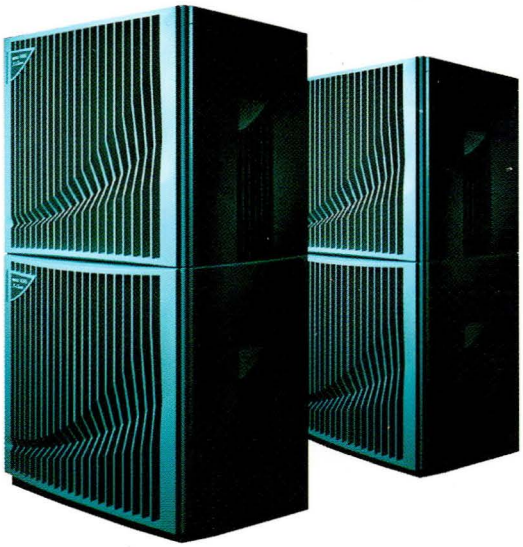
HEWLETT® PACKARD

S-Class and
X-Class Servers



Checkpoint Restart User's Guide

Third Edition

Checkpoint Restart User's Guide

Exemplar S-Class and X-Class Servers

B5655-90027

Third Edition

June 1997

Hewlett-Packard Company
Convex Division
Richardson, Texas
United States of America

Checkpoint Restart User's Guide

Exemplar S-Class and X-Class Servers

B5655-90027

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Revision Information for

Checkpoint Restart User's Guide

Exemplar S-Class and X-Class Servers

Edition	Document No.	Description
Third	B5655-90027	Released in June 1997 with SPP-UX V5.2.
Second	B5655-90005	Released in January 1997 with SPP-UX V5.1.
First	710-032230-000	Initial release June 1996.

Contents

Tables

Figures

Preface

Notational conventions

This section discusses notational conventions used in this book.

CR

CR is an abbreviation for Checkpoint Restart.

`Bold monospace`

In command examples, text shown in **`bold monospace`** identifies user input that must be typed exactly as shown.

`Monospace`

In paragraph text, `monospace` identifies command names and system calls.

In command examples, `monospace` identifies command output, including error messages.

In command syntax diagrams, text shown in `monospace` must be typed exactly as shown.

Italic

In paragraph text, *italic* identifies new and important terms and titles of documents.

In command syntax diagrams, *italic* identifies variables that must be supplied by the user.

[ ]

In command syntax diagrams, square brackets indicate optional data.

The following command example indicates that the variable *output_file* is optional:

`command` *input_file* [*output_file*]

KEYCAP

In paragraph text, text shown in **KEYCAP** indicates keyboard keys you must press to execute the command. 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.

Notes and cautions

This document presents notes and cautions in the following formats.

Note

A Note highlights supplemental information.

Caution

A Caution highlights information necessary to avoid damage to the system.

Associated documents

Using Checkpoint Restart may require information not specific to the tasks described in this document. You can order these books from the Hewlett-Packard:

- For more information on using SPP-UX, refer to the *HP-UX Reference* (B2355-90004).
- For information on administering SPP-UX, refer to the *SPP-UX System Administration Guide* (B5655-90002).
- For more information about batch processing, refer to the *NQS User's Guide: Exemplar S-Class Servers* (B5589-90002).

Ordering documents

To order additional copies of this document, send requests to:

Hewlett-Packard Company
Convex Division
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Please include the order number (xxxxx-9xxxx number) or the exact title of the document.

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard 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 Hewlett-Packard office.

You can also use the `contact` utility, if you would like to report any problems you may have with SPP-UX or its associated documentation. For more information refer to the contact(1) man page.

How Checkpoint Restart works

1

What is Checkpoint Restart?

Checkpoint Restart (CR) permits you to save the state of a selected process or process hierarchy to disk files and later restart the process (hierarchy) from the saved files.

CR is useful for application programs that must run for a long time and that—once halted—cannot be started again from the beginning without wasting significant time and resources. Using CR, you can save such applications to files or *checkpoint* them, either at the command prompt or by a periodically executed script that includes CR commands.

If the application is then halted (whether by a system crash, by scheduled system downtime, or by the operator for system maintenance), you can restart it as it was when it was last checkpointed. You can restart the process from the same file as often as you like.

User-level man pages available for Checkpoint Restart are:

chkpnt(1)	restart (1)	rmckpt(3)
chkpnt(3)	restart(3)	
cnx_chkpnt(3)	cnx_restart(3)	

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

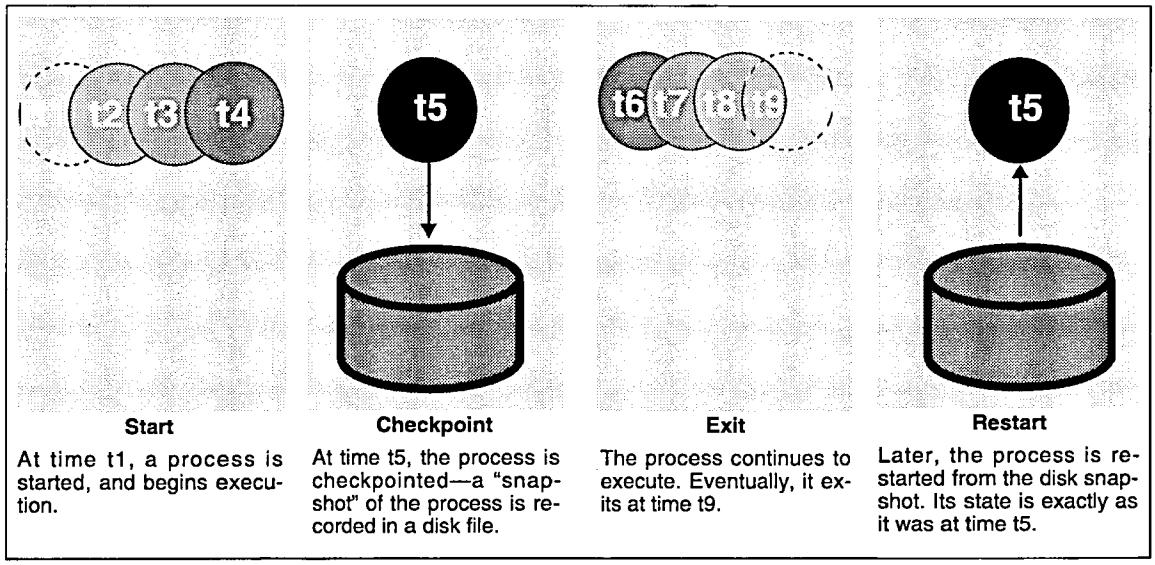


Figure 1 Basics of Checkpoint Restart

Because you can checkpoint related processes (processes linked through parent-child relationships) automatically as a group, you can restore entire process hierarchies. In this way, you can restore all processes related to a certain job or spawned by a particular shell.

CR consists of utilities and library routines that allow you to checkpoint and restart processes interactively, in shell scripts, through NQS, and in C, C++, and Fortran programs.

Utilities

Two utilities (shell commands) allow you to use Checkpoint Restart by entering commands at the SPP-UX prompt or in SPP-UX shell scripts:

`chkpnt`

Checkpoints processes (saves their states to a *checkpoint file*). See "The checkpoint file" section on page 14 for more information on these files.

`restart`

Restarts processes from checkpoint files.

These utilities and their options are discussed in the chapters, "Checkpointing processes" and "Restarting processes."

Library routines

The following library routines provide the main programming interface to Checkpoint Restart. These routines can be used in C, C++, and Fortran programs.

- `chkpnt()` — checkpoints processes
- `cnx_chkpnt()` — checkpoints processes
- `restart()` — restarts processes
- `cnx_restart()` — restarts processes
- `rmckpt()` — removes checkpoint files

Refer to the chapters "Programming in C and C++ with Checkpoint Restart" and "Programming in Fortran with Checkpoint Restart" for more information about the CR library routines. These routines are also described in the following man pages:

chkpnt(3)	restart(3)	rmckpt(3)
cnx_chkpnt(3)	cnx_restart(3)	

Checkpoint Restart and NQS

NQS is a set of utilities that allows you to control scheduling, queueing, and execution of processes in batch mode. You can checkpoint and restart processes under NQS control by using CR features built into NQS utilities. The following NQS utilities have Checkpoint Restart capabilities:

`qchkpnt`

Utility that allows you to checkpoint any process being run by NQS. You may specify an interval at which this process is periodically checkpointed. See the qchkpnt(1) man page for more information.

`qmgr`

Queue management program that provides commands that allow you to checkpoint processes being run through NQS and requeue them so that they will later be restarted. See the qmgr(1) man page for more information.

`qrestart`

Utility that allows the user or batch administrator to restart previously checkpointed NQS requests. See the qrestart(1) man page for more information.

For more information on the NQS batch queueing system, refer to the *NQS+ User's Guide* (B5589-90002).

Checkpoint Restart and large files

The `chkpnt` utility can checkpoint processes that have *large files* open. A large file is a file that is greater than 2^{31} - 1 bytes in size (approximately 2 gigabytes). The following utilities have been modified to properly handle large files:

- `cat`
- `chmod`
- `cp`
- `dd`
- `dump`
- `dumpf, cnx_dumpfs`
- `find`
- `fbackup`
- `frecover`
- `fsck`
- `fsclean`
- `fsirand`
- `ftp`
- `ls`
- `mkfs, cnx_mkfs`
- `mount`
- `mv`
- `ncheck`
- `newfs, cnx_newfs`
- `restore`
- `rcp`
- `rm`
- `tail`

See the largefiles(1m) man page for more information.

For additional information on using and programming large fiiles, refer to the *SPP-UX Large Files User's Guide* (B5655-90032).

Limitations

This section describes the following Checkpoint Restart limitations:

- Performance
- Uncheckpointable processes
- Files used by checkpointed process
- PID conflict
- Checkpointing and restarting on different systems
- Checkpointing and interprocess communication

Performance

Because it may take several seconds or minutes to checkpoint any given process, set the intervals at which processes are checkpointed in hours, not seconds or minutes.

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

The performance of CR is affected by the disk-read and disk-write times of the system you are working on. CR may seem slow when checkpointing and restarting large processes. However, response times are due mainly to system limitations, not CR itself.

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 CXdb, as well as utilities such as `ps`, `pstat`, `syspic`, `uptime`, `w`, and `chkpnt` itself).
- Use virtual memory addresses in the range from 0xEF801000 to 0xF0000000. Generally, the compilers do not produce programs that use this memory range.
- 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.

  However, 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 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, or `/dev/null`.
- Use `vfork()` to spawn a child, and the child has not yet performed an `exec()`.
- Have unlinked a file that was mapped into memory with the `mmap()` system call.
- Have process hierarchies with more than 250 members.
- Use PVM message passing. PVM's use of sockets prevents checkpointing. You can, however, checkpoint applications that use MPI message passing. To do so requires the MPI_CHECKPOINT environment variable to be set. For more informationn on MPI refer to the *HP MPI User's Guide* (B6011-90001)

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 files. Data contained in files open to the target process is *not* saved by the checkpoint process. However, you can use `chkpnt -C` to copy the open files to the checkpoint directory (the current working directory or the directory specified by using `-d` *directory_name*). By copying the files to the checkpoint directory, all data in files open to the target process is saved. Use `restart -C` when you restart the process to use the files in the checkpoint directory.

PID conflict

When you restart a process, it is by default given the same PID as it had when it was checkpointed. If that PID is the same as the PID of another process already on the system, restarting fails. For example, suppose that a process with PID 4004 is checkpointed and then killed. 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 now fails because two processes with the same PID cannot exist on the system at the same time.

You can force `restart` (using `-F`) 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 original PID. Also, forcing a restart can result in corrupted or lost data.

Caution

Do not force a restart unless you are prepared to handle possible corruption of or loss of data.

Checkpointing and restarting on different systems

Consider the following compatibility requirements when you restart 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 SPP-UX that is different from the release under which it is being restarted.
- Restarting a process on a different machine from the one on which it was checkpointed, requires the hardware architecture of the two machines be compatible.

  In general, if the executable is portable between two architectures (in other words, it runs on both machines), then you can checkpoint the process on one and restart it on the other.

  Occasionally, restrictions are imposed by hardware architecture. For instance, a checkpointed process that has threads on multiple nodes cannot be restarted on a system with fewer nodes than was used by the checkpointed process.

Information about subcomplexes and the system type is not saved to the checkpoint file. (A subcomplex is a logical entity that provides control over the allocation of processors and physical memory to different applications and users.)

Checkpointing and interprocess communication

Processes that use interprocess communication (IPC) can be checkpointed and restarted on Exemplar systems. IPC channels include message queues, shared memory, and semaphores. IPC channels are checkpointed if the following conditions are met:

- Message queues are checkpointed only if the last `msgsnd(2)` message operation was performed by one of the processes within the process family that is being checkpointed.
- Shared memory is checkpointed only if the memory region was acquired (using `shmget(2)`) by one of the processes within the process family that is being checkpointed.
- The maximum number of shared-memory regions that can be restarted for a family of processes is limited to the maximum attached shared-memory segments per process parameter that can be acquired using the `shmctl(2)` system call.
- Semaphores are checkpointed only if the last semaphore operation, `semop(2)`, was performed by one of the processes within the process family that is being checkpointed.
- All channels must have appropriate user permissions.
- All channels to be restarted must be removed before `restart(1)` is invoked.

