the other fields in this output.

Scheduling flags	The SHARE flag indicates that Share is running on the machine. When Share is not running, NOSHARE appears in this field.
Charging percentage	Indicates the current cost for resources. Your system manager may chose to reduce charges at off-peak periods such as evenings or weekends. During those times, this number will be less than 100%.
Half-life	A time period over which your usage will gradually be decreased by half. In this example, the usage half-life is 900 seconds, or fifteen minutes.
Charge	These fields display the amount being charged for all resources. The CONVEX Share Scheduler only supports charging for CPU use; the tick field always contains 100% and the other fields (syscall, bio, tio, and click) will always contain 0%.

Displaying your share information

You can view your Share information using the `pl` command. Figure 5 shows sample output from the `pl` command.

Figure 5 Sample `pl` output

```
% pl
Name: joe
Uid/Gid: 1313/51 Euid/Egid: 1313/51
Scheduling group: primaryg
Flags: Active
Memory used: 156kb
Processes: 2
Shares: 100
Intended share: 0.552%
Effective share: 0.00171%%
Normalised usage: 9.028540e+13
Usage: 1.133534e+06
Charge: 5.924420e+09
Last used: Sat Nov 9 08:00
```

Each field is described below.

Name	User's login name.
Uid/Gid	User ID and Group ID.
Euid/Egid	The user's effective user ID and group ID. (See the sl(1) man page for more information on how to change your effective UID and GID.
Scheduling	The scheduling group to which the user group belongs.
Flags	Indicates whether or not the user is currently logged in. This can be Active User is currently logged in. No Flags User is not currently logged in.
Memory used	The amount of memory the user's processes are using.
Processes	The number of processes the user is running
Shares	The number of shares assigned to the user. This number is significant only in relation to the number of shares assigned to other users and groups.

Intended share	The percentage of CPU resources that the user is entitled to, based on the number of shares assigned to the user and shares assigned to all other users that are currently logged in.
Effective share	The percentage of CPU resources the user is entitled to based on intended share and usage.
Normalised usage	This value is used internally by Share to calculate priority.
Usage	The amount of CPU resources the user has consumed. This value increases while the user is logged in and running processes. During periods of inactivity, usage decreases. The usage half-life determines the rate of decrease.
Charge	A running total of all the CPU resources consumed by the user. Unlike usage, this value does not decrease.
Last used	The last time the user consumed CPU resources, excluding the current login session.

For more information about `pl`, see the pl(1) man page.

Displaying current CPU consumption

The `rates` program displays the current CPU distribution among all active users. Figure 6 shows an example of `rates` output.

Figure 6 Sample `rates` output

```
% rates -u -C
User            No.    %Cpu                                  Mon Nov 18 09:16:15 1991
System            1     0.0 !                                                        I|
jones             1     0.5 I                                                        |
roberts           1     2.4 I                                                        |
henny             1     2.0 I                                                        |
wayne             1     0.6 I                                                        |
mary              1     2.5 I                                                        |
mckay             1     1.6 I                                                        |
kelly             1     1.6 I                                                        |
watson            1     1.8 I                                                        |
martin            1     7.7 I###                                                     |
jane              1     4.9 I#                                                       |
carl              1    10.2 I####                                                    |
evan              1     1.4 I                                                        |
susan             1     2.0 I                                                        |
shelley           1     7.0 I##                                                      |
farmer            1     2.0 I                                                        |
stone             1    12.3 I#####                                                   |
daemon            1    26.2 ############!       I                                    |
Other             1     0.0 I                                                        |
network           1    13.1 #######             I                                   |
Total =>         20
```

The `-C` option displays CPU consumption; the `-u` option displays information for all users, not just for scheduling groups.

The output shown in is described below.

User	The user represented by this line
No.	The number of users represented by this line. If this were a scheduling group display, it would indicate the number of users active in a scheduling group.
%Cpu	The percentage of resources each user is receiving, displayed numerically.
Bar Graph	A graphical display of the current distribution of resources among users.
#	Indicates the rate of consumption for each user. Very low rates (less than 4%) may not be displayed graphically.
I	Indicates the intended share.
!	Indicates the effective share.

The vertical bar (|) at the end of the graph indicates the 100% point; if placed end to end, all the pound signs (#) on the graph would extend to the vertical bar.

For more information about `rates`, see the rates(1) man page.

Tips for using Share

This section describes several ways to get the most out of your share.

Half-life

You should spread your work out evenly over the half-life period. CPU-intensive processes will increase your usage rapidly, but usage decreases quickly when you are not logged in. By spreading your work out evenly, your system response time will remain fairly constant.

Charges

Your system manager may choose to decrease the charge for CPU use during off-peak periods such as evenings or weekends. When charges are lowered, usage does not accrue quickly, and you can consume more CPU than when charges are high.

The `charges` program will display the current charging percentage; ask your system manager for information about specific time periods when charges are reduced.

The `nice` command

The `nice` command can be used to voluntarily lower the priority of a process. The charge for processes with low priorities is reduced; therefore less usage is accrued.

By default, processes have a nice value of 0. You may specify a nice value from +1 to +64. The higher the nice value, the lower the priority of the process. A process with a nice value between +32 and +64 will have no charge and will not accrue usage.

`nice` has a shell-dependent syntax. If you use `csh`, the syntax is

```
nice [+n] command
```

where *n* is a nice value of 1 through 64, and *command* is the command you wish to execute. If you do not use the +*n* switch, the priority is set to +4 by default.

For example, the following command starts a `make` process with a nice value of +64:

```
nice +64 make
```

If you are using `sh` or `ksh`, the syntax is:

```
nice [-n] command
```

where *n* is a nice value of 1 through 64 and *command* is the command you wish to execute.

For example, the following command starts a `make` process with a nice value of 64:

```
nice -64 make
```

The `renice` program can be used to lower the priority of a process that is already running. To use `renice` on a process, you must determine its process ID. To do this, enter

```
ps ux
```

Find the name of the command you used to start the process in the far right column. The second column from the left contains the process ID.

The `renice` command has the following syntax:

```
/etc/renice n processID
```

For example, the following command lowers the priority of a `make` process with process ID 482:

```
/etc/renice 64 482
```

For more information, refer to the nice(1), renice(8), and csh(1) man pages.

Share and CXwindows

If you use CONVEX CXwindows on a workstation and access a CONVEX machine remotely through `xterm` windows, you should run the `xterm` processes on your workstation and use the `rlogin` program to access the remote machine.

You do not accrue usage for processes running on your workstation, but you do accrue usage for processes running on the CONVEX machine. Running `xterm` on your workstation instead of on the CONVEX machine will help keep your usage low.

Share and CXbatch

Your system manager may choose to assign separate share allocations for batch queues. If this is the case, CPU consumption of jobs you run using batch queues will not affect your own usage.

Part 3 POSIX Compliance

POSIX and ConvexOS

3

What is POSIX?

In the early 1980's, /usr/group began a standards movement. Along the way, it was joined by other groups wanting to produce a UNIX-related standard. These efforts resulted in POSIX. POSIX actually refers to a group of proposed standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) was the first of the POSIX standards to be adopted. It was ratified on August 22, 1988 and represents a standard system call interface and environment based on the UNIX operating system. It is intended to support application portability at source-code level. ISO/IEC 9945-1, 1003.1-1990 is a revised version of 1003.1-1988 and was ratified on September 28, 1990.

POSIX.1 describes external characteristics and facilities that are important to application developers, rather than internal construction techniques used to achieve these capabilities. Special emphasis is placed on those functions and facilities needed in a wide variety of commercial applications. POSIX.1's objective is for a strictly conforming application to compile on a new host without any code modifications.

ConvexOS is an implementation of the Berkeley UNIX operating system containing POSIX.1 functionality with extensions for supercomputer environments. ConvexOS retains backward compatibility for existing binaries and sources.

POSIX is based largely on UNIX Seventh Edition, UNIX System III, UNIX System V, BSD V4.2, and BSD V4.3 documentation.

ConvexOS was first compliant at V8.0. CONVEX will keep its operating system current with all areas of POSIX as each working committee completes and ratifies its group of standards. Interested customers can monitor the progress of POSIX through participation in IEEE and by reading the comp.std.unix network news group.

How CONVEX interprets POSIX.1 is contained in the CONVEX POSIX Conformance document. This manual describes all the implementation-defined features required by the IEEE.

Why use POSIX?

POSIX is more of an environment than anything else. As mentioned earlier, there are many UNIX-based implementations in use today. POSIX will standardize many of the common areas for operating systems and interfaces to them, so application portability should be easier. That is why the IEEE has dealt with external characteristics and facilities and not internal constructions. POSIX standards should be seen as a way to bring many diverging systems together at last.

Working committees

Table 1 lists each POSIX working committee and its area of standardization.

Table 1 POSIX working committees

Committee	Subject
1003.0	POSIX Guide
1003.1	Operating System Interface
1003.2	Shell and Tools Interface
1003.3	Verification and Testing Methods
1003.4	Real Time Extensions
1003.5	Ada Bindings
1003.6	Security Extensions
1003.7	System Administration
1003.8	Distribution Services
1003.9	FORTRAN Bindings
1003.10	Supercomputing Application Environment
1003.11	Transaction Processing
1003.12	Distributed Services
1003.15	Batch Queuing
1003.17	Directory Namespace
1201.x	Interfaces for User Portability

POSIX and ConvexOS specifics

4

Introduction

In POSIX.1, the IEEE requires that a document be created "for an implementation claiming conformance." The CONVEX POSIX Conformance document contains all implementation-defined features for POSIX.1 but no information on extended facilities (that is, CONVEX extensions and information on backward compatibility). This CONVEX-specific information is documented in this chapter. Subheadings are identical to chapter names in POSIX.1 (where CONVEX has additional information to discuss) with backward-compatible and CONVEX-extension sections. Each section contains a function-by-function breakdown.

Process primitives

The following sections contain information on backward compatibility and CONVEX extensions as they regard process primitives.

Backward compatibility

`execve()`: In former versions of the operating system, when a program was `setuid()` to nonsuperuser but was executed when the real UID is "root," the program had the powers of a superuser as well.

`kill()`: Previous versions of the operating system based the permission test solely on a match of the effective UIDs of the sender and receiver.

`sigaction()`: The `sigaction` function replaces the `sigvec` function.

The struct `sigaction` replaces the struct `sigvec`.

Signal-handling functions formerly returned type `int`; now they return type void. This may be the source of many compile-time warnings.

By default, previous releases restarted system calls on signals; the default now is not to restart system calls. This is accomplished by initializing the sa_flags bit _SA_INTERRUPT to be set when a program is exec'ed; by clearing the bit, the old behavior may be obtained.

`sigprocmask()`: The `sigprocmask()` function replaces `sigblock()` and `sigsetmask()`.

`wait()`: Previous releases of the operating system documented use of a pointer to union `wait`, where the current release advocates use of a pointer to integer. To convert existing code, simply change calls like

```
union wait wait_union;

wait(&wait_union);
```

to one of the following:

```
wait(&wait_union.w_status);

wait((int *)&wait_union);
```

Existing code will then continue to work as before.

Previous versions of the operating system by default restarted the `wait` family functions when a signal was received while

awaiting termination of a child. Refer to sigaction(2) for details of controlling this behavior.

CONVEX extensions

The `exect` version is used when the executed file is to be manipulated with pattach(2). The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state.

`exec()`:

```
exect(name, argv, envp);

char *name, *argv[], *envp[];
```

`sigaction()`: Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special stack.

