Chapter 5

Checkpointing

The CM-5's checkpointing utility, CPR (for CheckPoint/Restart), lets you save a copy of a CM-5 program's state; you can subsequently restart execution of the program from this state. This utility is especially useful for a program with a long execution time; for such a program, it is important that execution not have to start over from the beginning because of a system crash, or because the program has used up its allotted resources under DJM.

You can use CPR to checkpoint CM Fortran and C* programs, and hostless CMMD programs written in C. Checkpointing requires CMOST Version 7.4 or later; CMMD programs using CPR require CMMD Version 3.3 or later.

NOTE: CPR is available only on CM-5s with an SDA.

5.1 Methods of Checkpointing

There are two ways to checkpoint and restart a program:

- A command-line interface lets you checkpoint and restart a program from a UNIX shell. The commands are described in Section 5.7.
- A set of C and CM Fortran functions lets you checkpoint and restart from within a program. (C* programs can use the C functions.) The C functions are described in Section 5.8; the CM Fortran functions are described in Section 5.9.

Copyright © 1993–1995 Thinking Machines Corporation

5.1.1 Active and Passive Checkpointing

There are two kinds of checkpointing:

- *active* checkpointing, in which a program checkpoints itself
- *passive* checkpointing, in which a program is checkpointed by an external process, or by a command

5.2 Writing a Program to Be Checkpointed

This section describes some issues involved in writing a program that can be checkpointed. See Sections 5.8 and 5.9 for descriptions of the actual checkpointing calls.

5.2.1 Limitations

Note these limitations in writing programs to be checkpointed:

- You cannot checkpoint a program that is linked with a shared library.
- You cannot checkpoint **`setuid`** or **`setgid`** processes.
- You can checkpoint only a single process, not a process group. Parent/child/sibling relationships, process group IDs, pipes, and sockets are not preserved through a checkpoint and restart.
- You cannot checkpoint a program that uses the DataVault.
- The CPR library relies on the use of a signal for checkpointing; any catchable signal except **`SIGCONT`** can be used. Therefore, a program that is going to be checkpointed must not use all catchable signals. The library attempts to use a default signal selected when the product is installed at a site, but the application program can override this selection. See Section 5.2.4 for more information.

See Section 5.5 for more limitations in using CPR. See Section 5.2.6 for a discussion of what happens if your program is not checkpointable.

Copyright © 1993–1995 Thinking Machines Corporation

5.2.2 Languages

As mentioned above, you can checkpoint programs written in C, C*, or CM Fortran.

NOTE: The CM Fortran interface has limited functionality. To gain access to complete CPR functionality, CM Fortran programmers should include C modules that contain the appropriate C calls.

5.2.3 Include Files for Checkpointing

C and C*

C and C* programs that use active checkpointing or the application callback mechanism (see Section 5.8) must include the checkpoint header file **`/usr/include/cmsys/chkpnt.h`**.

CM Fortran

CM Fortran programs that use CPR calls must include the file **`/usr/include/cmsys/CMF_chkpnt.h`**.

5.2.4 Specifying the Checkpoint Signal

CPR uses a signal to gain control of the process to be checkpointed. It provides mechanisms that let each application program override the site-specific default signal choice. Ask your local system administrator for the default signal at your site. You can override this default by setting an environment variable or by explicitly selecting the signal in the application, as described below. Note these requirements in choosing a signal:

- The signal must be catchable.
- The signal must not be **`SIGCONT`**.

Note also that the signal is used only by the internal mechanisms of CPR; it is not intended for use by the application program.

Copyright © 1993–1995 Thinking Machines Corporation

Setting an Environment Variable

You can set the environment variable **CMCPR_CHKPNT_SIGNAL** to override the local default checkpointing signal. For example, to choose signal **SIGALRM**, set the environment variable as follows (for the C shell):

```
% setenv CMCPR_CHKPNT_SIGNAL SIGALRM
```

Using a Global Variable

You can also choose a signal by statically assigning the signal to the global variable **CMCPR_chkpnt_sig** in your program. For example,

```
int CMCPR_chkpnt_sig = SIGALRM;
```

(in C) sets the signal to **SIGALRM**. This overrides both the setting of the environment variable **CMCPR_CHKPNT_SIGNAL** and the site-specific default signal.