What happens during checkpointing

When you checkpoint a process, the `cnx_pcontrol(2)` system call stops the execution of that process. Then, a checkpoint file is written to disk for the checkpointed process. The next section, "Checkpointing process hierarchies," explains what happens when you checkpoint a process hierarchy.

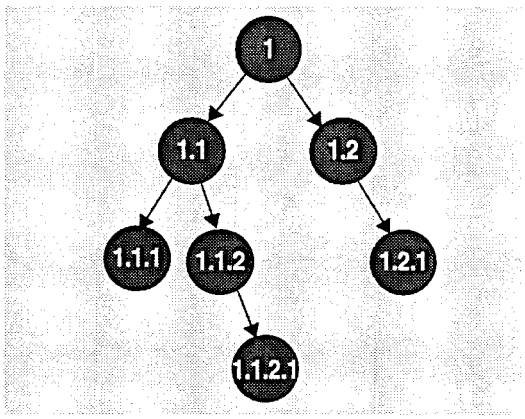
The resultant checkpoint file contains all information necessary to restart the process in the same condition as it was in when it was checkpointed. Refer to "The checkpoint file" section on page 14 for more information on the checkpoint file.

Checkpointing process hierarchies

In addition to checkpointing a single process, you can checkpoint and restart—as a group—an entire process hierarchy (that is, a group of processes that have a common origin). You cannot do this with unrelated processes—you must checkpoint and restart such processes individually.

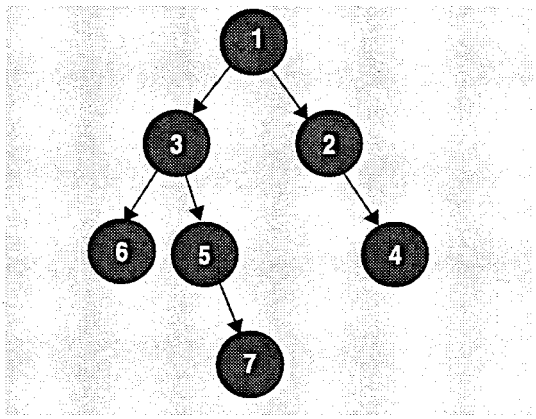
When a process hierarchy is checkpointed, every process in the hierarchy is stopped by `cnx_pcontrol(2)` before the first one is checkpointed. Processes are stopped by traversing the "family tree" of the hierarchy: the parent is stopped first, then its children are stopped (youngest child first).

After all children are stopped, the children of the youngest child are stopped (again, youngest child first), etc. Figure 2 illustrates this relationship.



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.

Figure 2 Checkpointing process hierarchies

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

Note

The chkpnt utility or library function will not overwrite existing checkpoint files with new ones that have the same name. (An error message will inform you if this situation arises.) If the same process must be checkpointed more than once, specify a new file name using `-f` *checkpoint_file* or force overwriting using `-F`.

You can restart any subtree (any set of branches beginning at a tree node) of a process hierarchy independently. For example, in Figure 2, processes 1.1, 1.1.1, 1.1.2, and 1.1.2.1 are members of a subtree starting at the node fro process 1.1.

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 process 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 creates a checkpoint file for itself that is 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 does not wait forever after it is restarted.

Files, pipes, and devices

Each target process has a file descriptor (unique identifier) for every file that has been opened by the process. Because all SPP-UX 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 (for example, ttys). (Regular files are files that are not directories or special files; see the intro(7) man page for information on special files.)

Because Checkpoint Restart 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.

Information about any regular files, device files, pipes, or named pipes that have been opened by the checkpointed process is recorded, and can be restored when the process is restarted. Any data that is in transit through a pipe is stored and replaced in the pipe when the hierarchy is restarted.

The contents of files open to a checkpointed process are not automatically saved (you must explicitly request this action with the chkpnt -C option). Ordinarily, only the file pathname and current position for each open file are 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 must not have changed since the process was checkpointed. If the modification timestamp of such a file indicates that the file has changed since the process was checkpointed, CR issues a warning—unless you deliberately override this safeguard with the -q (suppress warning messages) option of restart.

Access permissions

A number of restrictions are enforced to prevent the Checkpoint Restart system from being used to bypass system security. These restrictions are:

- When you checkpoint a process, the `chkpnt` process must have permission to access the target process. The `chkpnt` process (and the user who starts it) must be running as root, or must have the same effective UID as the target process.
- 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 `chkpnt` process must be a member of the group that has 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. Information recorded in the file for each checkpointed process includes:

- Register contents
- File descriptors (for files open to the process)
- Contents of private and shared memory regions used by the process
- Process state
- Thread state

Checkpoint file name

You can either allow the `chkpnt` utility to assign a default name to the checkpoint files of target processes or you can 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. For example, the default checkpoint file name of a `csh` process with a PID of 14768 would be csh.14768.

When you checkpoint a process hierarchy, the name of the checkpoint file for each member of the hierarchy is given the form

hierarchy_list.program_name.pid

where *hierarchy_list* 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 *hierarchy_list* 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 *hierarchy_list* is abbreviated to the name of the root process followed by three periods (...).

A checkpointed process hierarchy is illustrated in Figure 3. This hierarchy consists of four processes:

- a `csh` shell with a PID of 123 spawned two processes
- `myprog` with a PID of 124
- `vi` with a PID of 125
- a `spell` process with a PID of 126 spawned by the vi process

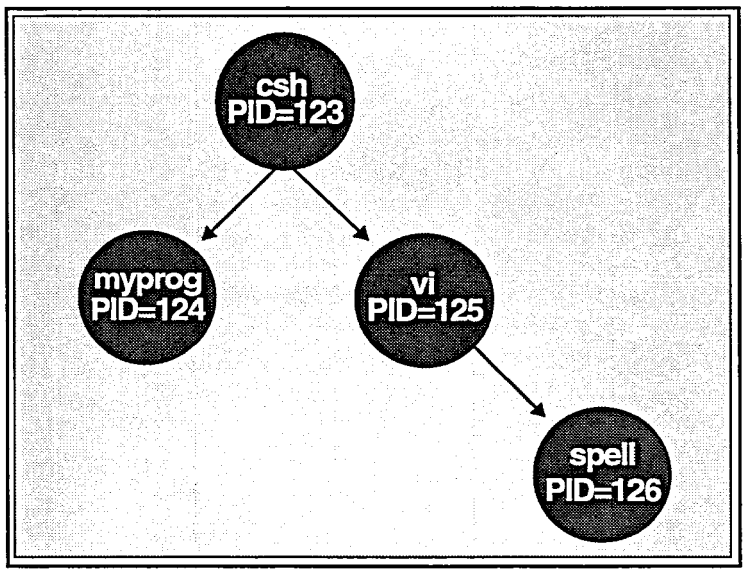


Figure 3 Checkpoint file names for a hierarchy

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 names described above, you may assign a checkpoint file name yourself by specifying it with the

- `-f` option to the `chkpnt` utility
- *name* parameter in the `cnx_chkpnt()` function

If you are checkpointing a single process and you assign a file name, that name is used without appending the PID of the target process.

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

name.hierarchy_list.program_name.pid

where

name
: is the name you specified

hierarchy_list
: is a list of the program names of all the processes in the hierarchy from the parent of the checkpointed process to the child of the root process

program_name
: is the program name of the checkpointed process

pid
: is the PID of the checkpointed process

For example, if the name "abc" is specified as the checkpoint file name when the hierarchy in Figure 3 is checkpointed, the following checkpoint files are created:

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

Checkpoint file format

The checkpoint file conforms to the System Object Module (SOM) format and is similar to a standard SPP-UX ESOM core file. Refer to the chkpnt(4) and core(4) pages for more information about this format.

Because the format of checkpoint files is similar to that of SPP-UX core files (though the checkpoint file contains information in addition to that normally found in core files), SPP-UX debugging utilities treat checkpoint files as though they were core files. For example, CXdb (the SPP-UX visual debugger) can read and alter the portions of the checkpoint file that correspond to core files.

Checkpoint file size

The exact amount of disk space used by the checkpoint file for any given process cannot be calculated in advance. The entire address range in use by the checkpointed process—including text, data, and stack—is written to the file.

Generally, the checkpoint file is *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**:

```
% loop &
[1]     356
% ps -l
  F S   UID   PID  PPID  C PRI NI  ADDR   SZ    WCHAN TTY       TIME COMD
  1 S 12345   128   127  0 100 20  1000   72 f00a473e ttyp1     0:10 ksh
  1 S 12345   356   128  0 104 24  1000   20 f007e32e ttyp1     0:00 loop
  1 S 12345   357   128  1 100 20  1000   16 f007efba ttyp1     0:00 ps
```

In the example above, 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. For example, the amount of virtual address space used by the `loop` process is 20 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` command after checkpointing the process, as shown here:

```
% chkpnt 356
% ls -l
total 350
-rwxr-xr-x   1 foo      os          90556 Aug 17 10:08 loop
-rw-rw-r--   1 foo      os          95284 Sep  7 12:49 loop.356
% ls -s
total 350       178 loop              172 loop.356
```

In this example, `ls -l` shows the checkpoint file (loop.356) to contain 95,284 bytes; `ls -s` shows that it actually occupies 89 kilobytes (178 blocks x 512 = 91,136 bytes) of physical disk space.

The two sizes may be different because the `chkpnt` utility does not write blocks of zeroes to disk, provided 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` "seeks" 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 `du` is the amount of physical disk space that is saved by "seeking over" blocks of zeroes in the process data structure.

What happens during restart

Restarting is the reverse of checkpointing: beginning with the root process, the checkpoint files for all members of checkpointed process hierarchies are read, and the processes are restarted. 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. However, the restart will fail if the PID of the checkpointed process is in use by another process when you attempt you to restart. See the section "PID conflict" for more information.

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, you might need 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 hands the file descriptors to the (restarted) target process. Because it has the file descriptors, the target process 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

You can checkpoint processes or process hierarchies on one Exemplar system, then restart them on a different system. Resources (such as files, CPUs, memory and nodes) that are needed by the process must be present on the new system. The subcomplexes used when checkpointing and restarting can be different, but the resources mentioned above must be the same upon checkpoint and restart. See the scm(1) man page for more information on subcomplexes; see the sysinfo(1) man page for information on system resources.

For example, if the process requires access to certain files, you must first copy those files to the new system. Because 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 section "Checkpointing and restarting on different systems," the hardware architecture and operating systems of the two machines must be compatible.

User ID, group ID, and parent PID of restarted processes

The following sections discuss how various IDs affect restarting:

- Restoring the target process UID, GID, and group access lists
- File permissions and communication problems after changing UID or GID
- Parent process ID (PPID) of target process

Restoring the target process UID, GID, and group access lists

If `restart` is running with the effective UID of root, the target process is 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 is "inherited" from the process that invoked `restart`.

Because checkpoint files may be altered by users who have write permissions to them, you must use caution 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, thus compromising the security of the system.

File permissions and communication problems after changing 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, it may not be able to access files owned by the original user or group.

Also, interprocess communication (for instance, 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. If you checkpoint such a process after it changed its UID or GID, it may not be possible for you to open the files that the target process needs, unless you are running `restart` as root.

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 itself, 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. Further, because 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 single processes also.

Restarting and the current working directory

The current working directory of the restarted target is the same as it was when the target was checkpointed, provided that this directory still exists when the process is restarted and permissions are set appropriately. If the current working directory of the target no longer exists or access is denied, the restart fails.

However, `restart` does not fail if the current working directory of the target was removed *before* it was checkpointed. Such processes are restarted with an unlinked current working directory in /tmp; this may cause any `fstat()` function call issued by the target to return unexpected values after restart.

Job control

The following sections discuss:

- Control terminals
- Process groups and terminal control
- Passing signals through to the target process

Control terminals

Most interactive processes have a *control terminal*—a terminal from which they expect input and to which they send output. The control terminal is usually the login terminal of the user who started the process.

When you restart such a process from a terminal, the target process does not attempt to use its old control terminal. Instead, it will "talk" to you via the terminal you are now using. You can send the process input from this terminal, and output is 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 invoked 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` (window size change) signal is sent to the target process.

Using the `restart -F` option to force restarts

When you restart 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 (that is, a process that was probably started interactively by a user) is checkpointed and then later restarted via NQS, 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 to what 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 (for instance, 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.

If you restart a process in the foreground, the situation is a bit more complicated. 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`—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 goes directly to `proc1`. If you enter commands at your terminal, `proc1` receives them.

When `proc1` dies, the restart 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 to the target process

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`
- `SIGUSR2`

All other signals are assumed to be input to `restart`.

The example in Figure 4 illustrates "passing through" signals.