As an extension, if the _SA_ONSTACK bit is set in `sa_flags`, the system will deliver the signal to the process on a signal stack specified with sigstack(2).

An alternate mode of signals can be used by setting the _SA_PARALLEL bit in `sa_flags`. When this bit is set, signals that are masked by the delivery of a signal will not be reflected in the masks returned by sigblock(2) and sigsetmask(2) system calls. It also changes the action taken upon return of the signal handler to unblock signals blocked when the handler was invoked, rather than reset the mask to what it was before signal delivery.

The handler routine can be declared:

```
void handler(sig, code, scp);

int sig, code;

struct sigcontext *scp;
```

`sig`	Signal number into which hardware faults and traps are mapped.
`code`	Parameter that further defines the type of hardware exception that occurred.
`scp`	Pointer to the sigcontext structure (defined in <signal.h>) and used to restore the context from before the signal.

defines the mapping of hardware traps to signals and codes. All of these symbols are defined in Table 2.

Table 2 Hardware traps mapped to signals and codes

Hardware condition	Signal	Code
Arithmetic traps		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Reserved Operand trap	SIGFPE	FPE_RESOP_TRAP
Square root negative argument	SIGFPE	FPE_SQRT_TRAP
Exp overflow	SIGFPE	FPE_EXP_TRAP
Ln argument less than or equal zero	SIGFPE	FPE_LN_TRAP
Argument too large	SIGFPE	FPE_SIN_TRAP
Argument too large	SIGFPE	FPE_COS_TRAP
Segmentation Violations		
Read access violation	SIGSEGV	SEG_READ_TRAP
Write access violation	SIGSEGV	SEG_WRITE_TRAP
Execute access violation	SIGSEGV	SEG_EXEC_TRAP
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP
I/O access violation	SIGSEGV	SEG_IOACC_TRAP
Invalid LEVT PTE fault	SIGSEGV	SEG_INVPTET_TRAP
Ring Violations		
Inward address reference	SIGBUS	BUS_INWADDR_TRAP
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP
Inward ring return	SIGBUS	BUS_INWRTN_TRAP

Table 2 Hardware traps mapped to signals and codes **(continued)**

Hardware condition	Signal	Code
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP
Invalid communication address	SIGBUS	BUS_INVCRA_TRAP
Invalid trap instruction	SIGBUS	BUS_INVTRPINS_TRAP
Illegal instruction		
Error exit instruction	SIGILL	ILL_ERRXIT_TRAP Privileged
instruction	SIGILL	ILL_PRIVIN_TRAP Undefined op
code	SIGILL	ILL_UNDFOP_TRAP
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Breakpoint	SIGTRAP	TRP_BKPT_TRAP
Instruction trace	SIGTRAP	TRP_TRACE_TRAP
Fork system call	SIGTRAP	TRP_FORK_TRAP
Exec system call	SIGTRAP	TRP_EXEC_TRAP
Thread creation	SIGTRAP	TRP_TDCREATE_TRAP
Thread join executed	SIGTRAP	TRP_TDJOIN_TRAP
Thread wfork executed	SIGTRAP	TRP_TDWFORK_TRAP
Thread idle executed	SIGTRAP	TRP_TDIDLE_TRAP
Last thread deadlock	SIGTRAP	TRP_TDLAST_TRAP
Mixed wfork/join	SIGTRAP	TRP_TDMIXED_TRAP
Process breakpoint trap	SIGTRAP	TRP_PBKPT_TRAP
Trap instruction code low	SIGTRAP	TRP_TRAPINSTLO
Trap instruction code high	SIGTRAP	TRP_TRAPINSTHI

`wait3()` provides an alternate interface for programs that must not block when collecting the status of child processes. The status and options parameters are defined as above. If `rusage` is nonzero, a summary of resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

`wait()`:

```
#include <sys/time.h>

#include <sys/resource.h>

int pid; pid = wait3(status, options,
rusage);

int *status;

int options;

struct rusage *rusage;

int pid;

pid = cvxwait(status, options, rusage);

int *status;

int options;

struct cvxrusage *rusage;
```

`cvxwait()` is identical to `wait3()` but returns a struct `cvxrusage` instead of a struct `rusage`. The primary difference between these is that struct `cvxrusage` contains information about the level of parallelism of the terminated process.

Process environment

The following sections contain information on backward compatibility and CONVEX extensions in the process environment.

Backward compatibility

`getgroups()`: In previous versions of the operating system, the array filled in by `getgroups()` was of type `int`; it is now of type `gid_t`.

`getpgrp()`: Previous versions of the operating system specified argument pid to `getpgrp()`. A value of zero meant the current process. This mechanism is supported only in backward-compatible mode.

`setpgid()`: This function was previously known as `setpgrp()`. The functionality of `setpgrp()` was essentially that of `setpgid()` without the restrictions on session membership.

`setuid()`: `seteuid()`, `setruid()`, `setegid()`, and `setrgid()` set the effective and real user and group IDs of the current process.

`times()`: Previous versions of `times()` filled the struct `tms` members with units measured in 60ths of a second. CLK_TCKs are not necessarily the same.

Previous versions of `times()` always returned zero or -1.

CONVEX extensions

Usage is otherwise as for `getgroups()`.

`getgroups()`:

```
#include <sys/types.h>
#include <unistd.h>
int ngrps, gsize;
ngrps = int_getgroups(gsize, intgidset);
int *intgidset;
```

Files and directories

The following sections contain information on backward compatibility and CONVEX extensions as they regard files and directories.

Backward compatibility

`access()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

`chdir()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

`chown()`: The owner and group parameters were formerly declared to be type `int`, although the high word was unusable.

`creat()`: Previous versions of the operating system did not allow the EINTR errno to occur by default (refer to `sigvec()`).

`directory()`: Previous versions of the operating system wanted <sys/dir.h>, not <dirent.h>, to be included.

`link()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

`open()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

`rename()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

`rmdir()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

`stat()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

The file status information was previously expressed as follows:

#define S_IFMT0170000/* type of file */

#define S_IFIFO0010000/* fifo special */

#define S_IFCHR0020000/* character special */

#define S_IFDIR0040000/* directory */

#define S_IFBLK0060000/* block special */

#define S_IFREG0100000/* regular */

#define S_IFLNK0120000/* symbolic link */

#define S_IFSOCK0140000/* socket */

#define S_ISUID0004000/* set user id on*/

/*execution */

#define S_ISGID0002000/* set group id on */

/*execution */

#define S_ISVTX0001000/* refer to sticky(8) */

#define S_IREAD0000400/* read permission,*/

/* owner */

#define S_IWRITE0000200/* write permission,*/

/* owner */

#define S_IEXEC0000100/* execute/search*/

/*permission, owner */

The mode bits 0000070 and 0000007 encode group and others permissions (refer to `chmod()`).

`unlink()`: Previous versions of the operating system allowed an empty path name as a synonym for the current directory.

CONVEX extensions

`chmod()`: The _S_ISVTX mode bit is added (refer to sticky(8)).

`chown()`: A value of _SAME_UID passed for owner or _SAME_GID for group results in the current owner or group remaining unchanged.

If the final component of path is a symbolic link, the ownership and group of the symbolic link is changed, not the ownership and group of the file or directory to which it points.

`directory()`:

```
long telldir(dirp);
DIR *dirp;
seekdir(dirp, loc);
DIR *dirp;
long loc;
```

`seekdir` sets the position of the next `readdir` operation on the directory stream. The new position reverts to the one associated with the directory stream when the `telldir` operation was performed. Values returned by `telldir` are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the `telldir` value may be invalidated due to undetected directory compaction. It is safe

to use a previous `telldir` value immediately after a call to `opendir` and before any calls to `readdir`.

stat():

```
lstat(path, buf);

char *path;

struct stat *buf;
```

`lstat` is like `stat` except in the case where the named file is a symbolic link, in which case lstat returns information about the link, while stat returns information about the file the link references.

CONVEX provides the following file test macros:

_S_IFLNK(st.st_mode)True if a symbolic link

_S_ISSOCK(st.st_mode)True if a socket

The file mode bits will contain _S_ISVTX if the sticky bit is set (refer to sticky(8)).

Input and output primitives

The following sections contain information on backward compatibility and CONVEX extensions as they regard input and output primitives.

Backward compatibility

`close()`: The EINTR return value was formerly difficult to observe (refer to `sigvec()`); it is now the default, and the use of non-POSIX extensions is required to avoid it (refer to `sigaction()`).

`read()`: The O_NONBLOCK mode of operation was formerly known as O_NDELAY.

The current implementation of O_NONBLOCK is slightly different than the previous one in that O_NONBLOCK will affect no processes other than the caller, whereas O_NDELAY would affect all processes with file descriptors open for that device.

The error return EWOULDBLOCK occurs in backward-compatible mode in those situations that now return EAGAIN.

`write()`: Where previous versions of the operating system returned EWOULDBLOCK for errno, EAGAIN is now returned.

The default signal behavior is now to interrupt and not restart a `write` operation.

CONVEX extensions

`fcntl()`: CONVEX adds the following cmd values as extensions:

_F_GETOWN Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values.

_F_SETOWN Set the process or process group to receive _SIGIO and _SIGURG signals; process groups are specified by supplying arg as negative; otherwise, arg is interpreted as a process ID.

_F_SETBLKSIZE Set the physical record size for block tape operations to the value of arg. This value defaults to 65536 when the file is opened.

CONVEX adds the following file status flags as extensions:

_FASIO

Use daemon processes to accomplish asynchronous I/O. Subsequent operations on this fd cause the user process to proceed in parallel with the I/O transfers. Several I/O transfers may be proceeding in parallel to or from the same file (fd), each under the control of a separate daemon process, limited only by a system-wide configuration maximum (`ps -axl` displays daemon processes as asiodaemon). Synchronization may be accomplished with the `asiostat`, `select`, or `fstat` system calls, or by the use of _FASYNC.

It is assumed that all requested I/O will be performed; the read or write system calls will return as if all data had been transferred, even though the transfer may be impossible. To determine results, the asiostat system call will return the sum of the number of bytes transferred on all asynchronous I/O requests on a given file descriptor since the last asiostat call. The _FASIO property may be inherited by forked children, and it is illegal to set _FASIO on socket descriptors.

Asynchronous I/O seems to help tape performance, but due to the existence of the buffer cache, read ahead, and write behind, sequential file transfer is already quite asynchronous, and no performance improvement may be noted. The `brk()`, `_exit()`, and `fork()` system calls will wait for all asynchronous I/O invoked by the calling process to be completed before being processed. The `asiostat()`, `close()`, `fcntl()`, `flock()`, `fstat()`, `fsync()`, `ftruncate()`, and `ioctl()` system calls will wait for all asynchronous I/O on the specified file descriptor (fd) to be completed before being processed.

_FASYNC — If the _FASIO bit is set, a _SIGIO signal will be sent to a process when all its outstanding asynchronous I/O (using daemon processes) has been completed. If the _FASIO bit is not set, _FASYNC enables the _SIGIO signal to be sent to the process group when I/O is possible, for example, upon availability of data to be read. (This latter facility is available only on tty operations.)

_FNCACHE — This bit requests the kernel to bypass the incore buffer cache and perform I/O transfers directly to/from user address space. As a side effect, file system read ahead and write behind are disabled. For files that are accessed sequentially, the net result is usually lost performance. _FNCACHE might be helpful in situations where the file in question is accessed randomly, but you should test results with and without this option set before using it. The operating system cannot bypass the buffer cache if transfers are not aligned on the file's blocksize boundaries because disk controllers cannot start transferring in the middle of sectors. The `fstat()` system call returns the proper blocksize as st_blksize. Transfers should start (by seeking if necessary) at integral multiples of st_blksize.