5.2.5 Using Callbacks

CPR lets you write functions, known as *callbacks*, that can be called on checkpoint and restart. You can use these callbacks for any purpose. For example,

- A callback executed on checkpoint can serve as an early warning that the process is about to be checkpointed. The process can use it to clean up temporary files and manually save whatever it needs that CPR doesn't handle automatically.
- A callback can notify other processes that this process is about to die or has just been restarted.
- Callbacks executed on restart can restore the state of physical devices used by the application, recreate temporary (unlinked) files, restore socket or pipe connections to other processes, or restore any other system state that CPR does not automatically restore.

The function **CM_on_chkpnt** installs a callback to be called at the time of checkpoint. See Section 5.8.4 for more information.

Copyright © 1993–1995 Thinking Machines Corporation

The function **CM_on_restart** installs a callback to be called at the time of restart. See Section 5.8.6 for more information.

Note these points in using callbacks:

- **CM_on_chkpnt** and **CM_on_restart** are available only in C and C*.
- Callbacks can be executed only on the partition manager, not on the nodes. Callbacks my not touch the nodes in any way. All callback logic must be executed exclusively on the partition manager.
- Callbacks return an integer. If the return value of a checkpoint callback is nonzero, the checkpoint will be aborted. If the return value of a restart callback is nonzero, the restart will be aborted.
- User-written callbacks called on checkpoint and restart are executed in the context of a signal handler, and so may not use nonreentrant library functions such as **malloc**, **fread**, and **fwrite**.
- You cannot install a callback from inside a signal handler.

5.2.6 Specifying What to Do if There Is a Problem

If a program that uses the CPR library is not checkpointable for some reason, an error message is sent to the UNIX system logger (via **syslog**). If you want the program to run anyway, set the environment variable **CMCPR_ERROR_OPTIONS** to **START_PROCEED**. If **CMCPR_ERROR_OPTIONS** does not exist or is not set to **START_PROCEED**, the program will send the error message to the logger and exit.

If a program inherits certain files and is checkpointed, a restart normally requires those files to be present at the same file descriptors as they were during the original execution. If they are not present, the restart normally fails. If you want the restart to proceed anyway, set **CMCPR_ERROR_OPTIONS** to **INHERIT_PROCEED**.

You can set **CMCPR_ERROR_OPTIONS** to both **START_PROCEED** and **INHERIT_PROCEED**. Separate them with a comma, with no spaces between. For example,

```
% setenv CMCPR_ERROR_OPTIONS START_PROCEED,INHERIT_PROCEED
```


Copyright © 1993–1995 Thinking Machines Corporation

5.3 Compiling and Linking a Program for Checkpointing

If a program is to be checkpointed, either actively or passively, you must link it with the CPR library, **libcpr**. For example (for a C* program),

```
% cs myprog.cs -lcpr
```

Note these points:

- The link must be static.
- You must include the CPR library in the object, even if none of the symbols are explicitly referenced by the program. You can do this by using the **-u** option to the linker, **cmld**, with the name of some CPR symbol, such as **chkpnt**.

5.4 What Happens in a Checkpoint

5.4.1 The Checkpoint Directory

When a program is checkpointed, the information describing the state of the checkpointed process is stored in a directory (the name of which you specify) on the SDA. There is only one checkpoint snapshot per directory. CPR sets permissions for the checkpoint directory so that only the owner of the checkpointed process can read it (mode 700). This means that only the owner of the checkpointed process (or root) can restart it, unless the owner (or root) changes the permissions for the checkpoint directory.

Moving a Checkpoint Directory

You can move a checkpoint directory to a different SDA file system. If the SDA to which you move it has a different disk geometry from that of the SDA on which the directory was created, you will not be able to restart the checkpointed process. You can, however, store the directory on an SDA with a different geometry, and subsequently move the checkpoint directory to an SDA with the same disk geometry as the SDA on which the checkpoint directory was originally created. You will then be able to restart the checkpointed process.

Copyright © 1993–1995 Thinking Machines Corporation

5.4.2 Files

A process's image of ordinary files and directories is preserved through checkpoint and restart. CPR automatically reopens files that were opened by the application, but file descriptors that the application inherited must be reopened to the same file descriptors before the application is restarted. It is adequate for the restarting process to simply reopen the files; CPR automatically restores the seek offsets and other state.

File descriptors set by the application program are preserved through the checkpoint and restart process. These includes seek pointers and modes (read/write/append). Inherited file descriptors can be handled by using **`CM_restart`**, as described on page 98, or **`CM_restart_me`**, as described on page 99; these calls let the restarting process fork, then have the child reestablish inherited file descriptors before restarting the checkpointed process.