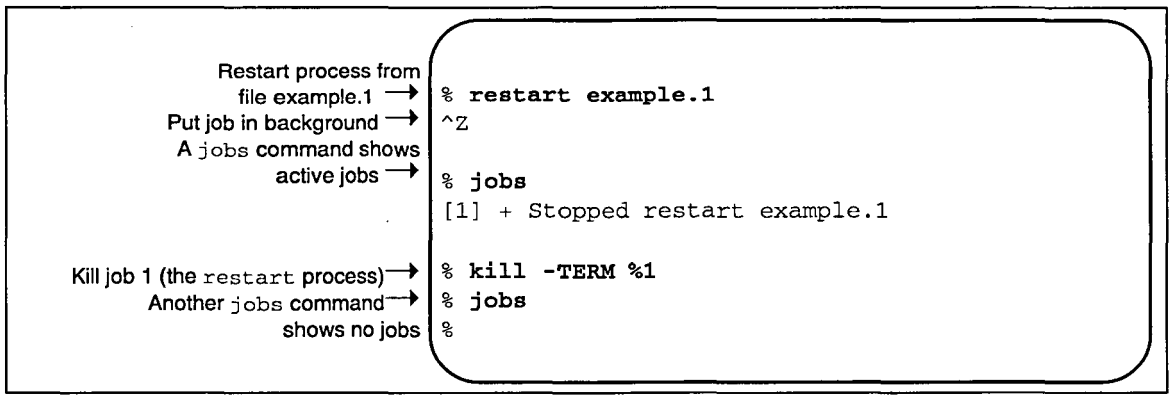


Figure 4 Passing signals through to the target process

In Figure 4, the `restart` job began in the foreground, but was then stopped with a **CTRL-z** (`SIGSTP`) command. When this command was given, the `restart` process passed it on to the target, which then stopped. Because 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; 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`.

Checkpointing processes 2

This chapter discusses the SPP-UX command-line utility `chkpnt`. This checkpointing utility allows you to save the state of a running process (hierarchy) to a checkpoint file. The `restart` command, discussed in the next chapter, is used to continue the execution of the process (hierarchy) based on the checkpoint file. Conceptual information about how checkpointing and restarting works is in the chapter "How Checkpoint Restart works."

Checkpoint Restart user-level man pages, which contain command descriptions, syntax, and options, are:

chkpnt(1)	restart (1)	rmckpt(3)
chkpnt(3)	restart(3)	
cnx_chkpnt(3)	cnx_restart(3)	

To read any of these man pages, enter

% man *section_number man_page_name*

at the SPP-UX system prompt. For example, entering

```
% man 3 cnx_restart
```

returns the cnx_restart(3) man page.

Using the `chkpnt` command

To checkpoint processes, enter the `chkpnt` command at the SPP-UX shell prompt. The `chkpnt` command has the following format:

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

where

pid

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

Summary of `chkpnt` options

Table 1 summarizes the meanings of `chkpnt` options. They are described in greater detail in the section "Using the chkpnt options" on page 30. You can specify options individually or in combination with other options.

The options shown in Table 1 do not require arguments.

Table 1 `chkpnt` options that do not require arguments

`-C`	Copy all open regular files to the checkpoint directory; use `restart` with its `-C` option to use these files when restarting the process (hierarchy).
`-F`	Force checkpointing despite error conditions.
`-i`	Invoke `chkpnt`'s interactive mode.
`-n`	Perform checkpointability tests on target, but do not checkpoint.
`-q`	(Quiet mode) Suppress warning messages; error messages are not suppressed.
`-v`	Produce verbose output.
`-X`	Print debugging output.
`-r`	Perform recursive checkpoint; checkpoint entire process hierarchy beginning at process given by *pid*.
`-j`	Perform recursive checkpoint (same as `-r`).
`-p`	Do not perform recursive checkpoint (the default).

The options in Table 2 take the specified arguments

Table 2 `chkpnt` options that require arguments

`-d` *checkpoint_directory*	Create the checkpoint file in the directory specified by *checkpoint_directory*; the default is the current working directory.
`-f` *checkpoint_file*	Name the checkpoint file *checkpoint_file* (may be full pathname).
`-I` *logfile*	Use a logfile (created with the `-L` option) as command input for `chkpnt`
`-L` *logfile*	Invoke interactive mode of `chkpnt` and record actions in a file having the name *logfile*.
`-k` *signo*	Send the target process a signal (specified by *signo*) after checkpointing of the target process or process hierarchy has been completed.
`-K` *signo*	Like the `-k` *signo* option, but send signal (specified by *signo*) to every process in the hierarchy.

Using the chkpnt options

You can checkpoint a process by simply specifying the process identifier (PID) with the chkpnt command. To checkpoint a process with PID 7365, the simplest form of the command is:

```
% 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.

By default, the chkpnt utility saves the state of a process then allows the process to continue executing once checkpointing has been completed. Use the -K *signo* option to kill the target after checkpointing.

The chkpnt utility does not normally overwrite existing checkpoint files. If you intend to checkpoint the same process several times and you want to preserve the old checkpoint files, specify a new file name with the -f *checkpoint_file* option or put the files in a different directory by using the -d *checkpoint_directory* option.

Be careful of the force (-F) option; if you use it, chkpnt overwrites checkpoint files.

In addition to the simple form of the command, a number of options can be specified with chkpnt. The effects of these options are described below.

Saving open files (-C)

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

Use the `chkpnt -C` option to copy all regular files in use by the checkpointed process to the checkpoint directory (the current working directory or the directory specified by using `-d` *directory_name*). Files copied using the `-C` option have names of 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 open file being used by the checkpointed process

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 (-) is used instead of a file name)

Files that you copy to the checkpoint directory with the `-C` option are not 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 locations before restarting the process (making certain that no other process has modified the original files) using the `restart -C` option.

When you checkpoint applications that use random file access (such as database applications), 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 you do not use the `-C` option, these applications may not restart properly if the contents of the files are modified by the process after the checkpoint.

The following is an example of chkpnt using the -C option to store all open regular files in the current working directory:

```
% chkpnt -C 7365
```

Forcing a checkpoint (-F)

The -F option forces checkpointing despite error conditions. When you specify this option, chkpnt attempts to complete checkpointing even though error conditions occur that would otherwise cause checkpointing to be aborted.

A checkpoint file created with this option may not restart correctly. Use this option only when the error conditions reported by chkpnt are not essential to correct execution of the target process (for instance, an unused pipe to a process outside of the restarted hierarchy).

Caution

Do not use -F if you want to preserve existing checkpoint files. If you previously checkpointed a process and want to preserve the old checkpoint file, specify a new file name or directory using -f *checkpoint_file* or -d *checkpoint_directory*.

Checkpointing interactively (-i)

Use the -i option to invoke checkpointing in interactive mode. In interactive mode you can make decisions about each open file descriptor and each process of a hierarchy being checkpointed. In particular, it allows you to choose which open files to copy to the checkpoint directory (as opposed to the -C option, which automatically copies all regular files open to the target process to the checkpoint directory).

Refer to the section "Using chkpnt in interactive mode" on page 38 for more information about -i.

Determining a process's checkpointability (-n)

To determine if a target process will checkpoint safely—without actually checkpointing the process—use the -n option. This causes chkpnt to perform all the checkpointability tests on the target that are usually performed, but not write a checkpoint file. All error and warning messages that chkpnt would normally output are displayed.

The following example tests the checkpointability of process 345, invokes the interactive mode of chkpnt, and writes a checkpoint logfile called checkit:

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

Running chkpnt in quiet mode (-q)

Use the -q option to invoke chkpnt in quiet mode. In this mode, chkpnt suppresses warning messages. Fatal error messages are still generated. Use -q when you use -F to force checkpointing. A forced checkpoint results in many warning messages; using -q prevents those messages from being displayed.

The following example checkpoints process 7519 in quiet mode:

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

Running chkpnt in verbose mode (-v)

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

```
% chkpnt -v 1232
```

Producing debugging output (-x)

Use the -x option to print debugging output.

Checkpointing a process hierarchy (-j or -r)

Use the -j option to checkpoint an entire process (job) hierarchy beginning with the process specified by PID. This is the same as using the -r option.

If neither -r nor -j are specified, only the process specified by *pid* is checkpointed. The following example recursively checkpoints all processes descended from process 1232:

```
% chkpnt -r 1232
```

See "Checkpointing a process hierarchy in interactive mode" on page 45 for an interactive mode example.

Checkpointing a single process (-p)

Use the -p option to checkpoint only the process specified by *pid* (this is the default).

Specifying the checkpoint directory (-d *checkpoint_directory*)

Use the -d *checkpoint_directory* option to name the directory in which the checkpoint file is to be created; this directory must exist before you use this option. The default directory is the current working directory.

checkpoint_directory may be either a relative or absolute pathname. 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
```

Specifying the checkpoint file name (-f *checkpoint_file*)

Specify the name of the checkpoint file with the -f *checkpoint_file* option. If no checkpoint file name is specified, the default name is used. Refer to "Checkpoint file name" on page 14 for information about the naming conventions. If you checkpoint a process hierarchy and specify -f *checkpoint_file*, then *checkpoint_file* is used as the first element of the checkpoint file name for each process.

Use -f *checkpoint_file* to specify a new file name for previously checkpointed processes.

If the name you specify with -f contains no slashes (in other words, is only a file name) and you do not use the -d *checkpoint_directory*, the checkpoint file is placed in the current working directory. The following example creates a checkpoint file called simul1 in the current working directory:

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

If you specify a directory with -d *checkpoint_directory*, the file is placed in that directory. This 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 *checkpoint_file* option, thus making -d *checkpoint_directory* redundant. This example causes 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` *checkpoint_file*. 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` *checkpoint_file* contains slashes, the `-d` *checkpoint_directory* option is redundant. If you specify `-d` *checkpoint_directory* along with `-f` *checkpoint_file*, the path specified by both must be the same, or `chkpnt` fails. The following example is wrong, and causes failure:

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

Checkpointing from a logfile (`-I` *logfile*)

Use the `-I` *logfile* option to use the checkpoint options specified in the file *logfile*. You must first create the logfile using the `-L` *logfile* option that is discussed in the next section.

Checkpointing interactively with a logfile (`-L` *logfile*)

Using the `-L` *logfile* option invokes the interactive mode of `chkpnt` (refer to the description of `-i` above) and records the session in the file *logfile*.

You can later "play back" the session using the `-I` *logfile* option; all the checkpoint options you specify during the session using `-L` *logfile* are repeated when `chkpnt` is later invoked with `-I` *logfile*. You can give either a relative or an absolute pathname; if you specify a relative pathname, the file is created in the checkpoint directory (the current working directory or the directory specified using `chkpnt`'s `-d` *checkpoint_directory* option).

The following command 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 logfile created by a previous interactive `chkpnt` session that used the `-L` *logfile* option (refer to the description of `-L` *logfile* above), invoke `chkpnt` with the `-I` *logfile* option. The selections that were made during the previous session are used for the current one; no further interactive input is accepted.

In the following example, the logfile called mylog is used to control the checkpoint of process 3448:

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

Refer to the section titled "Creating and using checkpoint logfiles in interactive mode" on page 47 for more information about using logfiles.

Sending a signal to the target process (-k *signo*)

The -k *signo* option sends the signal specified by *signo* to the target process when the checkpoint has been successfully completed. If the checkpoint fails, no signal is sent. You can specify either the signal name (for example, KILL or SIGKILL) or the signal number. See the signal(5) man page for a list of signal names and numbers.

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

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

The next example also sends a HUP signal to process 9778, using the signal number (01) instead of the signal name:

```
% chkpnt -k 01 9778
```

With -k *signo*, 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 *signo*.

Sending a signal to the target process hierarchy (-K *signo*)