`read()`: `read()` may operate synchronously (default) or asynchronously, depending on the _FASIO flag set with the fcntl() system call. Synchronous reads suspend the caller until the system has processed the read request and return the number of bytes read and placed in the buffer or 0 if end-of-file has been reached. The file position pointer is incremented by the number of bytes read. Asynchronous reads return before the request has been fully processed, and unless there are errors in the arguments passed, always return the number of bytes requested, whether or not it is possible to complete the request. The file position pointer is incremented by the number of bytes requested. To determine what happened during asynchronous transfers, refer to `asiostat()`.

`write()`: `write()` may operate synchronously (default) or asynchronously, depending on the _FASIO flag set with the `fcntl()` system call. Synchronous writes suspend the caller until the system has processed the write request, and return the number of bytes written. The file position pointer is increased by the number of bytes written. Asynchronous writes return before

the request has been fully processed, and unless there are errors in the arguments passed, always return the number of bytes requested, whether or not it is possible to complete the request. The file position pointer is increased by the number of bytes requested. To determine what happened during asynchronous transfers, refer to `asiostat()`.

Device- and class-specific functions

The following sections contain information on device- and class-specific functions with regard to terminal attributes, special characters, and TERMIOS flags.

Terminal attributes

Version 7.1 and earlier of the operating system used `ioctl()` to get and set terminal attributes. With POSIX.1, terminal attributes are stored in a termios structure and can be set and retrieved with `tcsetattr()` and `tcgetattr()`.

Window size

These functions also set and get window-size parameters. If the SETWINSIZE bit is set in the `c_lflags` field of the `termios` structure and passed to tcsetattr(), the window-size field in the termios structure becomes the new window size (and a SIGWINCH signal is sent if appropriate).

Raw mode

To go to raw mode, pre-POSIX applications used the following:

```
ioctl(fd, TIOCGETP, &sgttyb);
sgttyb.sg_flags |= RAW;
ioctl(fd, TIOCSETP, &sgttyb);
```

Applications under ConvexOS now use

```
tcgetattr(fd, &termios);
termios.c_iflag
&=~(IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK|ISTRIP|
INLCR|IGNCR|ICRNL|IXON|IXANY|IMAXBEL);
termios.c_oflag &= ~OPOST;
termios.c_cflag &= ~(PARENB|CSIZE);
termios.c_cflag |= CS8;
termios.c_lflag &= ~(ECHOCTL|ISIG|ICANON);
termios.c_cc[VMIN] = 1;
termios.c_cc[VTIME] = 0;
tcsetattr(fd, TCSANOW, &termios);
```

Special characters

Individual special characters can be disabled by setting them to the value of the DISABLE special character. Previous versions disabled special characters by setting them to the value '\377'. Previous versions of the operating system used the following:

```
ioctl(fd, TIOCGETP, &sgttyb);
sgttyb.sg_intrc = '\377';
ioctl(fd, TIOCSETP, &sgttyb);
```

ConvexOS uses

```
tcgetattr(fd, &termios);
termios.c_cc[VINTR] = termios.c_cc[VDISABLE];
tcsetattr(fd, TCSANOW, &termios);
```

termios flags

c_iflag input flags

IGNBRK — Ignore break. If a break condition is detected on the line, it is completely ignored and invisible to the application.

BRKINT — Map a break to an interrupt. If IGNBRK is not set and a break condition is detected on the line, a SIGINT is sent to the foreground process group of the tty.

IGNPAR — Ignore parity errors. If a parity or framing error is detected, it is completely ignored and invisible to the application.

PARMRK — Mark parity and framing errors. If a break condition is detected and both IGNBRK and BRKINT are not set, the sequence '\0377', '\0', '\0' is placed in the input stream. If a parity or framing error is detected and IGNPAR is not set, the sequence '\0377', '\0', c is placed in the input stream. No PARMRK would generate a '\0' when parity or framing errors occur.

Also when PARMRK is set and IGNPAR and ISTRIP are not set, a valid '\0377' is read as '\0377', '\0377'.

INPCK	Enable input parity checking. If set, characters with parity errors are ignored (if IGNPAR is set), read as '\0377', '\0377', c (if PARMRK is set), or read as '\0'. If INPCK is not set, parity errors are ignored and the character is placed in the input stream as received.
ISTRIP	Input bytes are stripped to seven bits. This occurs after parity checking but before any special character processing.
INLCR	Translate input \n to \r. If \n is input and INLCR is set, \r is read independent of the setting of IGNCR.
IGNCR	Ignore \r on input. This is independent of the setting of ICRNL (that is, \r is ignored before translated to \n).
ICRNL	Translate input \r to \n. This occurs only if IGNCR is not set. In the previous tty driver, this occurred on read if CRMOD was set and RAW was not.
IXON	Start/stop output control is enabled. A stop character on input causes output to stop. A start character on input causes output to resume. If start and stop characters are the same, they toggle output.
IFLOW	Same as IXON.
IXOFF	When input queue approaches a certain limit, the stop character is sent to the device to stop it from sending input. A start character is sent when the input queue drains below a certain limit. Do not send the stop character if it is VDISABLEd. This is equivalent to the TANDEM bit in the previous tty driver.
ITANDEM	Same as IXOFF.
IXANY	Allow any character to restart output after stop. If not set, all characters are dropped until a start character is seen. If the start character is equal to the stop character, any character will restart after stop. This is equivalent to the DECCTQ bit in the previous tty driver.

IEXTEN — Turn on extensions. Interpret non-POSIX control characters: VDSUSP, VEOL2, VQUOTE, VERASE2, VLNEXT, VFLUSHO, VWERASE, VREPRINT.

IMAXBEL — When set, a **CTRL-g** is sent to the terminal when the input queue is full. If not set, input and output queues are silently flushed when the input queue overflows. In either case the input character is ignored.

The previous tty driver sent **CTRL-g** only if using the new tty line discipline; it always dropped the character. It never flushed the queues.

c_oflag output flags

OPOST — Perform output processing. The values of the other flag bits are checked only if OPOST is set.

In the previous tty driver, this behavior was controlled by either the RAW or LITOUT bits.

ONLCR — Translate \n to \r\n on output.

In the previous tty driver, this happened if CRMOD was set.

ONLCRNL — Same as ONLCR.

OXTABS — Expand tabs to spaces. If a \t is printed, send the proper number of spaces to the terminal to move the cursor to the next tab stop.

ONOEOT — Do not deliver a \04 to the terminal. It is silently discarded.

In the previous tty driver, this was true if RAW, LITOUT, and CBREAK were not set.

c_cflag control flags

CSIZE — Character size mask:

CS5 Characters have 5 data bits

- CS6 Characters have 6 data bits
- CS7 Characters have 7 data bits
- CS8 Characters have 8 data bits.

The previous tty driver determined character size this way:

1. If B134: 6 bits, odd parity

2. else if RAW mode: 8 bits, no parity

3. else if EVENP set: 7 bits, even parity

4. else if ODDP set: 7 bits, odd parity

5. else no parity or both: 8 bits, no parity.

CSTOPB — If set, use two stop bits. Otherwise, use one.

The previous tty driver used two stop bits for 110 baud. Otherwise, it used one.

CREAD — Receiver is enabled. No characters are read if this is not set.

PARENB — Output parity is enabled.

PARODD — Use odd parity. Only used if PARENB is set.

HUPCL — Hang up on last close; drop DTR on device. Set with TIOCHUPCL ioctl.

CLOCAL — Ignore modem status lines. Do not wait for carrier present before I/O.

c_lflag local flags

ECHO — Echo input to the terminal.

ECHOKE — If set when a kill character is typed, visually erase the line with the string "\b\b" for each character. If either ECHOPRT is set or output has been interspersed with recent unread input, ECHOKE behaves exactly as ECHOK.

The previous tty driver used the CRTKIL bit.

ECHOE — If ECHOE is set, visually erase characters with the string "\b\b." If not set (and ECHOPRT is not set), simply echo the erase character when it is typed.

ECHOK — If ECHOKE is not set when a kill character is typed, the kill character is echoed and \n is printed to the screen.

ECHONL — Echo \n when input even if ECHO is off.

ECHOPRT — Visual erase mode for hardcopy terminals. If ECHOE is not set and ECHOPRT is set, consecutive erased characters are erased visually within and /.

This was PRTERA in the previous tty driver.

ECHOCTL — Echo control characters as "^c".

	This was CTLECH in the previous tty driver.
ISIG	If set, each input character is checked against INTR, QUIT, SUSP, and DSUSP. If the input character matches one of these special control characters, the function associated with that character is performed.
ICANON	Enable canonical mode input processing. Characters are assembled into lines, and line editing is performed. If not set, noncanonical mode input takes place with at least MIN bytes received or TIME timeout value expires before a read is satisfied.
ALTWERASE	Use the alternate word erase algorithm for. If set, words consist of adjacent alphanumeric and underscore characters or adjacent nonalphanumeric and underscore characters. If not set, the words are erased as white space delimited entities.
IEXTEN	Enable extended functions. Functions controlled by IEXTEN: EOL2, LNEXT, FLUSHO, WERASE, REPRINT, ERASE2, and DSUSP and the ability of to quote ERASE, ERASE2, or KILL. Also enables ECHOCTL, ECHOKE, ECHOPRT, IMAXBEL, IXANY, ALTWERASE, QUOTE, OXTABS, ONOEOT, and ONLCR.
ECHO	If set, characters are echoed to the terminal as they are typed.
TOSTOP	If set, processes receive a SIGTTOU when attempting to generate output while not in the foreground process group.
NOFLSH	If set, do not flush pending input and output on receipt of a signal. If not set, INTR and QUIT cause input and output to be flushed and a SUSP causes only pending input to be flushed.

Special characters

EOF	End of file. Causes a break in input.
EOL	Extra break-in-input character, like old `t_brkc`.
EOL2	Extra EOL character. Only valid if IEXTEN is set in `lflag`.
ERASE	Erase character char.
ERASE2	Extra ERASE character. Only valid if IEXTEN set in lflag.
WERASE	Word erase character.
KILL	Line kill character.
REPRINT	Line reprint character.
INTR	Interrupt character. Generate SIGINT on input.
QUIT	Quit character. Generate SIGQUIT on input.
SUSP	Suspend character. Generate SIGTSTP on input.
DSUSP	Delayed-suspend character. Generate SIGTSTP when read.
START	Start character. Start IXON flow-controlled output.
STOP	Stop character. Stop IXON flow-controlled output.
LNEXT	Literal-next character. Next character is literal input; no special meaning.
FLUSHO	Flush-output character. Causes output to be flushed; nothing delivered.
MIN	Minimum characters for noncanonical read.
TIME	Read timeout (in tenths of seconds) for noncanonical read.
QUOTE	Quote character. Used to quote erase/kill characters on input.
DISABLE	Disable character. Set others to this to disable special processing.

POSIX and non-POSIX behaviors

5

It is possible for a POSIX application to interact with a non-POSIX application. For example, you could have information from V7.1 of the operating system link with data in a POSIX-compliant version of ConvexOS. Areas of interaction would most likely be in a parent/child relation or a task-to-task communication such as signal handling. Although this interaction is outside the scope of POSIX.1, this chapter discusses some implementations chosen by CONVEX.