CPR does not verify the correctness of files. If the application was checkpointed, then modifies its data files, and is then restarted, the restarted program will incorrectly assume that the files are in exactly the same state they were in when the checkpoint was taken.

The state restored by CPR consists of

- The file descriptor, if the file was opened by the application (that is, if it was not inherited).
- The seek offset (that is, the return value of the **`lseek`** call).
- Flags (for example, the process's access to a file (read/write/append), and the set of flags read via the **`fcntl(F_GETFL)`** system call).
- Duplicate relationships. If a file is opened, the file descriptor can be duplicated via the **`dup`** or **`dup2`** system call. This is different from opening the file twice because the duplicate file descriptors share the same seek offset and read/write/append flags.

You can use application callbacks to make copies of the data files, if required. The restart callbacks can either open the copies for the application to use or restore the files to their checkpointed state. See Section 5.2.5 for more information on callbacks.

Copyright © 1993–1995 Thinking Machines Corporation

CMOST creates some files that have the process ID in their names. These files are not reopened on restart. Instead, new files are created that have the process ID of the restarted process.

5.5 Limitations

Note the limitations listed below in using CPR. See Section 5.2.1 for a list of limitations you should be aware of when writing programs to be checkpointed.

- You cannot checkpoint a process that is running under one version of CMOST and then restart it under a different version of CMOST.
- Signals are disabled during a checkpoint. Since checkpointing a process with a large memory may take several minutes, responses to any signals may be delayed considerably; under these circumstances, multiple instances of a signal may appear as a single instance. For example, any user-implemented real-time clock that relies on counting signals will lose time while the checkpoint is in progress.
- You must restart a process in a CM-5 partition that is exactly the same size as the partition in which it was checkpointed. In addition, the partition manager must have the same Sun architecture as the partition manager in which the process was checkpointed; if the PM was originally an SS-2, for example, you can't restart the process in a partition with an SS-10 PM. Finally, if a process using vector units is checkpointed, it must be restarted in a partition with vector units.
- A restart will fail if the program text (executable) is moved or modified between initial execution and restart.
- Checkpointing without application involvement will not work for a process containing state information that CPR cannot properly save — for example, one that
  - knows and uses its own process ID
  - uses unlinked files
  - communicates with another process via a socket or pipe

  Such a process must use CPR's callback mechanism to save and restore state; see Section 5.2.5.

Copyright © 1993–1995 Thinking Machines Corporation

- The NFS mount point for mounted file systems must not be moved between initial execution and restart. CPR does not remember mount points as part of the checkpoint.
- You cannot restart a program whose checkpoint directory has been moved to an SDA with a disk geometry that is different from that of the SDA on which the checkpoint directory was originally created; see Section 5.4.1.
- Devices are not guaranteed to be restored to the state they had at the time of checkpoint. Most devices are not reopened. If the application program reads from or writes to devices, it should reestablish the device state on restart via a restart callback; see Section 5.8.6.
- A program (or a library linked to a program) that uses **`syscall`** directly to make system calls bypasses the system call wrappers that CPR uses to capture file pathnames and mode information. If a program uses **`syscall`** directly, files that are opened, closed, created, or duplicated may not be properly opened, closed, or duplicated on restart.
- The following are not preserved through a checkpoint and restart:
  - asynchronous I/O
  - file locks set via **`flock`**, **`lockf`**, or **`fcntl`**
  - memory protections set via **`mprotect`**
  - memory locks set via **`mlock`** or **`plock`**
  - memory locking or memory/file synchronization set with **`mctl`**
  - scheduling optimizations set via **`madvise`**
  - physical device configurations set by **`ioctl`**
  - pipes, named pipes, and sockets
  - changes of root directory made via **`chroot`** or **`fchroot`**
- There is limited support for memory mapping and unmapping. The process's image of mapped files (mapped into user space via **`mmap`**) is saved and restored if the mappings meet two criteria:
  - There must be 256 or fewer memory-mapped segments. This limit includes any segments that are mapped by libraries with which the application program is linked. (Since some libraries

Copyright © 1993–1995 Thinking Machines Corporation

do use **mmap** to map files, we recommend that the application program not use more than 200 memory-mapped segments.)

- When a file is unmapped with **munmap**, the entire mapped segment must be unmapped with a single **munmap** command. CPR does not support mappings and then partial unmappings.