The -K *signo* option works just like -k *signo*, 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
```

Sample checkpoint sessions

There are two ways to use the chkpnt utility:

- As a single command issued from the shell command prompt
- Interactively

The following examples illustrate various options of chkpnt. The next section, "Using chkpnt in shell command-line mode," gives an example of the noninteractive (shell command line) mode of chkpnt. The section titled "Using chkpnt in interactive mode" on page 38 gives examples of interactive checkpointing.

Using chkpnt in shell command-line mode

To use chkpnt from the shell command line, enter chkpnt followed by any desired options, then the PID of the process you want to checkpoint. No further input is allowed by chkpnt; the shell prompt returns after checkpointing is complete.

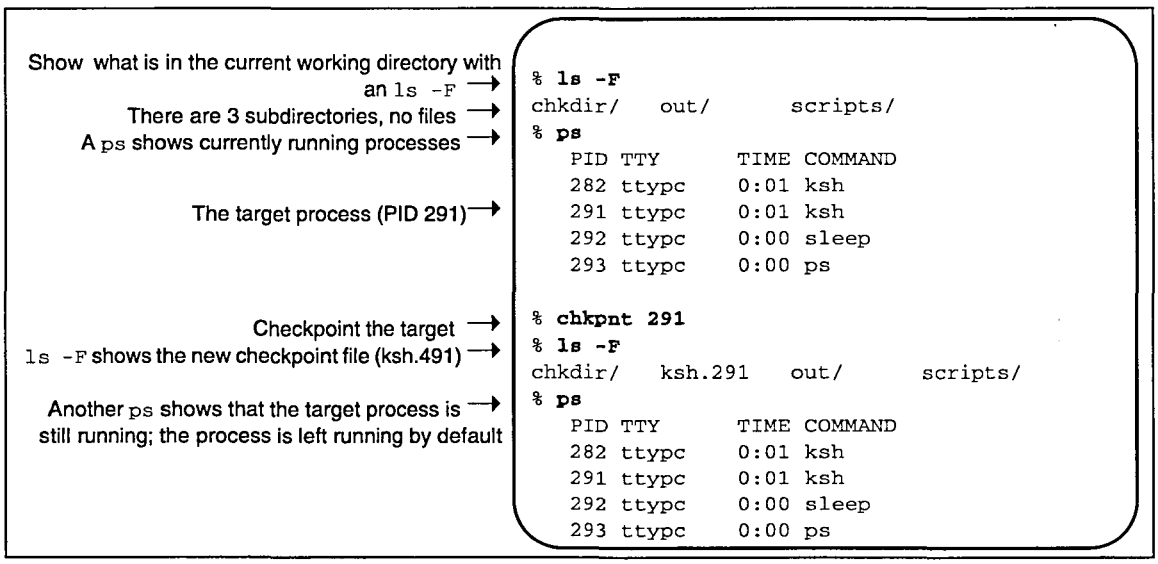


Figure 5 Checkpointing from the shell command line

Figure 5 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 291 is the target process. After checkpointing has been completed, the checkpoint file ksh.291 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.

Figure 6 shows how you can specify a checkpoint directory and a file name.

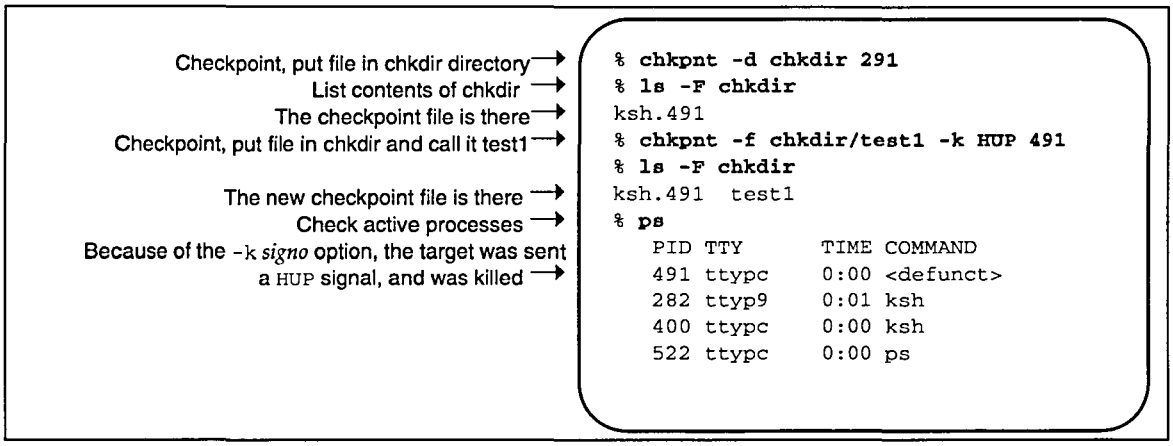


Figure 6 Specifying checkpoint directory and file name

The last `chkpnt` command in Figure 6 included a `-k` *signo* option that caused a `HUP` signal to be sent to the target process when checkpointing was completed. This signal caused the process to exit.

Using `chkpnt` in interactive mode

You invoke the interactive mode by using the `-i` option with `chkpnt`. Normally, all checkpoint options must be entered on the command line with `chkpnt`. In interactive mode, you are prompted to make the following decisions:

- Which processes should be checkpointed (in process hielorarchies)
- Which file descriptors should be saved in the checkpoint file
- Which files should be copied to the checkpoint directory

This is the only way you can copy a portion of the regular files open to the target process; the `-C` option automatically copies *all* open regular files to the checkpoint directory (the current working directory or the directory specified by the `-d` *checkpoint_directory* option).

Invoking chkpnt in interactive mode

In Figure 7, 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.

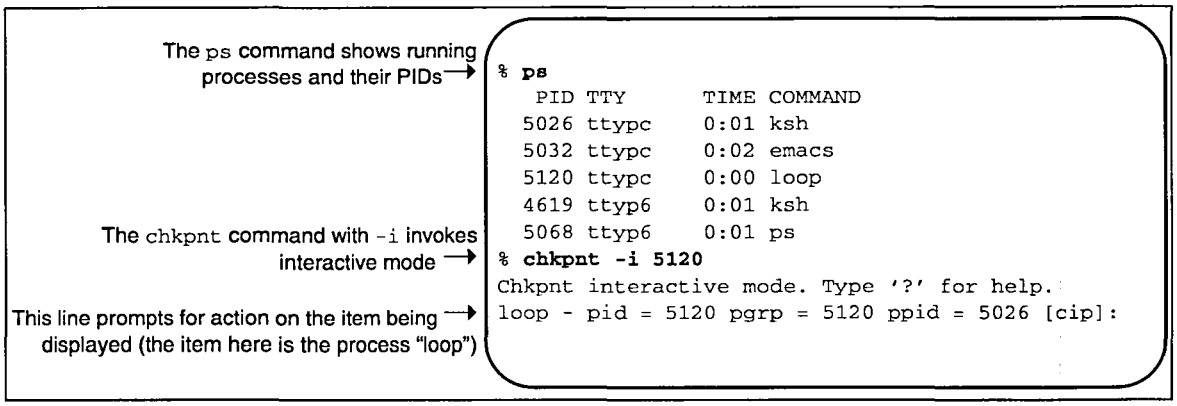


Figure 7 Invoking chkpnt in interactive mode

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 8.) Some of the legal commands for each prompt are shown in square brackets just before the colon.

The prompt shows information about the process or file descriptor that you can act 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, CR displays the next item. When you have taken action on the last item, the interactive session ends, and chkpnt writes the checkpoint files (one checkpoint file per process).

If you quit before this by responding Q to any prompt, no checkpoint files are written. After chkpnt has executed, the shell prompt returns.

chkpnt interactive-mode commands for processes

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 8.

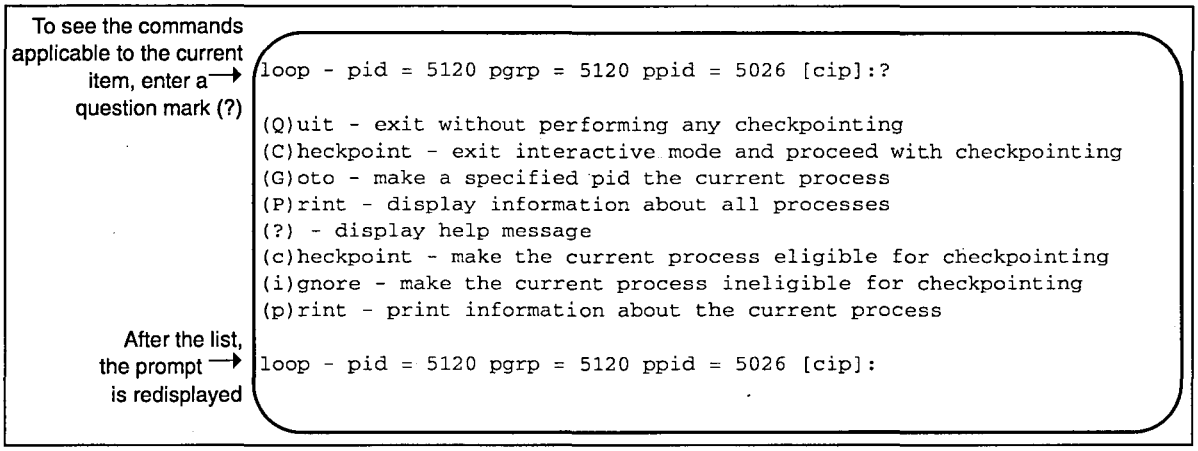


Figure 8 Interactive `chkpnt` commands for processes

As shown in Figure 8, the commands in the following table are available in the interactive mode of `chkpnt` when the current item is a process.

Table 3 `chkpnt` interactive-mode commands for processes

`Q`	Quit the interactive `chkpnt` session without performing any checkpointing; this cancels everything that you have done during this interactive session. This command can be used in response to any prompt at any time during the session.
`C`	Exit the interactive session and proceed with checkpointing. The selections that you have made up to this point are acted upon.
`G`	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 becomes the current item.
`P`	Show information about all processes in the hierarchy. (Information includes identifiers in the prompt line and all file descriptors open to the process.)
`?`	Display this list.
`c`	Checkpoint the process shown in the prompt. The interactive session continues and the next prompt is displayed, unless you have responded to all prompts. The actual checkpoint processing is done only when the interactive session has ended.
`i`	Ignore (do not checkpoint) the process shown in the prompt; you may still elect to checkpoint other processes in the hierarchy. Selectively checkpointing and restarting members of a process hierarchy (that is, restarting some and not others) has unpredictable consequences. Some restarted applications may not work properly if you selectively restart processes.
`p`	Show information about the process displayed in the prompt. This is like `P`, except that information is shown only for the process displayed in the prompt.

`chkpnt` interactive-mode commands for file descriptors

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. Enter a question mark (`?`) in response to a file descriptor prompt to see a list of legal commands for file descriptors.

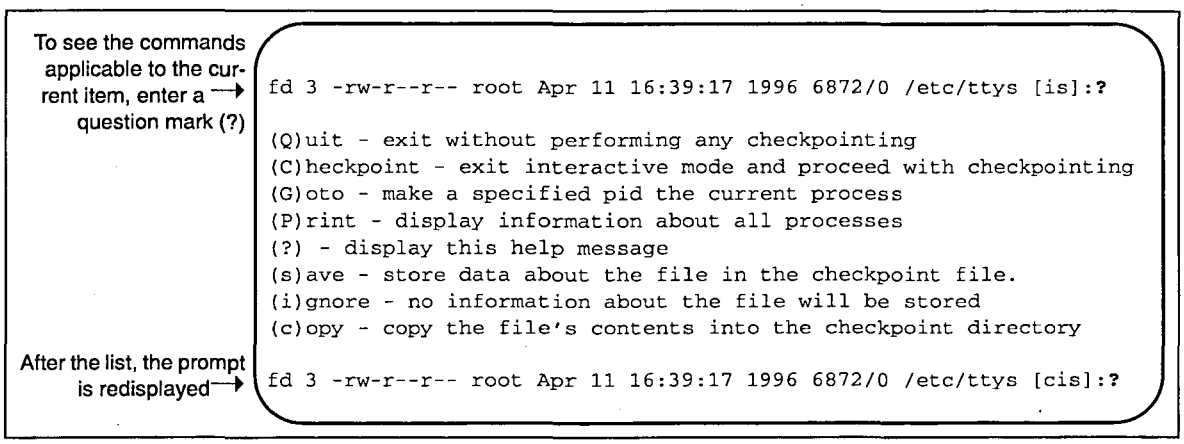


Figure 9 Interactive `chkpnt` commands for file descriptors

As shown in Figure 9, the following commands are available in the interactive mode of `chkpnt` for file descriptors:

Table 4 `chkpnt` interactive-mode commands for file descriptors

`Q`	Quit the interactive `chkpnt` session without performing any checkpointing; this cancels everything that you have done during this interactive session. This command can be used in response to any prompt at any time during the session.
`C`	Exit the interactive session and proceed with checkpointing. The selections that you have made up to this point are acted upon.
`G`	Go directly to a specific 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 becomes the current item.
`P`	Show information about all processes in the hierarchy. Information includes the identifiers in the prompt line and all file descriptors open to the process.
`p`	Show a list of open file descriptors for a specific process in the hierarchy.
`?`	Display this list.
`c`	Copy the entire file to the checkpoint directory. This command is not available if the current item is not a regular file.
`i`	Ignore. No information about the file is stored.
`s`	Store data about the file in the checkpoint file. Only file descriptor information—and not data—is saved.

Printing information about processes during checkpoint

When a process is the current item in `chkpnt`'s interactive mode, you can see a list of open file descriptors for that particular process by entering `p` (lowercase) as in Figure 10.

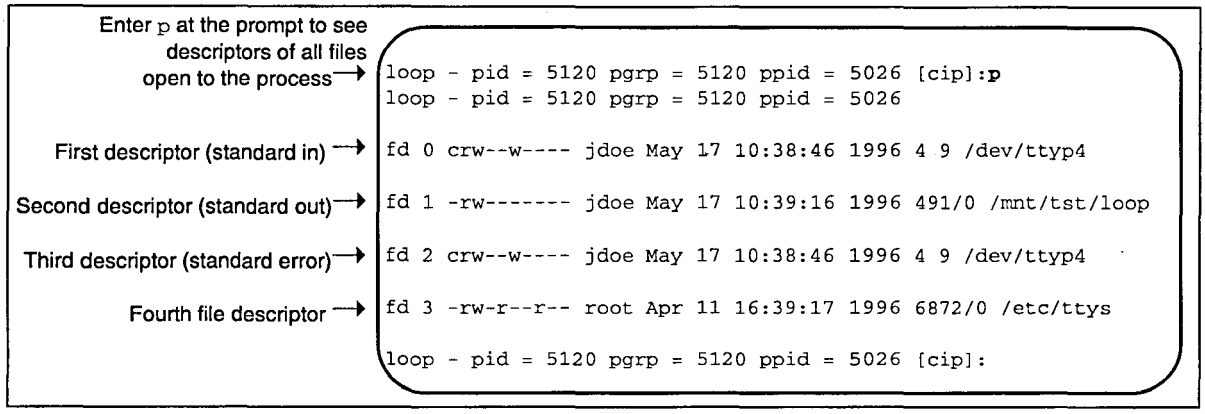


Figure 10 Print information about processes

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 are shown.

In addition to the file descriptor number, access privileges, owner, and file name, 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 11 shows the information contained in each file descriptor entry.

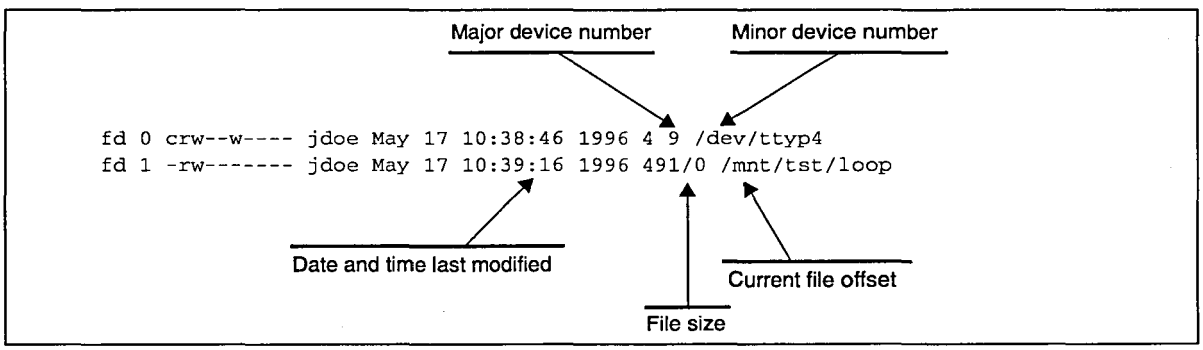


Figure 11 File descriptor information

Checkpointing a process hierarchy in interactive mode

This section provides you a step-by-step example of checkpointing a process hierarchy in interactive mode.

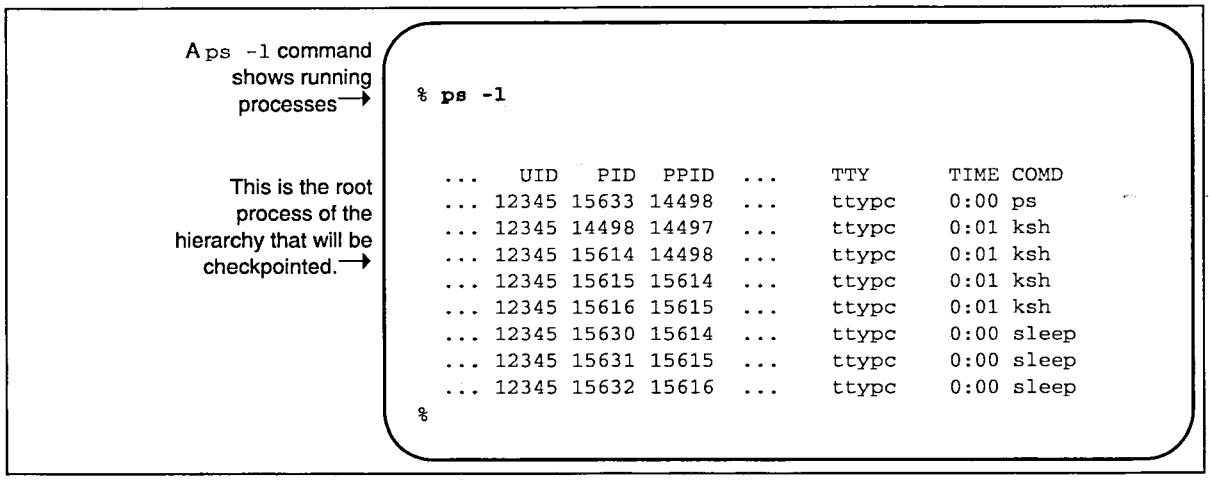


Figure 12 What processes are running?

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 `ksh` process that has a PID of 15614. To checkpoint this hierarchy interactively, you must use the `-r` (recursively checkpoint) and `-i` (interactive mode) options. As shown in Figure 13, a `P` (uppercase) command will display information about the whole hierarchy.

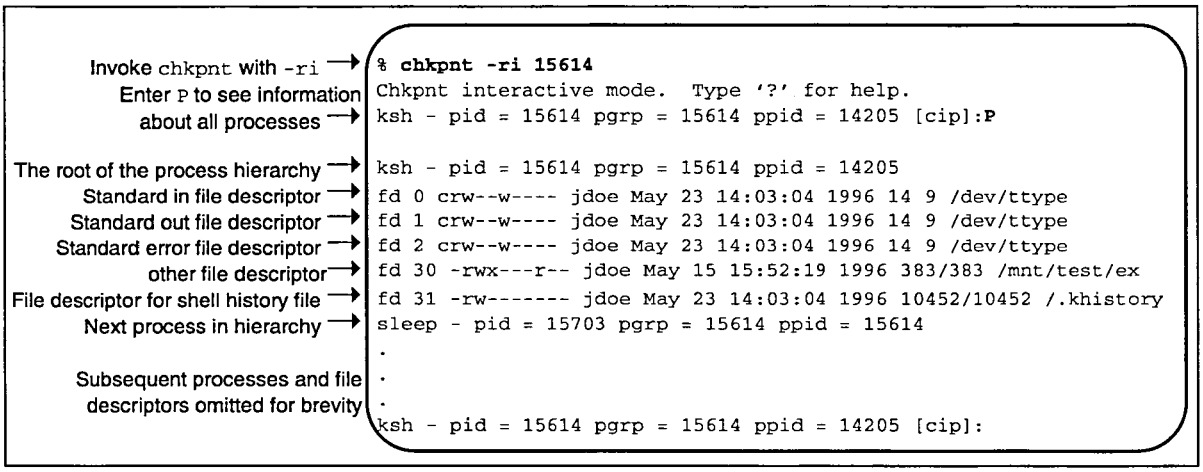


Figure 13 List the target hierarchy

As shown in Figure 13, the root process (15614) has five open file descriptors. The other members of the process hierarchy are not shown. In Figure 14, 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.

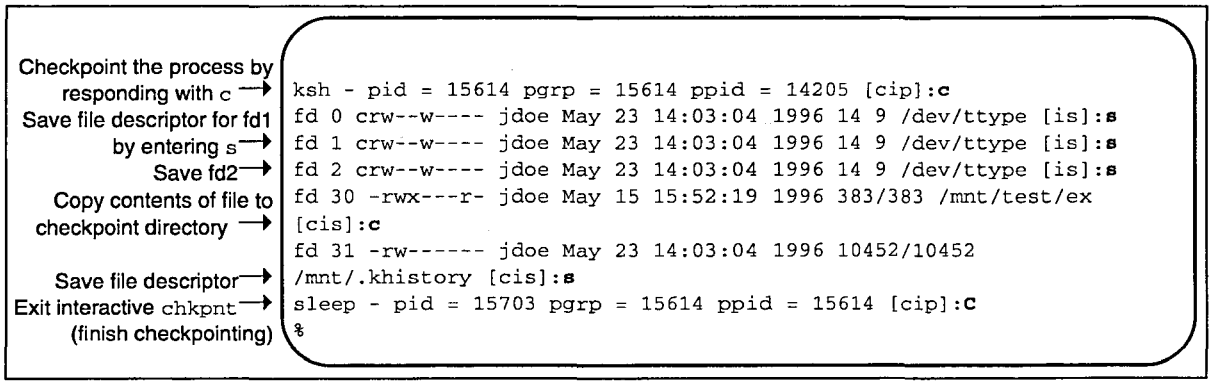


Figure 14 Checkpoint the process

After you enter C (uppercase), the checkpointing operation continues without user intervention. When chkpnt is finished, the shell prompt returns.

Creating and using checkpoint logfiles in interactive mode

If you want to checkpoint the same process multiple times, you need only checkpoint it interactively once if you use `-L` to create a logfile.

The logfile contains information about the decisions you made during the interactive session. You can "play back" this session by using this logfile as input by specifying the `-I` *logfile* option with a command-line mode `chkpnt` command, as shown in Figure 16.

Checkpoint the process interactively and create logfile called "mylog" →

Examine contents of logfile →