Behaviors

Certain system calls behave differently depending on whether POSIX or non-POSIX processes are involved. The following sections explain the behavior of these functions:

- `fork()`
- kill()
- exit()

For more information on signal and process handling, refer to Chapter 3, "Process Primitives," in the CONVEX POSIX Conformance document and IEEE Std 1003.1-1990.

fork()

When a signal is delivered to a process and the process has the signal blocked with sigblock(2) or sigsetmask(2), the signal delivery is pended until the process unblocks the signal. In V7.1 and earlier of the operating system, when a new process was created using `fork()`, pending signals were not propagated to the new process. That behavior is still true for non-POSIX processes that `fork()` while signals are pending. When a POSIX process forks, signals pended for delivery to the parent process are also propagated as pended signals for the child process.

kill()

`kill()` has special cases that allow the sender to send a signal to an entire process group. In these cases, the signal is sent to all processes in the process group (both POSIX and non-POSIX).

A SIGCONT may be sent from a non-POSIX process to any descendent (POSIX or non-POSIX) even if the receiving process has changed sessions and `setuid()`. A POSIX process may not send a SIGCONT to a descendent in the case where the descendent has changed both sessions and `setuid()`.

exit()

If the exiting process is a POSIX session leader, a SIGHUP is sent to each process in the foreground process group of the exiting process' controlling terminal. If the exiting process is a POSIX group leader (but not a session leader) and any other member of the process group is STOPPED, SIGHUP and SIGCONT are sent to each process in the process group. If the exiting process is a non-POSIX group leader, SIGHUP is sent only to STOPPED children. In all of these cases, when a signal is sent to all processes in a process group, the signal is sent to both POSIX and non-POSIX processes.

POSIX and languages 6

POSIX.1 is currently defined in terms of a C programming language interface to the operating system, though interfaces for Ada and FORTRAN will soon be defined.

POSIX and C

The C language was created for use on the UNIX operating system as a general-purpose programming language. Up to a point, applications written in C are portable between computer systems. Programs that utilize only high-level features of C are easier to port than programs that use assembly-language features. But the original definition of C left areas of the language undefined.

To increase the portability of applications, clarify some ambiguities, and set down formal requirements for all aspects of the language, the American National Standards Institute (ANSI) began formulating a standard C language that dealt with the preprocessor, libraries, and the language itself.

Benefits of ANSI C are

- Ease of maintenance
- Increase in efficiency
- New way to declare functions that helps reduce errors by allowing type checking between the call and declaration
- Increase in reliability of C applications
- Requirement that parentheses enforce the order of evaluation of expressions
- Greater optimization because of new rules for floating-point expression evaluation and aliasing

The ANSI C standard is defined in *American National Standard for Information Systems—Programming Language C*, document number X3-J11/90-013.

Compilers

There are two versions of the CONVEX C compiler: one capable of only scalar optimizations and another that performs vector and parallel optimizations. While the former is available on all CONVEX machines, the latter is an optional product available to those who purchase or have a CONVEX C license. This compiler is invoked with the `cc` command name.

Compatibility modes

The CONVEX C compiler, which conforms to the ANSI C standard, offers four modes of compatibility that provide different language features, libraries, and include file contents:

- Default
- Conforming
- Strict
- Backward Compatible

The default mode accepts programs written in ANSI C with POSIX.1 functions and CONVEX extensions. POSIX.1 defines the functions that a C program can use to interface with ConvexOS. These functions are listed in Appendix A.

The conforming mode provides specifications of ANSI C and access to POSIX.1 functions. However, no CONVEX extensions are permitted.

The strict mode of the compiler accepts applications that use only ANSI C language features and functions. Neither POSIX.1 functions nor CONVEX extensions are supported.

The backward-compatible mode accepts no ANSI C features or POSIX.1 functions.

Backward-compatible mode and the implementation-defined features of CONVEX C are detailed in the *CONVEX C Guide*.

POSIX and libraries

The four compatibility modes also select libraries that define system functions for

- CONVEX extensions
- POSIX.1 functions
- ANSI C language specifications
- Backward-compatible features

The following options allow you to select a compiler mode and/or link with the specified libraries:

- `-ext`—CONVEX extensions, POSIX.1, and ANSI C libraries (default)
- `-std`—POSIX.1 and ANSI C libraries
- `-str`—ANSI C library
- `-pcc`—Backward-compatible library

These libraries are searched automatically in the correct order for object code that will be linked into the executable program.

Shipped with CONVEX C are include files that can be interpreted in several ways when a program is compiled. By default, include files are interpreted to define POSIX functions, ANSI C features, and CONVEX extensions. By specifically defining certain conditional compilation symbols, various features can be defined by include files. These symbols can be defined by any of these equivalent methods:

- `#define` directive in the program text
- `-D` option on the command line
- `-D` option in the `CCOPTIONS` environment variable

Defining the symbol _POSIX_SOURCE alone causes the include files to be interpreted to define POSIX.1 and ANSI C features. The symbol _CONVEX_SOURCE may be additionally defined for CONVEX extensions only if the _POSIX_SOURCE symbol is defined. You cannot define _CONVEX_SOURCE if _POSIX_SOURCE has not been defined. The compiler, in the default mode, automatically defines both _POSIX_SOURCE and _CONVEX_SOURCE for you.

When both _CONVEX_SOURCE and _POSIX_SOURCE are defined, include files are interpreted to define all ANSI C features, POSIX.1 functions, and CONVEX extensions.

For compiler usage and examples, refer to the *CONVEX C Guide*.

POSIX and Ada

After IEEE Std 1003.5 has been completed and ratified, CONVEX intends the next major release of the Ada compiler to be POSIX compliant.

POSIX and FORTRAN

After IEEE Std 1003.9 has been completed and ratified, CONVEX intends the next major release of the FORTRAN compiler to be POSIX compliant.

Creating a POSIX application

7

This chapter contains examples of POSIX applications including

- `gid_t`
- `sprintf()`
- `getpwent()`

gid_t example

If the code in is compiled in the conforming mode, as in Figure 7.

```
cc -std pgroups.c
```

the errors in Figure 8 are generated by the C compiler.

Figure 7 pgroups.c source

```
#include <sys/types.h>
#include <limits.h>
#include <unistd.h>
#include <grp.h>

main()
{
int i;
int ngrps;
#ifdef _POSIX_SOURCE
gid_t gidset[NGROUPS_MAX];
#else
int gidset[16];
#endif
struct group *grp;

ngrps = getgroups(NGROUPS_MAX, gidset); /* get my group IDs */
if (ngrps < 0) {
perror("getgroups"); /* oops, something went wrong */
exit(1);
}
for (i = 0; i < ngrps; i++) {
grp = getgrgid(gidset[i]); /* lookup gid in /etc/group */
if (grp)
printf("%s\n", grp->gr_name); /* print group name */
else
printf("%d\n", gidset[i]); /* otherwise, print group ID */
}
exit(0);
}
```

Figure 8 Sample compiler output

```
% cc -std pgroups.c
cc: Error on line 17 of pgroups.c: variable 'NGROUPS_MAX' undefined.
cc: Warning on line 17 of pgroups.c: actual and formal point to different
types
cc: Warning on line 23 of pgroups.c: illegal pointer/integer combination.
%
```

The source of the first error is the definition of the NGROUPS_MAX constant. The next two errors are generated because `gid_t` has a different definition than the pre-POSIX OS. Defining the C preprocessor symbol _POSIX_SOURCE clears these errors. This can be done on the cc command line:

```
cc -std -D_POSIX_SOURCE pgroups.c
```

The `uid_t` type generates similar errors without the preprocessor symbol _POSIX_SOURCE defined.

sprintf() example

Another common problem is the declaration of `sprintf()`. The ANSI C specification in <stdio.h> is

```
extern int sprintf(char *, const char *,
...);
```

Previous versions of ConvexOS and the C compiler defined `sprintf` as

extern char *sprintf();

The local declaration of `sprintf` should be removed from the example in Figure 9. It causes the error message in Figure 10 to be printed.

Figure 9 openfile.c example

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
extern char *sprintf();

int
openfile(char *path, int flags, mode_t mode)
{
int fd;
char buf[128];

fd = open(path, flags, mode);
if (fd < 0) {
(void)sprintf(buf, "open failed on %s", path);
perror(buf);
exit(1);
}
return(fd);
}
```

Figure 10 Compiler output from openfile.c

```
% cc -c -std openfile.c
cc: Error on line 4 of openfile.c: 'sprintf' redeclared: incompatible types.
%
```

Using `getpwent()`

The function `getpwent()` is a CONVEX extension that is not defined by POSIX.1. To successfully compile a program that calls it, the conforming mode should be replaced by the default mode (no command line option, or `-ext`).

Terminal I/O example

Figure 11 shows the original source and Figure 12 shows the changes (in bold face type) required to convert an application using `stty()` or `ioctl()` calls for POSIX control of terminal I/O.

Terminal I/O operations have been standardized in Chapter 7, "Device- and Class-Specific Functions" of POSIX.1

Figure 11 Original source of getpass.c

```
#include <signal.h>
#include <sgtty.h>
#include <sys/types.h>
#include <fcntl.h>

char getpass_buf[9];

char *
getpass(prompt)
char *prompt;
{
struct sgttyb ttyb;
int flags;
char *p;
int c;
FILE *fi;
int (*sig)();

if ((fi = fdopen(open("/dev/tty", O_RDWR), "r")) == NULL)
fi = stdin;
else
setbuf(fi, (char *)NULL);
sig = signal(SIGINT, SIG_IGN); /* ignore interrupt signals */
gtty(fileno(fi), &ttyb); /* get tty flags */
flags = ttyb.sg_flags; /* save flags to restore later */
ttyb.sg_flags &= ~ECHO; /* clear ECHO bit */
stty(fileno(fi), &ttyb); /* set tty characteristics */
fprintf(stderr, "%s", prompt);
fflush(stderr);
for (p = getpass_buf; (c = getc(fi)) != '\n' && c != EOF;) {
if (p < &getpass_buf[8])
*p++ = c;
}
*p = '\0';
fprintf(stderr, "\n"); /* new line for user */
fflush(stderr);
ttyb.sg_flags = flags; /* get old tty flags */
stty(fileno(fi), &ttyb); /* restore tty characteristics */
signal(SIGINT, sig); /* restore interrupt signal handler */
if (fi != stdin)
fclose(fi);
return(getpass_buf);
}
```

Figure 12 POSIX-conformant changes to getpass.c

```
#include <stdio.h>
#include <signal.h>
#include <sys/termios.h>
#include <sys/types.h>
#include <fcntl.h>

char getpass_buf[9];

char *
getpass(prompt)
char *prompt;
{
struct termios ttyb;
tcflag_t flags;
char *p;
int c;
FILE *fi;
void (*sig)();

if ((fi = fdopen(open("/dev/tty", O_RDWR), "r+")) == NULL)
fi = stdin;
else
setbuf(fi, (char *)NULL);
sig = signal(SIGINT, SIG_IGN); /* ignore interrupt signals */
tcgetattr(fileno(fi), &ttyb); /* get tty flags */
flags = ttyb.c_lflag; /* save flags to restore later */
ttyb.c_lflag &= ~ECHO; /* clear ECHO bit */
tcsetattr(fileno(fi), &ttyb); /* set tty characteristics */
fprintf(stderr, "%s", prompt);
fflush(stderr);
for (p = getpass_buf; (c = getc(fi)) != '\n' && c != EOF;) {
if (p < &getpass_buf[8])
*p++ = c;
}
*p = '\0';
fprintf(stderr, "\n"); /* start new line for user */
fflush(stderr);
ttyb.c_lflag = flags; /* get old tty flags */
tcsetattr(fileno(fi), &ttyb); /* restore tty characteristics */
signal(SIGINT, sig); /* restore interrupt signal handler */
if (fi != stdin)
fclose(fi);
return(getpass_buf);
}
```

POSIX functions

8

Table 3 lists the POSIX functions and their purposes.

Table 3 POSIX functions

Function	Purpose
access	File accessibility
alarm	Schedule alarms
asctime	Extensions to time functions
cfgetispeed	Baud rate functions
cfgetospeed	
cfsetispeed	
cfsetospeed	
chdir	Change current directory
chmod	Change file modes
chown	Change owner and group of a file
close	Close a file
closedir	Directory operations
opendir	
readdir	
rewinddir	
creat	Create a new file
ctermid	Generate terminal path name
cuserid	Get user name

Table 3 POSIX functions **(continued)**

Function	Purpose
dup	Duplicate an open file descriptor
dup2	
exec	Execute a file
execl	
execle	
execlp	
execv	
execve	
execvp	
_exit	Terminate a process
fcntl	File control
fdopen	Open a stream on a file descriptor
fileno	Map a stream pointer to a file descriptor
fork	Process creation
fpathconf	Get configurable path name variables
pathconf	
fstat	Get file status
stat	
getcwd	Get working directory path name
getegid	Get real and effective user and group IDs
geteuid	
getgid	
getuid	
getenv	Environment access
getgrgid	Group database access
getgrnam	

Table 3 POSIX functions **(continued)**

Function	Purpose
getgroups	Get supplementary group IDs
getlogin	Get user name
getpgrp	Get process group ID
getpid	Get process and parent process IDs
getppid	
getpwnam	User database access
getpwuid	
group	Group database access
isatty	Determine file descriptor association with a terminal
kill	Send a signal to a process
link	Link to a file
longjmp	Nonlocal jumps
setjmp	
siglongjmp	
sigsetjmp	
lseek	Reposition read/write file offset
mkdir	Make a directory
mkfifo	Make a FIFO special file
open	Open a file
passwd	User database access
pause	Suspend process execution
pipe	Create an interprocess channel
read	Read from a file
rename	Rename a file
rmdir	Remove a directory
setgid	Set user and group IDs

Table 3 POSIX functions **(continued)**

Function	Purpose
setlocale	Extensions to setlocale function
setpgid	Set process group ID for job control
setsid	Create session and set process group
setuid	Set user and group IDs
sigaction	Examine and change signal action
sigaddset	Manipulate signal sets
sigdelset	
sigemptyset	
sigfillset	
sigismember	
sigpending	Examine pending signals
sigprocmask	Examine and change blocked signals
sigsuspend	Wait for a signal
sleep	Delay process execution
sysconf	Get configurable system variables
tcdrain	Line control functions
tcflow	
tcflush	
tcsendbreak	
tcgetattr	Get and set state
tcsetattr	
tcgetpgrp	Get foreground process group ID
tcsetpgrp	Set foreground process group ID
termios	General terminal interface
time	Get system time
times	Process times

Table 3 POSIX functions **(continued)**

Function	Purpose
ttyname	Determine terminal device name
tzset	Set time zone
umask	Set file creation mask
uname	System name
unlink	Remove directory entries
utime	Set file access and modification times
wait	Wait for process termination
waitpid	
write	Write to a file

Part 4 Checkpoint/Restart

Checkpoint Restart overview

9

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.

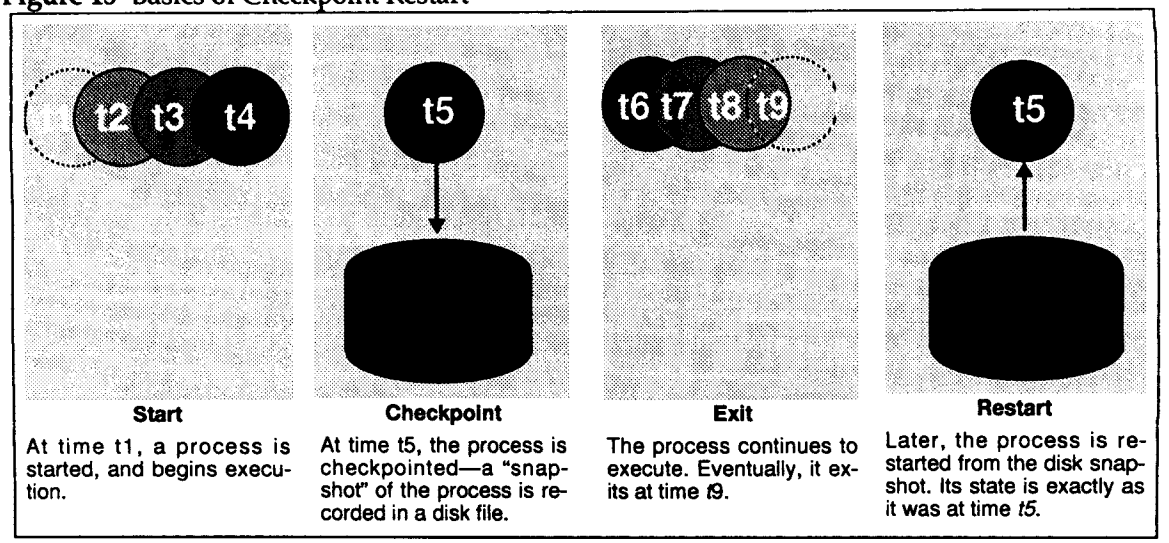
User-level man pages available online for Checkpoint Restart are

- chkpnt(1)
- chkpnt(3)
- chkpnt(3f)
- restart(1)
- restart(3)
- restart(3f)

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 13 shows the basic principles of checkpointing and restarting a process.

Figure 13 Basics of Checkpoint 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 10 "Checkpoint and Restart utilities" and the chkpnt(1) and restart(1) man pages 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 11 "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.

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 0xE4001000 to 0xE6001000.
- 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 memory segments than are specified by the `maxregions` boot time parameter. By default, `maxregions` is set to 1024.
- Use any device other than tty, pty, tape, or `/dev/null`.

[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.

- 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 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, chkpnt saves only the file path name and current offset for each file descriptor that references a files. 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.

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. (You should be certain that no other process had modified the original files.)

When checkpointing applications that use random file access (such as database applications), it is important to use the -C option. These processes do not necessarily modify the end of a file, instead they may make changes anywhere in a file. If the -C option is not used, these applications may not restart properly if the contents of the files are modified by the process after the checkpoint.

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 restarted, 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) man page 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 89 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:

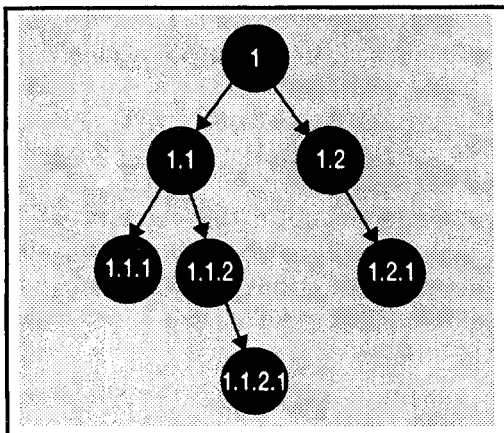
- Register contents.
- File descriptors (for files open to the process).
- Contents of private and shared memory regions used by the process.
- *proc* structure.
- *user* structure.
- *thread* structure.

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.

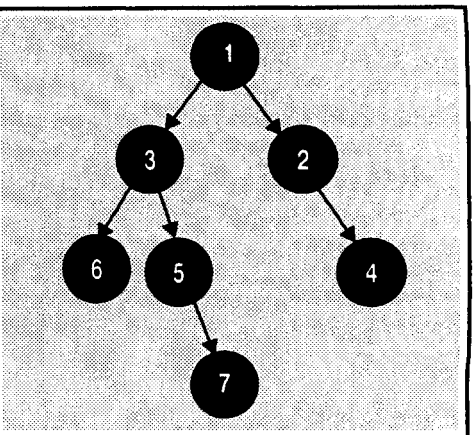
If a process hierarchy is checkpointed, every process in the hierarchy will be stopped by `pattach` before the first one is checkpointed. Processes are stopped by traversing the "family tree" of the hierarchy: the parent is stopped first, and then its children are stopped (youngest child first). After all the children are stopped, the children of the youngest child are stopped (again, youngest child first), etc. Figure 14 illustrates this relationship.

Figure 14 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 14 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 10 "Checkpoint and Restart utilities" and Chapter 11 "Checkpoint Restart programming interface" 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 95.)
- 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.

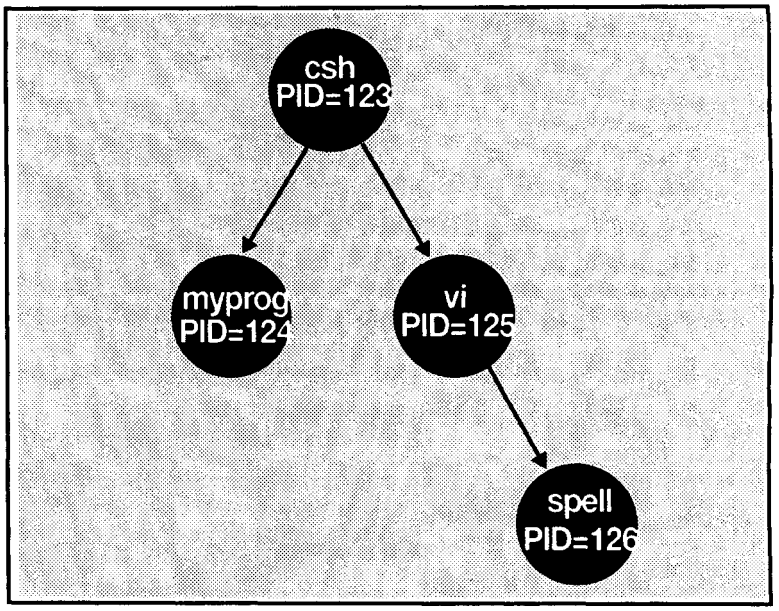
Default checkpoint file name

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 15 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 15 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 15 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) man pages 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 16.

Figure 16 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
  9 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 16). For example, the amount of virtual address space used by the `loop` process is 60 kilobytes. (Refer to the ps(1) man page 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 17.

Figure 17 Using `ls -l` 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 chkir       126 loop.28921     2 out
 126 loop.272     30 loopy          2 scripts
%
```

In Figure 17, `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.

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.

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.

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 82, 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 of 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,[2] 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 for information on how to use this utility.)

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.