```
% chkpnt -L mylog 650
Chkpnt interactive mode.  Type '?' for help.
ksh - pid = 650 pgrp = 650 ppid = 644 [cip]:c
fd 0 crw--w---- jdoe May 27 16:37:55 1996 0 9 /dev/ttyp0 [is]:s
fd 1 crw--w---- jdoe May 27 16:37:55 1996 0 9 /dev/ttyp0 [is]:s
fd 2 crw--w---- jdoe May 27 16:37:55 1996 0 9 /dev/ttyp0 [is]:s
fd 3 -rw-r--r-- root Apr 11 16:39:17 1996 6872/0 /etc/ttys [cis]:s
fd 30 -rwx------ jdoe May 27 15:17:19 1996 383/383 /mnt/ex3 [cis]:c

% cat mylog
650 ksh checkpoint
0 :/dev/ttyp0: path
1 :/dev/ttyp0: path
2 :/dev/ttyp0: path
3 :/etc/ttys: path
30 :/mnt/jdoe/bin/victim: copy
31 :/mnt/jdoe/.khistory: path
%
```

Figure 15 Creating a checkpoint logfile

The logfile mylog contains information you can use to cause subsequent invocations of chkpnt to take the same actions as the first time. Figure 16 shows how subsequent chkpnt commands can use the logfile.

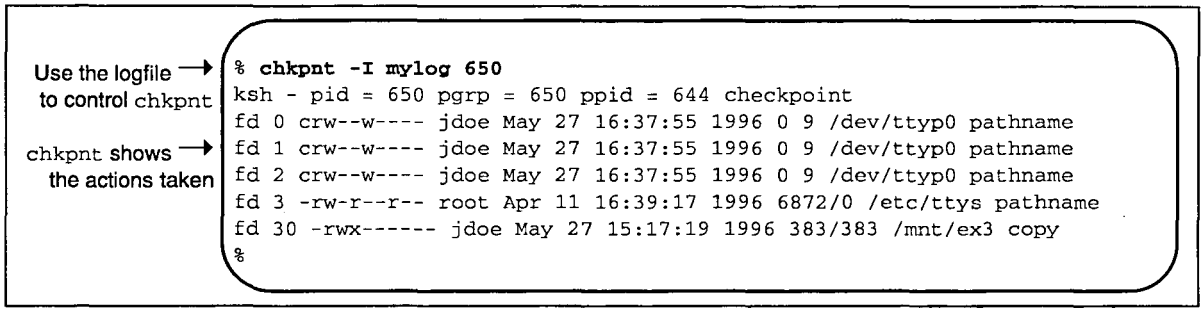


Figure 16 Using a checkpoint logfile

Caution

Do not use logfiles if your processes use files that have names containing colons. The checkpoint logfile format uses colons as delimiters; file names containing colons will cause errors.

Restarting processes

3

This chapter discusses the SPP-UX command-line utility `restart`. Use this utility to restart process (hierarchy) that was checkpointed by the `chkpnt` command discussed in the previous chapter. Conceptual information about how checkpointing and restarting works is in the chapter "How Checkpoint Restart works."

CR user-level man pages, which contain command descriptions, syntax, and options, are:

chkpnt(1)	restart (1)	rmckpt(3)
chkpnt(3)	restart(3)	
cnx_chkpnt(3)	cnx_restart(3)	

To read any of these man pages, enter

% man *section_number man_page_name*

at the SPP-UX system prompt. For example, entering

```
% man 3 cnx_restart
```

returns the cnx_restart(3) man page.

Using the `restart` command

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

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

where

checkpoint_file

is the name of the checkpoint file that was created for the process to be restarted.

Summary of `restart` options

Table 5 summarizes the meanings of the `restart` options. They are described in greater detail in the section "Using the restart options" . You can specify options individually or in combination with other options.

The options in Table 5 do not require arguments.

Table 5 `restart` options that do not require arguments

`-C`	Copy files back into same directories as at checkpoint; this option requires that you had used the `-C` option to `chkpnt` when you checkpointed the process (hierarchy)
`-F`	Force restart despite error conditions
`-i`	Invoke `restart`'s interactive mode
`-q`	(Quiet mode) Suppress warning messages; error messages are not suppressed
`-v`	Produce verbose output
`-w`	Wait for restarted process to complete (this is the default behavior; the option is supplied for compatibility with other implementations)
`-W`	Do not wait for restarted process to complete
`-X`	Print debugging output
`-z`	Restart the process in a stopped state

The options in Table 6 take the specified arguments.

Table 6 `restart` options that require arguments

`-k` *signo*	Send the target process a signal (specified by *signo)* when restart is complete
`-K` *signo*	Like the `-k` *signo* option, but send signal to every member of the process hierarchy rooted at the target process

Using the `restart` options

Although a large number of options are available, 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, the simplest form of the command is:

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

The process about which information was stored in this file will now be continued 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` option of `chkpnt`), the entire process hierarchy is restarted. Alternatively, you can selectively restart processes in the hierarchy as discussed in "Restarting a process hierarchy in interactive mode" .

Unless the `restart` command was issued in background mode (by ending the command with the & character), the shell from which the command was issued is suspended until the restarted processes have exited.

You can kill and restart a process from the same checkpoint file as often as desired.

In addition to this simple form of the command, a number of options can be specified with `restart`. Descriptions of these options are given below.

Copying files back to their checkpoint locations (-C)

If the files open to the target process were copied to the checkpoint directory by specifying -C (uppercase) with the chkpnt command), then you must copy them back from the checkpoint directory to their original locations by specifying -C with restart.

Caution

Do not use -C unless you are prepared to possibly lose files. This option causes restart to 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.

Forcing restart (-F)

Use the -F option to force restarting despite error conditions.

If you specify this option, the utility attempts 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
```

Restarting in interactive mode (-i)

To restart a process in the interactive restart mode, use the -i option. Like its counterpart in chkpnt, it 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 the section "Using restart in interactive mode" for more information on using this mode.