[2] Share is an optional CONVEX product

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 (**CTRL-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 proc1
```

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 (`proc1`), it causes the process group ID of `proc1` to become the terminal group. Thus, any input from your terminal will go directly to `proc1`. If you enter commands at your terminal, `proc1` will receive them.

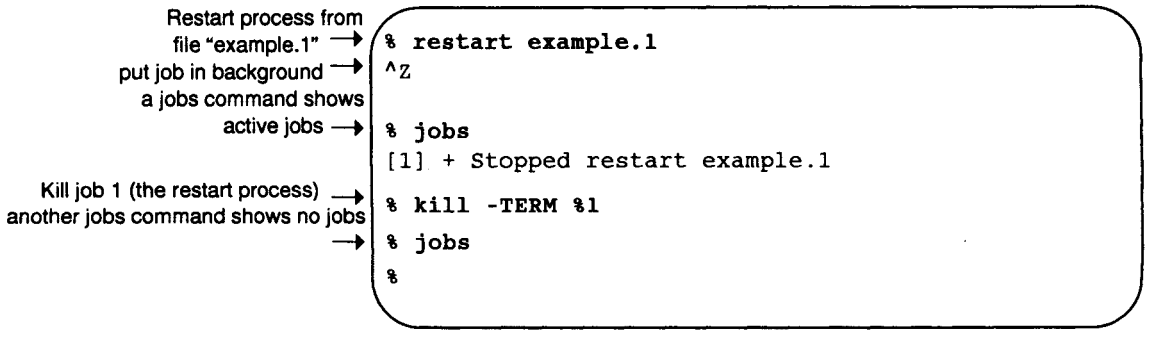
When `proc1` dies, the process is reversed—the `restart` process makes its own group the terminal group, and then exits. (If it did not do this, the shell would generate spurious error messages.)

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 18 illustrates "passing through" signal.

Figure 18 Passing signals through



In this example, the `restart` job was begun in the foreground, but then stopped with a **CTRL-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`.

Checkpoint and Restart utilities

10

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 9 "Checkpoint Restart overview". The chkpnt(1) and restart(1) man pages 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 103.

The options in Table 4 can be specified singly or in combination with other parameters or options.

Table 4 chkpnt 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 parameters in Table 5 can be specified alone or in combination with other parameters or options.

Table 5 `chkpnt` parameters

Parameter	Meaning
-d *directory*	Create checkpoint file in specified directory. Default is current working directory.
-f *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; 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 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 9 "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.

Saving open files

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 directory. (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
```

Specifying checkpoint file name and checkpoint directory

The **`-d`** *directory* option may be used to specify 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
```

The name of the checkpoint file can be specified with the **`-f`** *filename* option. If no checkpoint file name is specified, the default name will be used. Refer to "Checkpoint file name" section on page 89 for information about 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. `-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 specified 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"
```

Forcing checkpointing

The -F option forces 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.

Checkpointing in interactive mode

The -i option is used to invoke checkpointing in interactive mode which 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 110 for more information about -i.

The -L *logfile* option invokes the interactive mode of chkpnt (refer to the description of -i above), and records 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 110 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
```

To use the log file created by a previous interactive chkpnt session that used the -L option (refer to the description of -L above), invoke chkpnt with the -I logfile option. 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

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

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
```

The -p option will 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`.

Sending signals to the target process

The -k signo option sends 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
```

The -K signo option 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

The -n option performs all the checkpointability tests on the target (for diagnostic purposes or to write an interactive log file with `-L`), but does 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
```

A file descriptor (obtained from `pattach`); can be provided with the -p option used instead of pid when debugging the target process.

The -q option will invoke chkpnt in quiet mode (for use with `-F`), which suppresses warning messages. Fatal error messages are still generated. The following example checkpoints process 7519 in quiet mode:

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

The -v option generates verbose output, which includes additional information during checkpointing, such as file descriptors of files open by the target process, and other diagnostics.

```
% chkpnt -v 1232
```

The -X option prints 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 110 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 19 Checkpointing from the shell command line

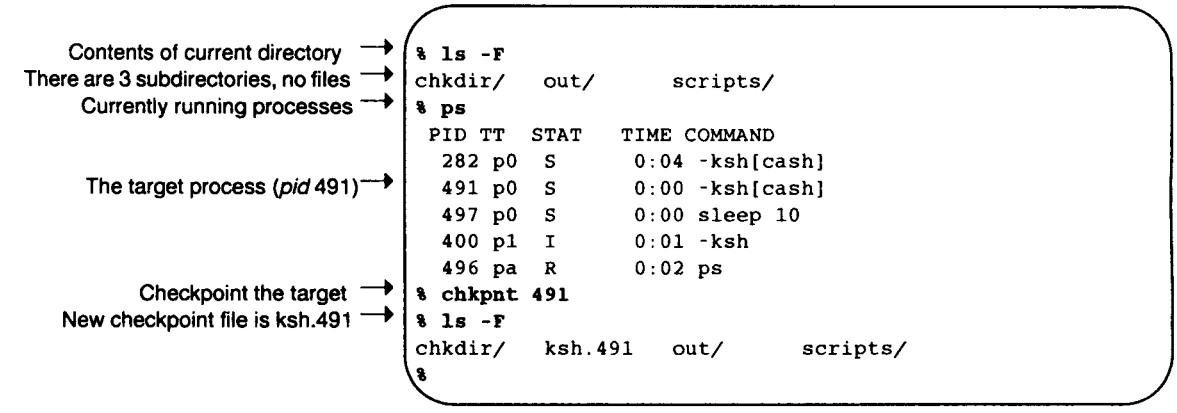
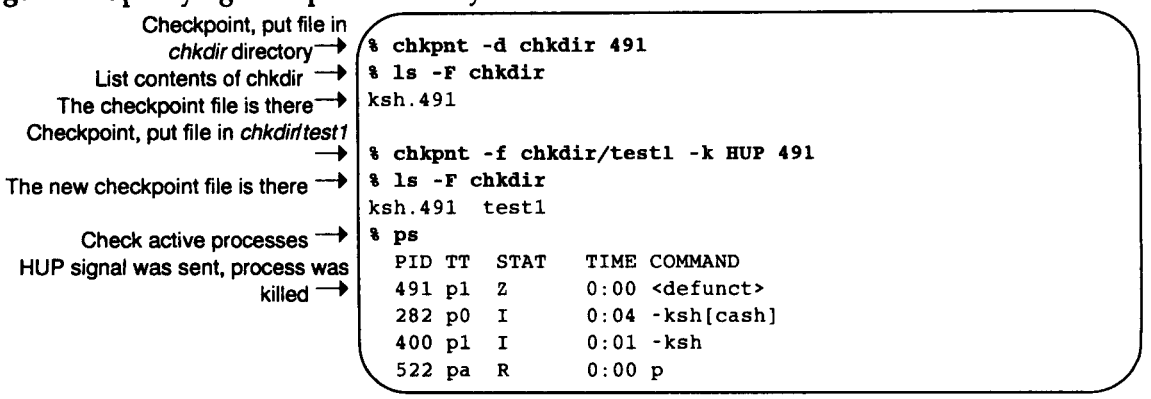


Figure 19 shows a simple checkpointing example. An `ls -F` command shows that there are no files (only subdirectories) in the current working directory. The ps command shows the processes that are owned by the user; the one with pid 491 is the target process. After checkpointing has been completed, the 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 20 show how you can specify a checkpoint directory and a file name.

Figure 20 Specifying checkpoint directory and file name



The last chkpnt command in 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.

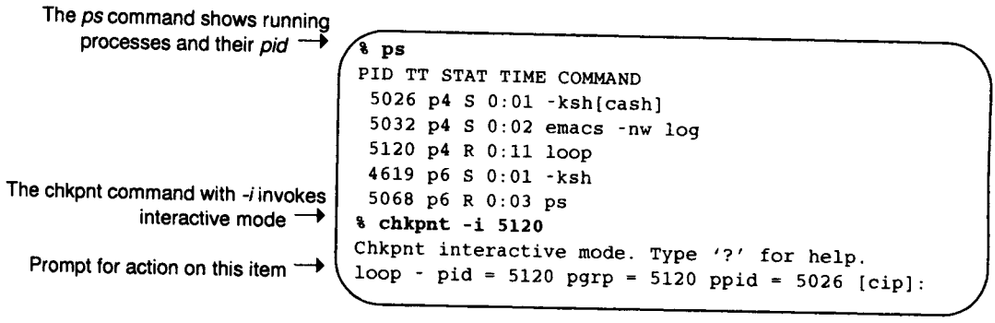
Interactive mode

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 21, 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 21 Invoking *chkpnt* interactive mode



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 22.) 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 22.

Table 6 chkpnt interactive mode commands for processes **(continued)**

Command	Meaning
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 heirarchy (i.e., restarting some and not others) has unpredictable consequences. Some 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 23 Interactive chkpnt commands for file descriptors

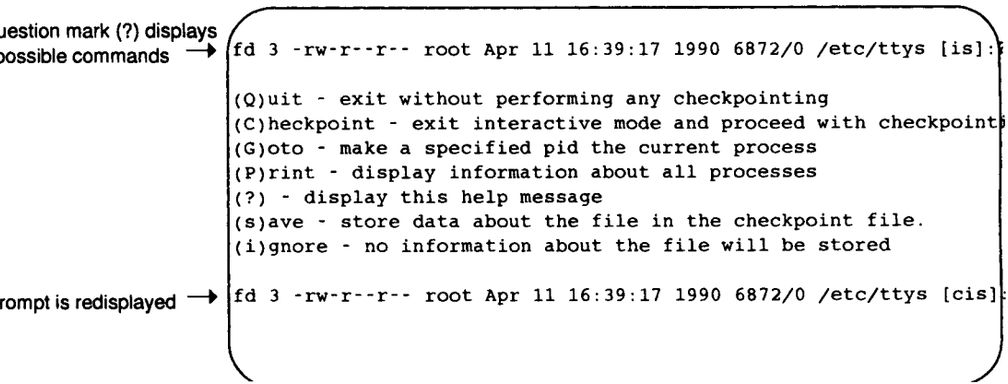
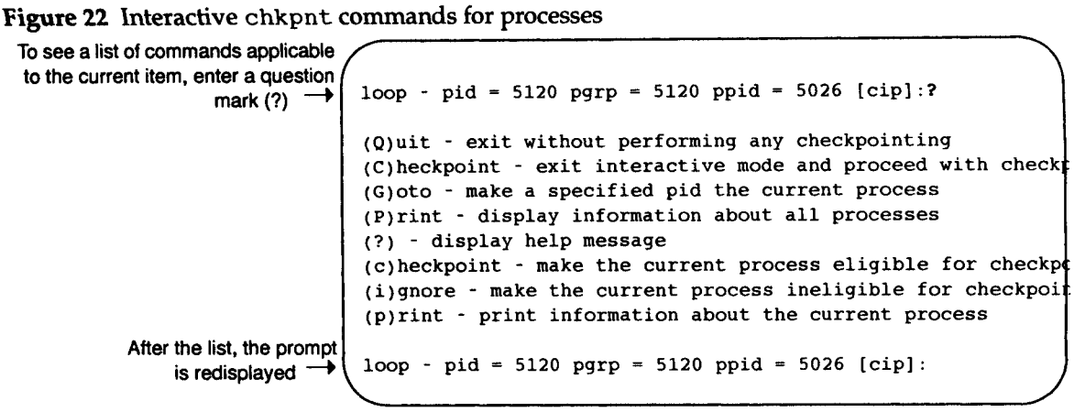


Figure 22 Interactive chkpnt commands for processes



As shown in Figure 22, the commands in Table 6 are available in the interactive mode of chkpnt when the current item is a process.

Table 6 chkpnt interactive mode commands for processes

Command	Meaning
Q	To quit the interactive chkpnt session without performing any checkpointing, enter Q; this cancels everything you have done during this 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.tbi
P	Show information about all processes in 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 will be displayed, unless you have responded to all prompts.) The actual checkpoint processing will be done only when the interactive session has ended.

As shown in Figure 23, the commands in Table 7 are available in the interactive mode of chkpnt for file descriptors.

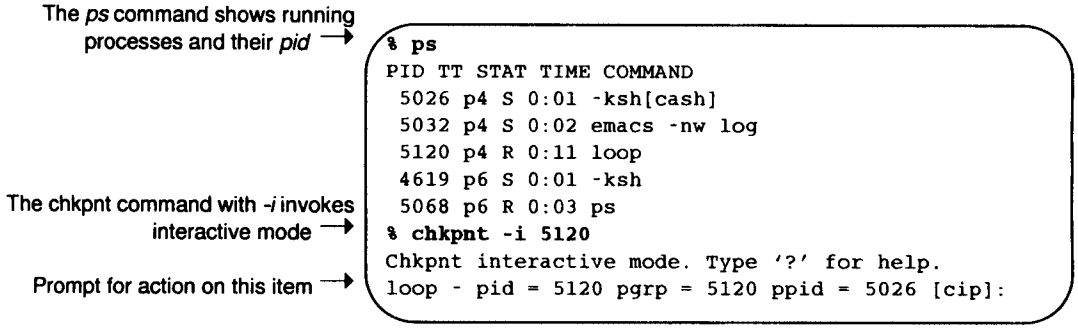
Table 7 chkpnt interactive mode commands for file descriptors

Command	Meaning
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 24.

Figure 21 Invoking *chkpnt* interactive mode



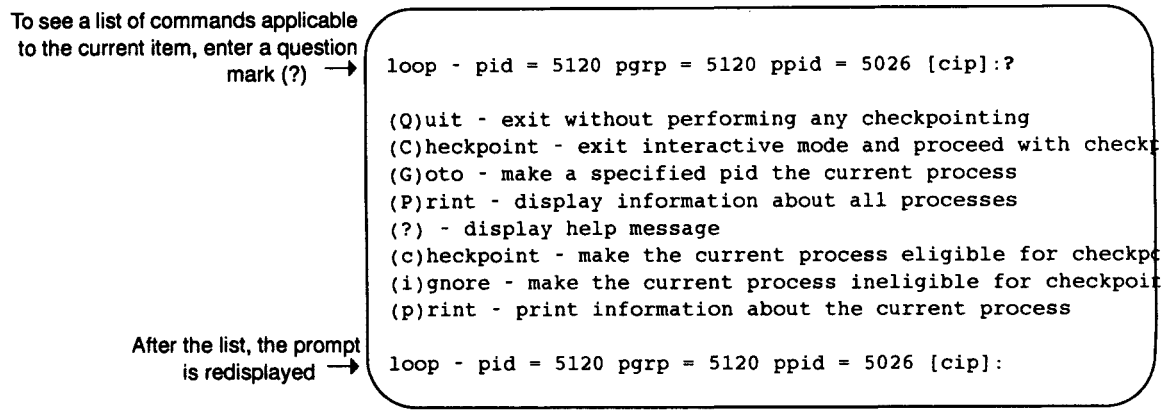
The last line in the screen above is the prompt; it always ends in a colon. You can give chkpnt single character commands by entering them after the prompt. (A list of legal commands is shown in Figure 22.) Some (but not necessarily all) of the commands that are legal for each prompt are shown in square 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 22.

Figure 22 Interactive `chkpnt` commands for processes



As shown in Figure 22, the commands in Table 6 are available in the interactive mode of `chkpnt` when the current item is a process.

Table 6 `chkpnt` interactive mode commands for processes

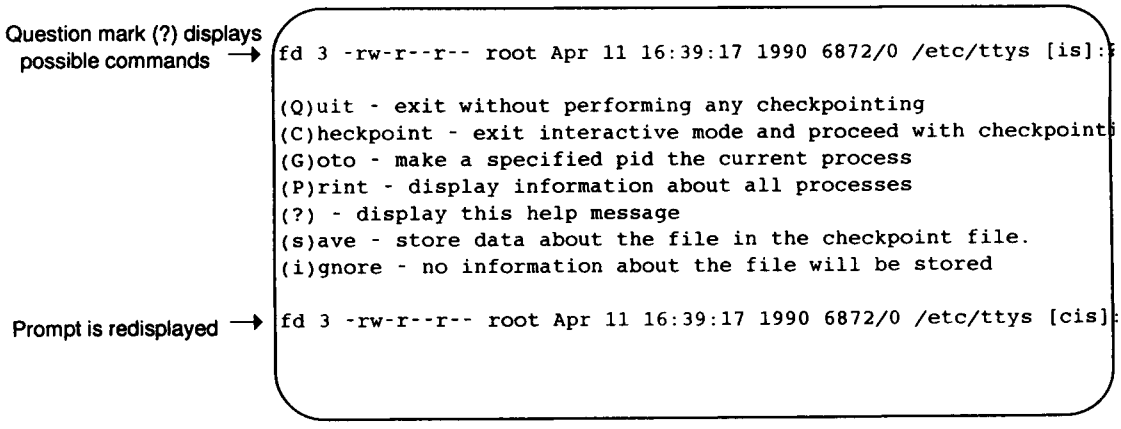
Command	Meaning
Q	To quit the interactive `chkpnt` session without performing any checkpointing, enter `Q`; this cancels everything you have done during this session. This command can be used in response to any prompt at any time during the session.
C	To exit the interactive session and proceed with 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.`tbi`
P	Show information about all processes in 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 will be displayed, unless you have responded to all prompts.) The actual checkpoint processing will be done only when the interactive session has ended.

Table 6 `chkpnt` interactive mode commands for processes **(continued)**

Command	Meaning
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 heirarchy (i.e., restarting some and not others) has unpredictable consequences. Some 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 23 Interactive `chkpnt` commands for file descriptors



As shown in Figure 23, the commands in Table 7 are available in the interactive mode of `chkpnt` for file descriptors.

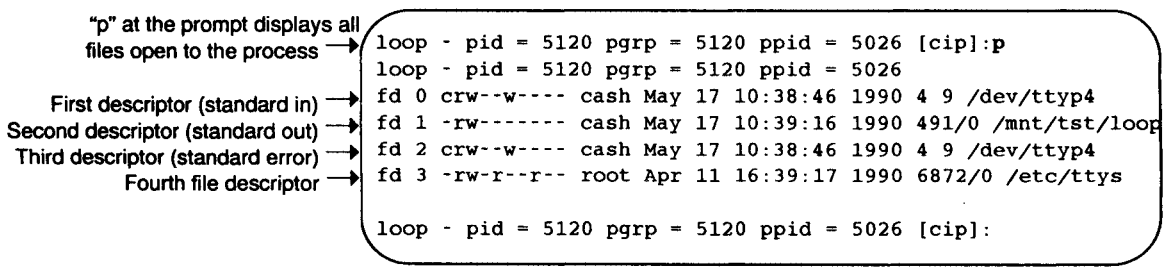
Table 7 `chkpnt` interactive mode commands for file descriptors

Command	Meaning
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 24.

Figure 24 Print information about a process

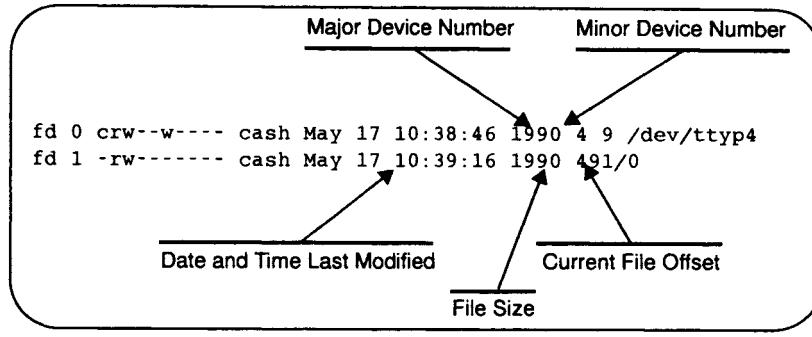


If you enter an upper case P instead of a lower case p, and if a process hierarchy is being checkpointed, all processes and their file descriptors in the hierarchy will be shown.

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 25 shows the information that is contained in each file descriptor entry.

Figure 25 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 26 What processes are running?

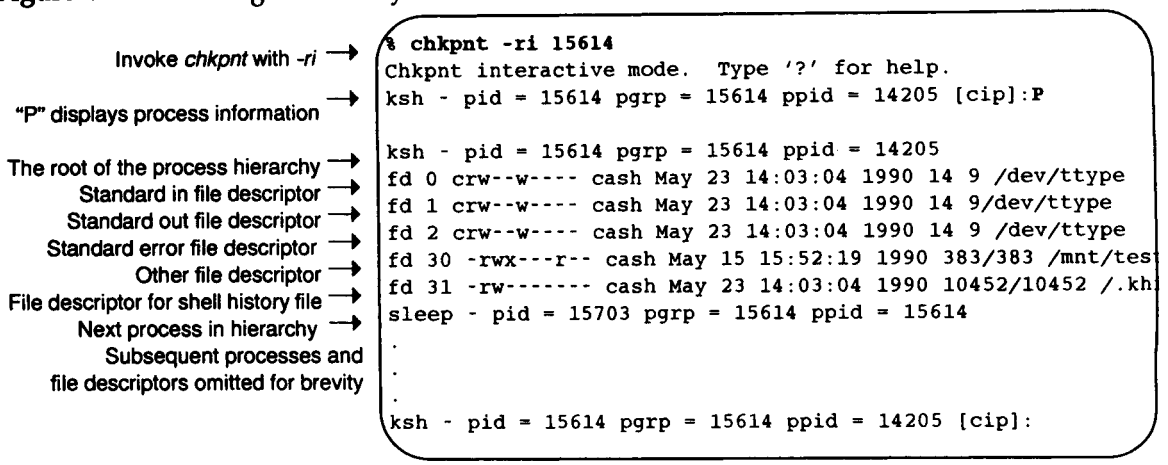
Running processes →

Root process of the hierarchy that will be checkpointed. →

```
% ps -l
... UID     PID   PPID ...    STAT  TT  TIME COMMAND
... 1916  15633  14498 ...     R    p1  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]
%
```

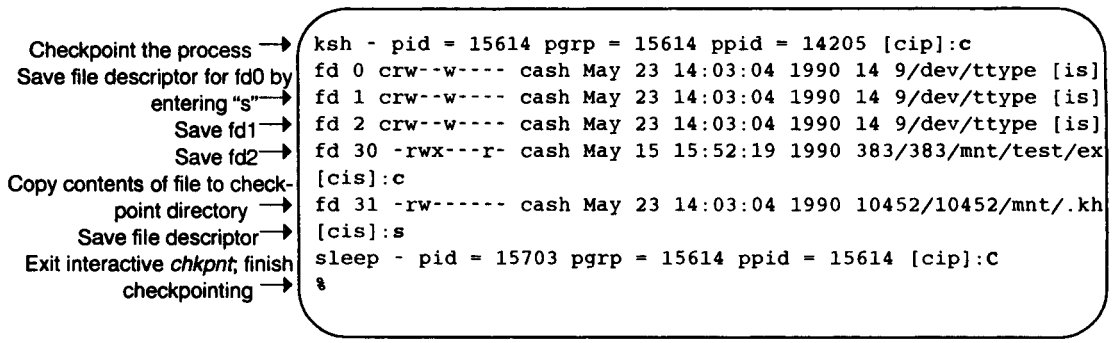
This `ps -l` command shows a number of processes owned by the user. (Not all the fields usually shown by this command appear in the example. Omissions are indicated by the ellipsis marks.) The checkpoint target is a process hierarchy rooted at the `kshv` that has a pid of 15614. To checkpoint this hierarchy interactively, you must use the `-r` (to checkpoint the whole hierarchy recursively) and `-i` flags. As shown in Figure 27, a `P` command will display information about the whole hierarchy.