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

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

Running restart in quiet mode (-q)

Use the -q option to invoke restart in quiet mode. In this mode, restart suppresses warning messages. Fatal error messages are still generated.

Running `restart` in verbose mode (`-v`)

The `-v` option generates verbose output; it gives additional information during restarting.

Waiting for the target process to exit (`-w`)

Using the `-w` option causes `restart` to wait for the target process to exit; because this is the default, the `-w` option 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.)

Not waiting for the target process to exit (`-W`)

Using the `-W` option causes `restart` not to wait for the target process to exit. If you specify this option, 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 option is only for noninteractive invocation of `restart` (do not use with `-i`). Do not specify it from an interactive shell. Because `restart` changes its PID when `-W` is used, the shell becomes "confused" and remains suspended even after the target process exits.

Producing debugging output (`-x`)

Use `-X` to print debugging information.

Restarting a process in a stopped state (`-z`)

Use `-z` to 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` *signo* or `-K` *signo*.

Sending a signal to the target process (`-k` *signo*)

To send the signal specified by *signo* to the target process (in other words, the process being restarted) when the restart is complete, use the `restart -k` *signo* command.

By default, a `SIGCONT` signal is sent to the target process when restart is complete; use `-k` *signo* to override this default. Specify

the signal by either signal name (for example, CONT) or by number. If the specified signal is zero (-k 0), no signal is sent. See the signal(5) man page for a listing of signal names and numbers.

Sending a signal to the target process hierarchy (-K *signo*)

If you are restarting a process hierarchy, the entire hierarchy is 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 *signo*. (See the signal(5) man page for a listing of signal names and numbers.)

Using diagnostics (-Fqv)

Use the -F (force restart) and -v (verbose mode) options to diagnose restart problems. These options cause `restart` to output extra information that may help you.

When you use -F, use the -q option (for quiet mode) to suppress warning messages so you will only get error messages.

The -v option provides verbose output giving 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
```

Sample restart sessions

There are two ways to use `restart`:

- As a single command issued from the shell prompt
- Interactively

The following examples illustrate the use of various options of `restart`. The next section, "Using restart in shell command-line mode," gives an example of the noninteractive (shell command line) mode of `restart`. "Using restart in interactive mode" shows examples of `restart`'s interactive mode.

Using `restart` in shell command-line mode

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

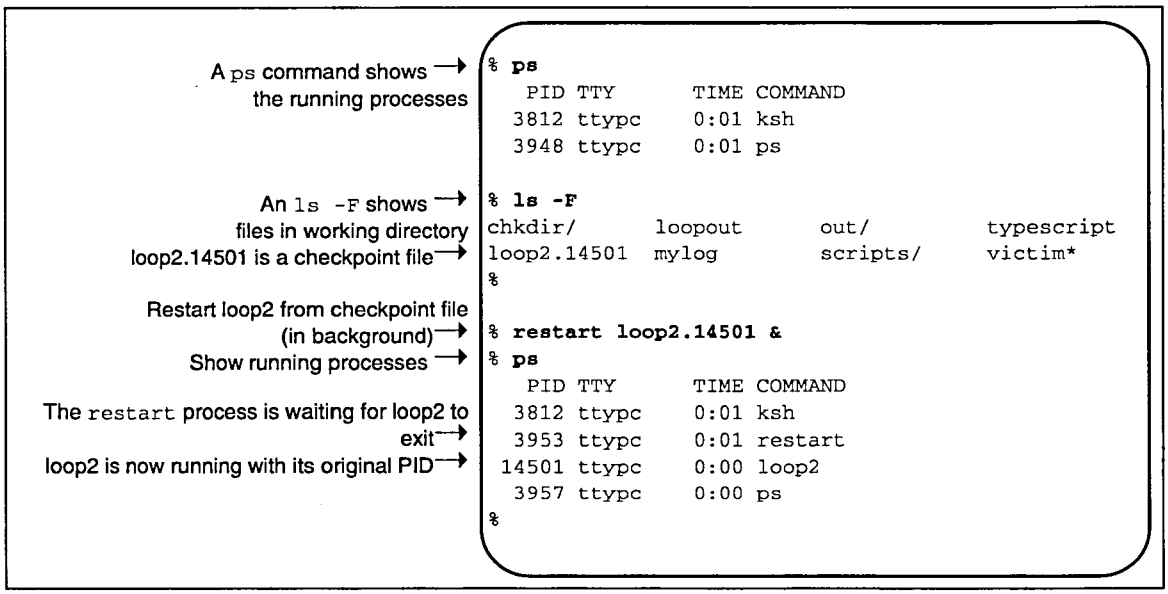


Figure 17 Restarting from the shell command line

Figure 17 shows a simple restart example. The `ps` command shows the running processes to be a `ksh` session and the `ps` command itself.

Performing 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, performing 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 ampersand (&)—then the shell prompt would not have returned until `loop2` exited.

Using `restart` in interactive mode

Invoke this mode by using the `-i` option with `restart`. Normally, all options must be entered on the command line with `restart`. In interactive mode, however, you are 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 18 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.

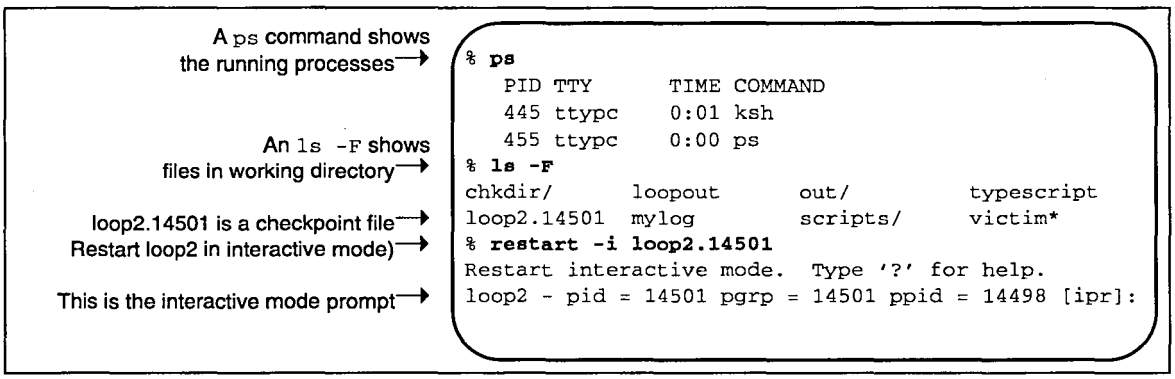


Figure 18 Invoking `restart` in interactive mode

As in interactive `chkpnt` mode, the last line in Figure 18 is a prompt, and shows the "current item" (that is, 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 19. Some commands that are legal for each prompt are shown in square brackets just before the colon.

`restart` interactive-mode commands for processes

To display a list of all legal commands for the current item, enter a question mark after the prompt. Figure 19 shows the legal commands for processes.

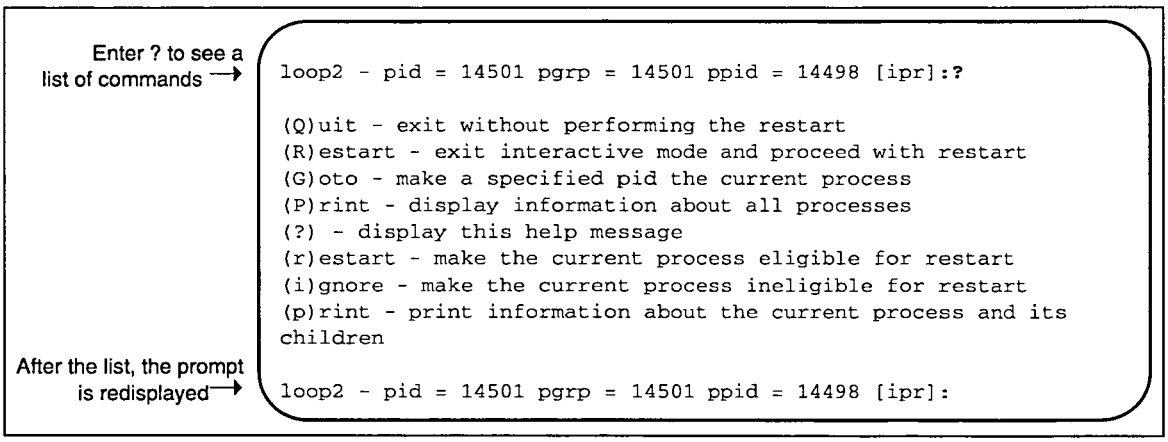


Figure 19 Interactive `restart` commands for processes

The commands shown in Table 7 are available in the interactive mode of `restart` when the current item is a process.

Table 7 `restart` interactive-mode commands for processes

`Q`	Quit the interactive `restart` session without restarting anything; this cancels everything you have done during the current interactive session. You can issue this command at any time in response to any prompt.
`R`	Exit the interactive session and proceed with restarting. The selections you have made up to this point will be acted upon.
`G`	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.
`P`	Show information about all processes in the hierarchy (including file descriptors that were open to the process).
`?`	Display this list.
`i`	Ignore the current process when restarting. If you enter `i`, the process shown in the prompt is not restarted; you may still elect to restart other processes in the hierarchy. Selectively restarting members of a process hierarchy (that is, restarting some and not others) has unpredictable consequences. Some restarted applications may not work properly if you perform selective restarting.
`p`	Print information for the current process and its children. This is like `P`, except that information is shown only for the process displayed in the prompt.
`r`	Restart the process shown in the prompt. The interactive session continues and the next prompt is displayed, unless you have responded to all prompts. Restarting is done only when the interactive session has ended.

`restart` interactive-mode commands for file descriptors

The commands in `restart`'s interactive-mode differ for a file descriptor. When the current item is a file descriptor (not a process), there are additional commands; some of the commands have a different meaning, and some are not available.

Enter a question mark (?) in response to a file descriptor prompt to see a list of legal commands for file descriptors, as shown in Figure 20.

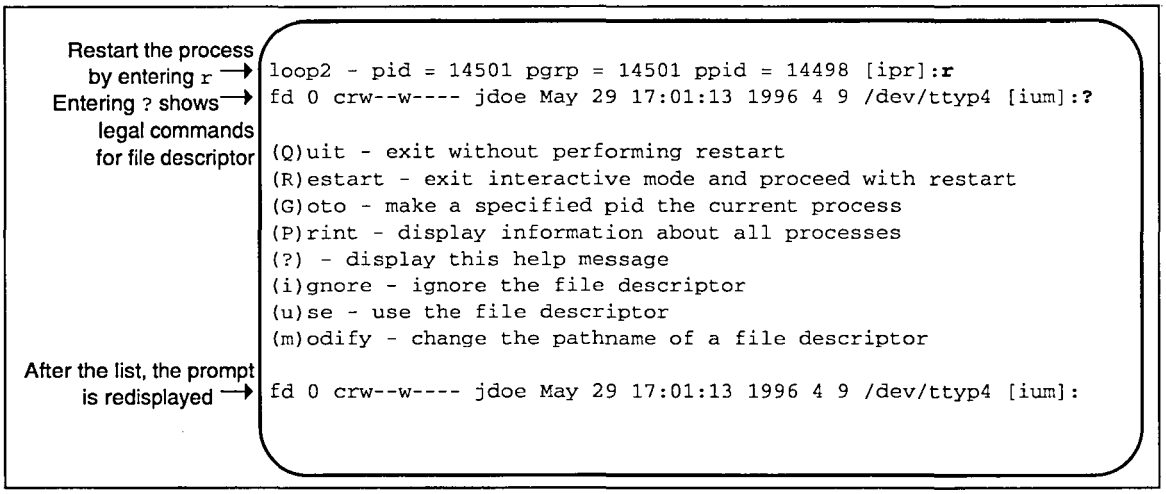


Figure 20 Interactive `restart` commands for file descriptors

The commands in Table 8 are available for file descriptors in interactive `restart`.

Table 8 `restart` interactive-mode commands for file descriptors

`Q`	Quit the interactive `restart` session without restarting anything; this cancels everything you have done during the current interactive session. You can issue this command at any time in response to any prompt.
`R`	Exit the interactive session and proceed with restarting. The selections that you have made up to this point will be acted upon.
`G`	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.
`P`	Show information about all processes in the hierarchy (including file descriptors that were open to the process).
`?`	Display this list.
`i`	Ignore (do not use) this file descriptor when restarting the process. If the restarted process later tries to use the ignored file descriptor, file access will fail, and an EBADF error message will be returned to the process.
`u`	Use this file descriptor as it is.
`m`	Modify the current 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 modify the file descriptor so that it points to the file /mnt/user/old.data1 by entering: **`m /mnt/user/old.data1`** Changing the path referenced by a descriptor will affect any shared file descriptors created with a `dup` system call or that are shared across an `exec()` system call.

Printing information about processes during restart

When a process is the current item, you can see a list of open file descriptors for that particular process by entering a lower case p. If you enter an upper case P, and if a process hierarchy is being restarted, all the processes and their file descriptors in the hierarchy are shown.

This works like the corresponding features in interactive chkpnt. Refer to the section titled, "Printing information about processes during checkpoint" .

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

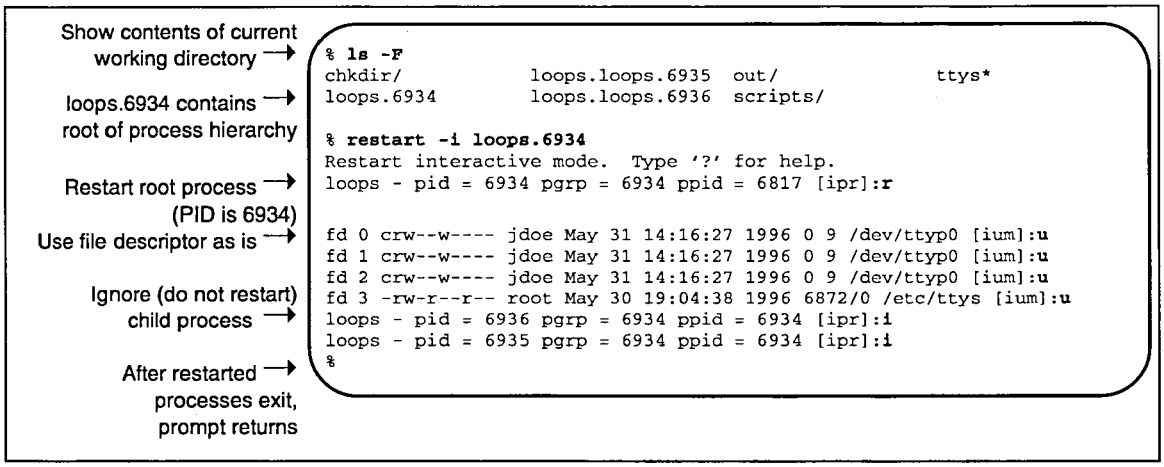


Figure 21 Interactive `restart`

In Figure 21, the first (root) process is restarted by responding with r (restart current process) to the prompt. The file descriptors for that process are then displayed; because the response to each prompt was u (use current file descriptor as is), they are all used.

Because the response to the prompt for each of the two child processes was `i` (ignore current item when restarting), these processes are not restarted. After the last prompt is answered, the process is restarted. When the restarted process exits, the shell prompt returns.

Selectively checkpointing and restarting members of a process hierarchy (that is, restarting some processes and not others) has unpredictable consequences. Some restarted applications may not work properly if you perform selective checkpointing and restarting.

Programming in C and C++ with Checkpoint Restart

4

This chapter:

- Describes the library functions you can use in C and C++ to build Checkpoint Restart capability into applications
- Discusses related programming considerations
- Provides code samples

For additional information about the library functions, refer to the following man pages:

- chkpnt(3)
- cnx_chkpnt(3)
- restart(3)
- cnx_restart(3)
- rmckpt(3)

Conceptual information about how CR works is located in Chapter 1, "How Checkpoint Restart works."

Using checkpoint in C and C++

The C library functions used to checkpoint processes are

- `chkpnt()`
- `cnx_chkpnt()`

Use and parameters of `chkpnt()` and `cnx_chkpnt()` are described below. The functions are also described in the chkpnt(3) and cnx_chkpnt(3) man pages.

The `chkpnt()` and `cnx_chkpnt()` functions invoke the SPP-UX `chkpnt` utility at run time; the utility is then used to do the actual checkpointing. The utility does not need to be in the default search path of the `chkpnt()` process when it is invoked;it is automatically sought in its default directory. `chkpnt` must reside in the default directory in which it was placed by the installation script (/usr/bin/chkpnt). If `chroot` is used to change the root directory, /usr/bin/chkpnt must exist at the new root.

The `chkpnt()` and `cnx_chkpnt()` functions cannot be called from within a multithreaded region of a program.

To compile with any `cnx` function, the user needs the library `libcnx_syscall.a`. The `chkpnt()` library call conforms to the POSIX API for the C language.

cnx_chkpnt() format and parameters

The cnx_chkpnt() function has the following format:

```
#include<sys/types.h>
#include<cnx_chkpnt.h>

cnx_chkpnt (int class, int pid, char *name, int options, int signo)
```

You can specify the following information and options in the parameters:

`int` *class*

where *class* is CHKPNT_PROC or CHKPNT_FAMILY

If you specify CHKPNT_PROC or 0 (zero), only the process indicated by *pid* is checkpointed.

If you specify CHKPNT_FAMILY, the entire process hierarchy (beginning with the process having the identifier specified in the *pid* parameter) is checkpointed.

`int` *pid*

where *pid* is the 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 cnx_chkpnt() call is itself checkpointed.

`char` **name*

where **name* gives the name of the checkpoint file

You must specify a relative or absolute pathname for the checkpoint file of the target process. If a process hierarchy is being checkpointed, this name is the checkpoint file of the root process. The maximum length of *name* is 16 characters, in addition to the null terminator.

`int` *options*

where *options* is 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 ordinarily fails. 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 process in the hierarchy, the target processes continue normal execution.

`CHKPNT_SIGFAMILY` (cannot use with `CHKPNT_SIGROOT`)

Send signal *signo* to the 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` (cannot use with `CHKPNT_SIGFAMILY`)

Send signal *signo* to the 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*

where *signo* is a signal to send to the process family (using `CHKPNT_SIGFAMILY`) or to the target process (using `CHKPNT_SIGROOT`)

The signal specified in this parameter is sent only if either the `CHKPNT_SIGFAMLY` option or the `CHKPNT_SIGROOT` option is specified in the *options* parameter. If either of these options is given, then *signo* must be specified. *signo* can be any of the signals described in the signal(5) man page.

`cnx_chkpnt()` return values and error codes

Recall that the `cnx_chkpnt()` function has the following format:

`cnx_chkpnt (int` *class*, `int` *pid*, `char` **name*, `int` *options*, `int` *signo*`)`

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

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

`EACCESS`

Search permission denied on a component of the path specified by the *name* parameter. Change the permissions or change the name.

`EEXIST`

The checkpoint file already exists. Delete the file or specify another name using the *name* parameter.

`EINTR`

The checkpointing operation was interrupted by a signal.

`EINVAL`

An invalid parameter was specified with the `cnx_chkpnt()` function call.

`ENAMETOOLONG`

The string specified for *name* is too long (that is, longer than PATH_MAX).

`ENOSPC`

There is no free space on the device that contains *name*.

`EPERM`

The process making the `cnx_chkpnt()` call does not have permission to checkpoint one or more of the target processes.

EPIPE

One or more 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.

chkpnt() format and parameters

The chkpnt() function has the following format:

```
#include<chkpnt.h>

chkpnt (int class, pid_t id, const char *path, int flags)
```

You can specify the following information in the parameters:

int *class*

where *class* is one of the values listed below:

CHKPNT_SESSION

Checkpoint the processes that are members of the current session.

CHKPNT_PGRP

Checkpoint the processes that are members of the current process group.

CHKPNT_PROC

Checkpoint a single target process.

int *id*

where *id* is the process identifier

The value of *id* should be the current PID or zero, which has the same meaning as the current PID. With appropriate privilege, the value of *id* may take on the value of the PID of a different process.

const char **path*

where **path* gives a checkpoint file path

The path argument specifies the name of the checkpoint directory to be created. The directory created will contain all the files created by chkpnt(1).

`int` *flags*

where *flags* is 0 (zero) or `CHKPNT_KILL`

If 0 is specified, no action is taken. If the `CHKPNT_KILL` flag is specified, all target processes are killed upon successful checkpoint. If `CHKPNT_KILL` is specified, and if `chkpnt()` fails for any reason, the target processes continue normal execution.

chkpnt() return values and error codes

Recall that `chkpnt()` has the following format:

`chkpnt (int` *class*`, pid_t` *id*`, const char *`*path*`, int` *flags*`)`

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

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

`EACCESS`

Search permission denied on a component of the path specified by *path* parameter. Change the permissions or change the name.

`ECHKPNTFAIL`

Checkpoint operation could not be completed or there is an unrecoverable resource associated with the target process.

`EEXIST`

The checkpoint file already exists. Delete the file or specify another name using the *path* parameter.

`EINVAL`

An invalid parameter was specified with the `chkpnt()` function call.

`ENOENT`

The *path* argument names a nonexistent directory or points to an empty string.

ENOTDIR

A component of the *path* is not a directory.

ENAMETOOLONG

The string specified for *path* is too long (that is, longer than PATH_MAX).

ENOSPC

There is no free space on the device that contains *path*.

EROFS

The file specified by *path* resides on a read-only file system.

Using restart in C and C++

Use the `restart()` and `cnx_restart()` C library functions to restart processes from checkpoint files. These functions are also described in the restart(3) and cnx_restart(3) man pages.

The `restart()` and `cnx_restart()` functions invoke the `restart` utility at run time; the utility is then used to do the actual restarting. Refer to the chapter, "Restarting processes,"for more information on the `restart` utility.

The utility does not need to be in the default search path of the `restart` utility when it is invoked—the utility is automatically sought in the directory in which it was invoked.

For the restart functions to work, the `restart()` utility must reside in the default directory in which it was placed by the installation script (/usr/bin/restart).

The `restart()` or `cnx_restart()` functions 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 are 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. You can modify this default using the *flags* parameter.

The `restart()` function conforms to the POSIX API for the C language.

cnx_restart() format and parameters

The cnx_restart() function has the following format:

```
#include<sys/types.h>
#include<cnx_chkpnt.h>

cnx_restart (char *path, int flags, int signo)
```

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

`char` **path*

where **path* gives the pathname of the checkpoint file

path gives the pathname of the checkpoint file from which the target process is 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*

where *flags* is 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
(cannot use with RESTART_SIGROOT)

Send signal *signo* to the process family

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

RESTART_SIGROOT
(cannot use with RESTART_SIGFAMILY)

Send signal *signo* to the target process

Send a signal only to the root process of the hierarchy being restarted when the entire hierarchy is completely restarted (no signal is sent if restarting fails).

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

RESTART_FORCE

Proceed despite error conditions

This option causes `cnx_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 ordinarily fails. If RESTART_FORCE is specified, the process is restarted, but its PID is different from what it was when the process was checkpointed.

Caution

Do not force a restart unless you are prepared for possible incorrect execution of the process. Restarting a process with a PID other than the one it had when it was checkpointed has unpredictable results.

Another example is a process that had a file open when it was checkpointed. If the file is not available when the process is restarted, `cnx_restart()` fails unless RESTART_FORCE is used. If you use this option, the process may run correctly—if it does not try to access the unavailable file.

RESTART_DEBUG

Restart in debug state

The process starts in debug state as though `exect()` had been called. A process restarted with this option can easily be placed under control of a debugger.

RESTART_SUSPEND

Restart in suspended state

The process restarts, but is left in a "stopped" state as though it had been sent a SIGSTOP signal.

`int` *signo*

where *signo* is a signal to send to the process family (using CHKPNT_SIGFAMILY) or to the target process (using CHKPNT_SIGROOT)

The signal specified in this parameter is sent if the RESTART_SIGFAMILY or RESTART_SIGROOT options are specified in the *flags* parameter. If either of these options is specified, *signo* must also be specified.

`cnx_restart()` return values and error codes

Recall that `cnx_restart()` has the following format:

`cnx_restart (char *`*`path`*`, int `*`flags`*`, int `*`signo`*`)`

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

`EACCESS`

Search permission denied on a component of the path specified by the *path* 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`

Invalid parameter was specified with the `cnx_restart()` function call.

`ENAMETOOLONG`

String specified by *path* is too long (that is, longer than PATH_MAX).

`ENOENT`

The checkpoint file *path* does not exist.

`restart()` format and parameters

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

```
#include<chkpnt.h>

restart (const char *path, int flags)
```

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

`const char *path`

where *path* gives the pathname of the checkpoint directory

path gives the pathname of the checkpoint directory from which the target process and associated files will be restarted.

`int flags`

where *flags* is one or more of the options listed below:

`RESTART_SIGALL`

Upon successful restart, a `SIGCONT` signal is sent to each of the processes.

`RESTART_SIGROOT`

Upon successful restart, a `SIGCONT` signal is sent to the eldest process in the restart group.

`restart()` return values and error codes

Recall that `restart()` has the following format:

`restart (const char *`*path*`, int` *flags*`)`

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

`EACCESS`

Search permission denied on a component of the path specified by the *path* 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.

`EBUSY`

The process ID of one or more of the processes to be restarted is currently in use in the system.

`EINVAL`

Invalid parameter was specified with the `restart()` function call.

`ENAMETOOLONG`

String specified by *path* is too long (that is, longer than PATH_MAX).

`ENOENT`

The checkpoint file given by the *path* parameter does not exist.

`ENOEXEC`

The named file does not exist, or the *path* argument points to an empty string.

`ENOTDIR`

A component of the *path* is not a directory.

`EPERM`

The calling process does not have permission to restart one or more of the target processes.

`ERESTARTFAIL`

The restart operation could not be completed.

Removing a checkpoint file in C and C++

Use the C library function rmckpt(3) to remove a checkpoint file created with the POSIX function, chkpnt(3). The function is also described in the rmckpt(3) man page.

`rmckpt()` format and parameters

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