Figure 27 List the target hierarchy



As shown in Figure 27, the root process (15614) has five open file descriptors. (The other members of the process hierarchy are not shown.) In Figure 28, 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 28 Checkpoint the process



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 29.

Figure 29 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
fd 30 -rwx------ cash May 27 15:17:19 1990 383/383 /mnt/ex3 [ci
% 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 30 shows how subsequent chkpnt commands can use the log file.

Figure 30 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 pathnam
fd 1 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 pathnam
fd 2 crw--w---- cash May 27 16:37:55 1990 0 9 /dev/ttyp0 pathnam
fd 3 -rw-r--r-- root Apr 11 16:39:17 1990 6872/0 /etc/ttys pathn
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 options in Table 8 can be specified singly or in combination with other parameters or flags.

Table 8 `restart` options

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 parameters in Table 9 can be specified by themselves or in combination with other parameters or options.

Table 9 `restart` parameters

Parameter	Meaning
checkpoint_file	Restart the process from the specified checkpoint file (required).
-k *signo*	Send signal by *signo* to target process when restart is complete.
-K *signo*	Like `-k`, but signal is sent to every member of the process heirarchy 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 9 "Checkpoint Restart overview" (especially the sections called "Limitations" and "What happens during checkpointing") 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.

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 124 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 exec 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" 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 31 Restarting from the shell command line

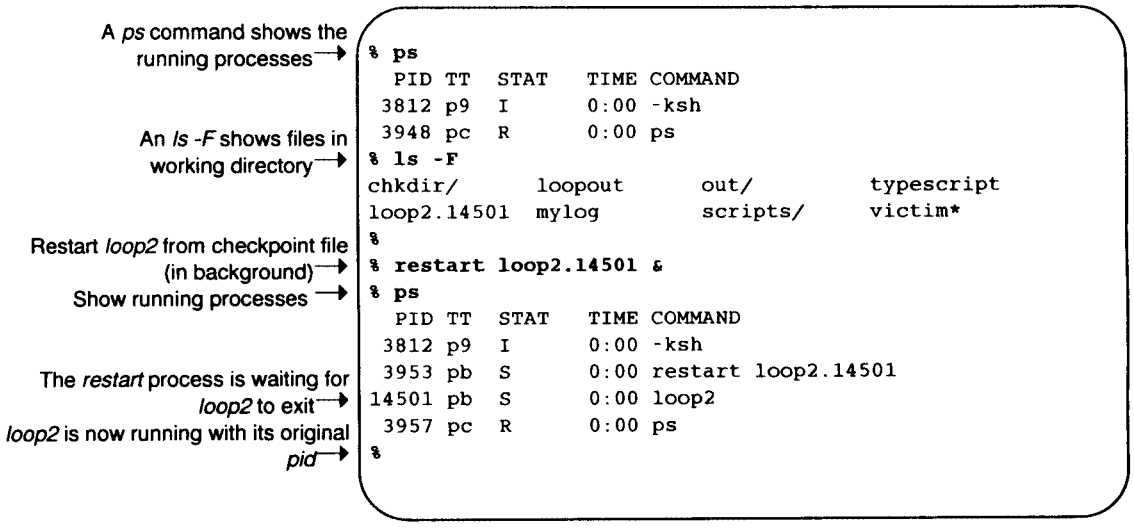


Figure 31 shows a simple restart example. The `ps` command shows the running processes to be a `ksh` and the `ps` command itself. An `ls -F` shows a checkpoint file called "loop2.14501" among the contents of the current working directory. (The name of the checkpointed process was `loop2`; its pid was 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.)

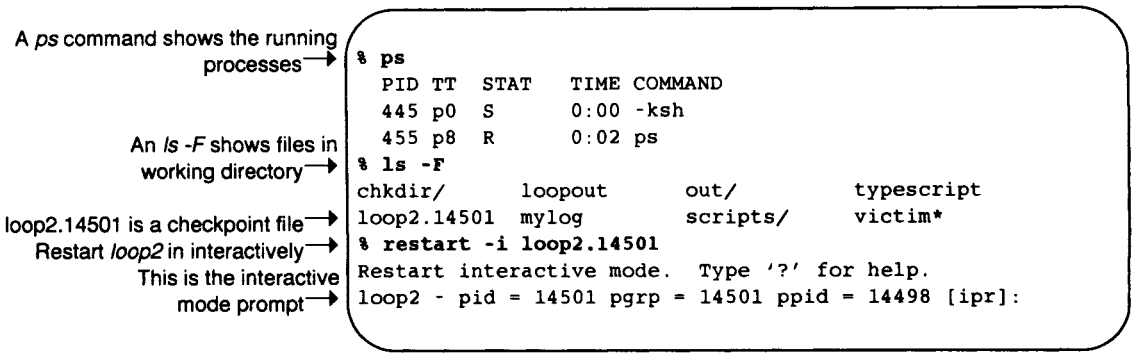
Interactive mode

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 32 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 32 Invoking restart interactive mode



As in interactive `chkpnt` mode, the last line in Figure 32 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 33.) 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 33 shows the legal commands for processes.

Figure 33 Interactive `restart` commands for processes

Enter "?" to see a list of commands →

```
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 children
```

After the list, the prompt is redisplayed →

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

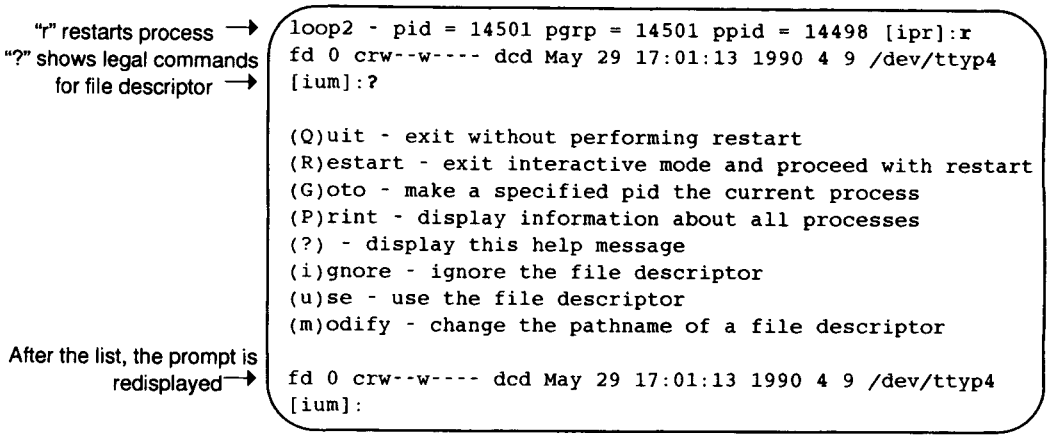
As shown in Figure 33, the commands in Table 10 are available in the interactive mode of `restart` when the current item is a process.

Table 10 Interactive `restart` commands for processes

Command	Meaning
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 34.

Figure 34 Interactive `restart` commands for file descriptors



As shown in Figure 34, the commands in Table 11 are available for file descriptors in interactive `restart`.

Table 11 Interactive `restart` commands for file descriptors

Command	Meaning
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 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 114).

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 35 shows an example of interactively restarting only the root of a hierarchy containing three processes.

Figure 35 Interactive `restart`

Show contents of current working directory →

Restart root process (*pid* is 6934) →

Ignore child process →
Ignore child process →
Ignore child process →
Ignore child process →

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
fd 1 crw--w---- cash Jun  2 14:16:27 1990 0 9 /dev/ttyp0 [ium
fd 2 crw--w---- cash Jun  2 14:16:27 1990 0 9 /dev/ttyp0 [ium
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 the 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

11

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.

General information about checkpointing and restarting processes can be found in Chapter 9, "Checkpoint Restart overview," on page 77.

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.

The `chkpnt()` function invokes the ConvexOS `chkpnt` utility at run time; the utility is then used to do the actual checkpointing. (Refer to Chapter 10, "Checkpoint and Restart utilities," on page 101 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 entire 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. (Refer to the "The checkpoint file" section on page 89 of Chapter 9 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)* man page 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) man pages 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 error codes in may be returned (defined in /usr/include/sys/errno.h)[1]. Table 12 lists return values and error codes.

Table 12 `chkpnt` return values and error codes

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.

[1]Typically, the perror() library function is used to handle error conditions. Refer to the perror(3) man page for more information about this function.

Table 12 `chkpnt` return values and error codes **(continued)**

Code	Meaning
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.

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) man page.

The `restart()` function invokes the `restart` utility at run time· the utility is then used to do the actual restarting. (Refer to Chapter 10 "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 `exec()` had been called (refer to the exec(3) man page 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 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 error codes in Table 13.

Table 13 restart return values and error codes

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.

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 150 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 130 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 134 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 code fragment in Figure 36 is sample user-level code to checkpoint and restore device state in much the same way that the `chkpnt` and `restart` utilities do.

Figure 36 Sample code for checkpoint and restore device state

```
#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, as shown in Figure 37.

Figure 37 Sample code for checkpoint and restore device state

```
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 as shown in Figure 38.

Figure 38 How the driver handles IOCCHKPNT

```
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 as shown in Figure 39.

Figure 39 How the driver handles IOCRESTART

```
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;
```

Figure 39 How the driver handles IOCRESTART (continued)

```
/* 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 40 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.

Figure 40 C programming example

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

main ()
{
        pid_t pid;                      /*Process-id of a 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();
        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 40 C programming example **(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);
}
```

Figure 40 C programming example **(continued)**

```
    /*
     * 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 40, 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 40, 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

Figure 41 contains an example of a FORTRAN program that uses the FORTRAN chkpnt library call.

Figure 41 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
```

Figure 41 FORTRAN programming example **(continued)**

```
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


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

12

This chapter 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). 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.).

`%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.)

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

The target process has a different root directory (refer to the chroot(2) man page) 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.)

`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.)

`%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) man page.)

`%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.)

`%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.) 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) 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 nonswappable, 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) but has been restarted by a non-root user. The nonroot 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) but has been restarted by a non-root user. The nonroot user can not restore the proper root directory. Restart the process as the superuser.

POSIX glossary

A

ANSI
American National Standards Institute

ANSI C
Abbreviated name for ANSI X3J11 committee programming C language standard

C

CONVEX C
CONVEX C compiler that supports the ANSI C language

ConvexOS
CONVEX implementation of the Berkeley UNIX operating system containing POSIX.1 functionality with extensions for supercomputer environments

F

FIFO
First-in-first-out or named pipe

FIPS
Federal Information Processing Standards

N

NBS
National Bureau of Standards

P

PCTS
POSIX Conformance Test Suite; the FIPS test suite that verifies the compliance of an implementation to POSIX.1

POSIX
Portable Operating System Interface for Computer Environments; the general group of proposed standards sponsored by various working committees of the IEEE

POSIX.1
IEEE Std 1003.1-1990, ISO/IEC 9945-1

Index

D

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

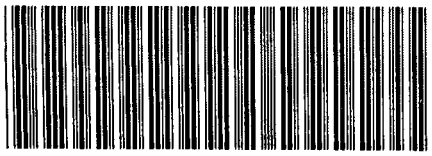
U

V

X

ORDER NUMBER
DSW-053

DOCUMENT NUMBER
710-018330-001



CONVEX PRESS