```
#include<chkpnt.h>

rmckpt (const char *path)
```

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

`const char *path`

where *path* gives the pathname of checkpoint directory that will be removed.

`rmckpt()` return values and error codes

Upon successful completion, `rmckpt()` returns a zero. If `rmckpt()` fails, a -1 is returned. In case of failure, the external variable `errno` is set to one of the following error codes:

EACCESS

Search permission denied on a component of the path specified by the *path* parameter (change the permissions or change the name).

ENAMETOOLONG

String specified by *path* is too long (that is, longer than PATH_MAX).

ENOENT

The checkpoint file given by the *path* parameter does not exist.

ENOTDIR

A component of the *path* is not a directory.

EROFS

The checkpoint file to be removed resides on a read-only file system.

Programming guidelines

If you are writing an application that might get checkpointed, adhere to the following guidelines:

- Do not use any resources not supported by Checkpoint Restart. For example, do not use:
  - Sockets
  - File locks
  - Memory locks
  - Unsupported devices
- 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 fails if it is issued after the application is restarted, because `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. It is difficult for `chkpnt` to locate all the processes of the hierarchy when no top-level parent exists.

C programming example

The programming example below demonstrates a call to `cnx_chkpnt()` and to `cnx_restart()`. In it, 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()`.

Explanation of functions

This is what these functions do:

- `main()`
  - Makes a `sigblock()` 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 `fork()` returns a negative value, it failed; the program then prints an error message and exits.
  - If `fork()` returns 0, the process is the child; it then calls `do_child()`.
  - If `fork()` returns a nonzero, nonnegative value, this is the PID of the child (and the process is the parent); it then unblocks the `SIGCONT` signal and 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()` is called.
  - Prints a message: "Before checkpoint."
  - Makes a `sigpause()` call to suspend the process. While it is paused, the process is 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 is also called.
  - It then prints a message: "After restart." and exits.
- `do_parent()`
  - 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.
- `sighandler()` handles signals.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <cnx_chkpnt.h>
#include <sys/wait.h>

int
main(void)
{
    pid_t           pid;        /* Process-id of a forked child */
    void            do_child(void);
    void            do_parent(pid_t);
    long            oldmask;

    oldmask = 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(). */

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

    if (pid == 0) {
        do_child();
    } else {
        sigsetmask(oldmask);
        do_parent(pid);
    }

    return 0;
}
```

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

    signal(SIGCONT, sighandler);
    puts("Before checkpoint.");
    sigpause(0);                /* Wait for restart */
    puts("After restart.");}

/*
 * void do_parent(pid_t);
 *
 * Checkpoint and restart child process.
 */
void
do_parent(pid_t kid)
{
    pid_t           rpid;       /* PID of restarted proc */

    if (cnx_chkpnt(CHKPNT_PROC, kid, "ex1",
                   CHKPNT_SIGROOT | CHKPNT_FORCE, SIGKILL) < 0) {
        perror("chkpnt failed");
        /* Kill the child because the checkpoint failed */
        kill(kid, SIGKILL);
        exit(1);
    }

    /*
     * We just killed the child with the cnx_chkpnt(). We must
     * wait for it to exit.
     */
    if (waitpid(kid, 0, 0) < 0) {
        perror("waitpid failed");
        exit(1);
```

```
    }    /*
     * Restart the child from the checkpoint file. A SIGCONT will be sent.
     * This will end the child's sigpause call and cause it to continue.
     */
    if ((rpid = cnx_restart("ex1", RESTART_SIGROOT, SIGCONT)) < 0) {
        perror("restart failed");
        exit(1);
    }

    /*
     * Wait for the restarted child to exit.
     */
    if (waitpid(rpid, 0, 0) < 0) {
        perror("waitpid failed");
        exit(1);
    }
}

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

The checkpoint call

In the code example above, the following call to `cnx_chkpnt()` in `do_parent()` checkpoints the child process:

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

where

`CHKPNT_PROC`

causes only the target process specified by PID to be checkpointed

`kid`

is a variable that contains the PID of the child process that is to be checkpointed

`ex1`

is the name of the checkpoint file (because this is a relative and not an absolute pathname, a file called "ex1" is created in the current working directory)

`CHKPNT_SIGROOT|CHKPNT_FORCE`

is a combination in which the first option causes a signal (specified in the last parameter—`SIGKILL` in this example) 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`

is a signal to send to the root process (because this is a "kill" signal, the target is killed)

The restart call

In the code example above, the following call to `cnx_restart()` restarts the child process:

```
cnx_restart ("ex1",0)
```

where

`ex1`

is the name of the checkpoint file for the process to be restarted; the file is in the current directory

`0`

because the *flags* parameter is zero, no options are used

Programming in Fortran with Checkpoint Restart

5

This chapter:

- Describes the Checkpoint Restart Fortran library functions you can use to build CR capability into applications
- Discusses related programming considerations
- Provides code samples

For additional information about the library functions, refer to the following man pages:

- chkpnt(3)
- cnx_chkpnt(3)
- restart(3)
- cnx_restart(3)
- rmckpt(3)

Conceptual information about how CR works, is located in Chapter 1, "How Checkpoint Restart works."

Fortran Checkpoint Restart functions

Five Fortran library functions that perform checkpointing and restarting are provided:

- `cnx_chkpnt()`
- `chkpnt()`
- `cnx_restart()`
- `restart()`
- `rmckpt()`

These functions work like their C library analogues and are described below. For more information, see the man pages cited above or refer to the chapter "Programming in C and C++ with Checkpoint Restart."

There is no header file analogous to /usr/include/chkpnt.h or /usr/include/cnx_chkpnt.h for the CR Fortran library calls. You must, therefore, make your own provisions for defining constants for your Fortran program that are defined 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 section titled "Fortran programming example" for more information.

The Fortran functions are only available with the Convex Fortran product, `fc`.

Fortran checkpoint functions

The Fortran `chkpnt()` and `cnx_chkpnt()` functions work like the corresponding C library functions. Like the C counterparts, the Fortran functions invoke /usr/bin/chkpnt at run time.

For the Fortran functions to work, the `chkpnt` utility must reside in the default directory in which it was placed by the installation script (/usr/bin/chkpnt).

Formats and parameters

The Fortran `cnx_chkpnt()` function has the following format:

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

This function has the same parameters as its C counterpart. The parameters are summarized below; refer to the section titled "cnx_chkpnt() format and parameters" for a complete description.

`integer` *class*

`integer` *pid*

`character*(*)` *name*

`integer` *options*

`integer` *signo*

See "cnx_chkpnt() return values and error codes" for information on return values and error codes.

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

```
integer function chkpnt (class, id, path, flags)
```

This function has the same parameters as its C counterpart. They are summarized below; refer to the section titled "cnx_chkpnt() format and parameters" for a complete description.

`integer` *class*

`integer` *id*

`character*(*)` *name*

`integer` *flags*

See "cnx_chkpnt() return values and error codes" for information on `chkpnt`'s return values and error codes.

Fortran restart functions

The `restart()` and `cnx_restart()` Fortran functions work like the corresponding C library functions. Like its C counterparts, the Fortran functions invoke /usr/bin/restart at run time.

For the Fortran functions to work, the `restart` utility must reside in the default directory in which it was placed by the installation script (/usr/bin/restart).

Formats and parameters

The Fortran `cnx_restart()` function has the following form:

`integer function cnx_restart (`*path*`,` *flags*`,` *signo*`)`

This function has the same parameters as its C counterpart. The parameters are summarized below; refer to "cnx_chkpnt() format and parameters" for a complete description.

`character*(*)` *path*

`integer` *flags*

`integer` *signo*

See "cnx_chkpnt() return values and error codes" for information on return values and error codes.

The Fortran `restart()` function has the following parameters:

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

This function has the same parameters as its C counterpart. They are summarized below; refer to "restart() format and parameters" for a complete description.

`character*(*)` *path*

`integer` *flags*

See "restart() return values and error codes" for information on `restart`'s return values and error codes.

Fortran rmckpt function

The `rmckpt()` Fortran function works like the corresponding C library function.

Format and parameters

The Fortran `rmckpt()` function has the following parameters:

`integer function rmckpt(`*path*`)`

This function has the same parameters as its C counterpart. The parameters are summarized below; refer to "rmckpt() format and parameters" for a complete description.

`character*(*)` *path*

See "rmckpt() return values and error codes" for information on the associated return values and error codes.

Programming guidelines

If you are writing an application that might get checkpointed, adhere to the following guidelines:

- Do not use any resources not supported by Checkpoint Restart. For example, do not use:
  - Sockets
  - File locks
  - Memory locks
  - Unsupported devices
- 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 fails if it is issued after the application is restarted, because `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. It is difficult for `chkpnt` to locate all the processes of the hierarchy when no top-level parent exists.

Fortran programming example

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

```
C-------------------------------------------------------------------------C
C A FORTRAN example of how to have a program checkpoint itself          C
C-------------------------------------------------------------------------C

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

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

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

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()
        write(*,*)'pid = ',pid
        ret_val = cnx_chkpnt(CHKPNT_PROC, pid, 'ex1', 0, 0)
        if (ret_val .ne. 0) call perror('cnx_chkpnt')
            write(*,*)'cnx_chkpnt failed'
            call exit(1)
        end
```

chkpnt error messages

A

This appendix lists error messages produced by the `chkpnt` utility when used directly or through a programming interface. Because the content of many error messages is variable, the messages are listed in this chapter as `printf` format strings, where:

`%d`	Stands for a numeric (decimal) element in the actual error message
`%s`	Stands for a character string
`%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*).

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

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`

Performing a `stat` command 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): cnx_fdpath failed: %m`

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

`%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) 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` command option `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 hw_regs (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): cannot checkpoint non-swappable process`

The indicated process was created from an executable that was marked as nonswappable. 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 nonswappable.

`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*) **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(*%d*) **fd** *%d %s*: **drain fifo close for write:** %m

%s(%d) **fd** *%d %s*: **drain fifo open for write:** %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*: **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 overwrites existing checkpoint files if the -F (force checkpointing) 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* **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 (verbose output) option is used with the -n (perform checkpointability tests) 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 use stat on the mapped file. This could happen if the target process has unlinked the file after mapping it into its address space.

missing pid argument

No process ID was given on the command line.

pgetaddrmap *%d*: *%m*

The cnx_pcontrol, CNX_GETADDRMAP 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 (in other words, 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): region 0x%x overlaps with restart address space

This process contains valid virtual addresses in the range 0xb0000000 to 0xbfffffff. This address range conflicts 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.

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

%s: core_write %s: %m

An error occurred while writing to the checkpoint file.

restart error messages

B

This appendix lists error messages produced by the restart utility when used directly or through a programming interface. Because the content of many error messages is variable, the messages are listed in this chapter as printf format strings.

%d	Stands for a numeric (decimal) element in the actual error message
%s	Stands for a character string
%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).

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

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 line contains a slash, and restart encountered an error by using 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(%d): can't restore pending signal %d: kill: %m

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

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

`restart` 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 (cwd) at the time of checkpoint. In this case, `restart` makes (using mktemp(3)) a temporary directory in 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(%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` attempts to change the root directory only if invoked by the super user.

close fifo %s: %m

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

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 command 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 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* **cnx_sc_fork: node** %d %m

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

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

cnx_sc_fork: failed for *%s(%d)*: **node** *%d %m*

These messages indicate 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*

the restart 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 the restart 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 the restart 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: **unknown region type** 0x%x **at** 0x%x

A region was found that was not one of CNX_PI_TEXT, CNX_PI_OTHER, CNX_PI_STACK, CNX_PI_BSS, CNX_PI_SHM, or CNX_PI_MMAP.

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.

%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_ANONYMOUS 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.

%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(*%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 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.

munmap: *%m*

cnx_vtmflags: %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 slash (/) and the directory component of this path is not a directory.

%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. The most likely error is that 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.

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 non-swappable process as normal process

This process was nonswappable, but a checkpoint was forced. The process cannot be restarted as nonswappable, but is restarted as a normal demand-paged process. This verbose message is printed 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 was truncated when written by `chkpnt`. This may happen if the file system became full during the checkpoint.

`%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): cnx_setpattr: %m`

`restart` has failed to restore the process attributes.

`%s(%d): setpgrp2 %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 happens 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): setpgrp2 to %d: %m`

`restart` was unable to restore the process group. This happens 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 happens 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, %d): %m`

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

`%s(%d): setreuid(%d, %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 (using mktemp(3)) a temporary shared memory file in the pattern "/tmp/restartmemXXXXXX."

`sigstack: %m`

restart has failed to restore the process signal stack.

`sigvector (%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 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 happens 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 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 happens 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 nonroot 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 cannot restore the proper root directory. Restart the process as the superuser.

Index

A

B

C

D

E

F

G

H

I

R

S

T

U

V

W

X

Z

B5655-90027