5.6 Performance

The time required to execute a checkpoint or a restart is the sum of

- A small fixed overhead.
- The time required to execute user-installed application callbacks; this is application-dependent.
- The time required to move all the allocated node memory between physical memory and disk. If a process uses a large amount of node memory, this can take on the order of minutes, eclipsing the time required for the fixed overhead and the application callbacks.

NOTE

During the time that CPR is moving a process's node memory to disk, no CM-5 process can run (since no process can run while parallel I/O is taking place). This means that checkpointing a process with a large node memory allocation can suspend all CM-5 processes for several minutes.

The CPR library uses wrappers to capture certain system calls. These wrappers add some overhead to these system calls:

```
open
creat
close
dup
dup2
```


Copyright © 1993–1995 Thinking Machines Corporation

```
chdir
fchdir
mmap
munmap
```

NOTE: **`fchdir`** is provided by the operating system purely as a performance-enhancing device; it is not guaranteed to work. The **`fchdir`** wrapper in the CPR library simply sets **`errno`** to **`EINVAL`** and returns a value of −1.

5.7 Checkpointing and Restarting from the Command Line

You can use the commands discussed in this section to checkpoint and restart a program. As described in Section 5.3, the program must have been linked with the CPR library.

Refer to Section 5.5 for limitations associated with checkpointing and restarting.

5.7.1 Checkpointing a Process: The chkpnt Command

Use the **`chkpnt`** command to checkpoint a process. The **`chkpnt`** command has this syntax:

`chkpnt` [**`-hkv`**] [**`-t`** *timeout*] **`-p`** *pid path*

where

`-h` requests help.

`-k` kills the checkpointed process (via **`SIGKILL`**) once the checkpoint is complete.

`-v` requests verbose output.

`-t` *timeout* specifies a timeout in seconds. If the target process cannot be checkpointed within the specified number of seconds, the **`chkpnt`** command returns. The checkpoint will not be aborted, but the command does not wait for it to complete.

Copyright © 1993–1995 Thinking Machines Corporation

-p *pid* — specifies the UNIX process ID of the process to be checkpointed. This switch is required.

path — is the pathname of the checkpoint directory on the SDA. When the checkpoint is made, files that describe the state of the checkpointed process are stored in this directory. There is no default directory, and the directory must not already exist. **NOTE:** The target process must have sufficient permission to create the checkpoint directory.

For information about what happens during a checkpoint, see Section 5.4.

5.7.2 Restarting a Process: The restart Command

Use the **restart** command to restart a process that has been checkpointed (either from the command line or from within a program). The process will be restarted as if it were originally executed from the shell in which you execute the command; it will inherit the file descriptors of the shell, not the file descriptors inherited by the original execution.

The **restart** command's syntax is

restart [**-hsv**] *path*

where

-h — requests help.

-s — stops the restarted process via the **SIGSTOP** signal. The process must then be externally restarted via a **SIGCONT** signal.

-v — requests verbose output.

path — is the pathname of the directory containing the checkpointed version of the program you want to restart.

Copyright © 1993–1995 Thinking Machines Corporation

5.7.3 Removing a Checkpoint Directory: The rmchkpnt Command

Use the **rmchkpnt** command to remove a checkpoint directory. Its syntax is

rmchkpnt [**-hv**] *path*

where

-h	requests help.
-v	requests verbose output.
path	is the pathname of the checkpoint directory to be removed.

5.8 Checkpointing and Restarting from within a Program (C and C*)

This section describes how to checkpoint and restart a program via function calls within a C or C* program. The program must include the header file **<cmsys/chkpnt.h>**, as described in Section 5.2.3, and be linked with the CPR library, as described in Section 5.3.

Refer to Section 5.5 for limitations associated with checkpointing and restarting.

The functions are listed in Table 1.

Table 1. CPR functions (C and C*).

Function	Use
chkpnt	Checkpoints a program from the partition manager; POSIX-compliant.
CM_chkpnt	Checkpoints a program (other than the calling program) from the partition manager; includes a timeout.
CM_chkpnt_me	Checkpoints the calling program from the partition manager.
CM_cpr_get_inherited_fds	Fills in a table of inherited file descriptors.

Copyright © 1993–1995 Thinking Machines Corporation

Function	Use
CM_cpr_get_num_inher-ited_fds	Returns the number of file descriptors inherited by the process.
CM_cpr_get_lib_revision	Returns the revision number of the CPR library.
CM_enable_chkpnt	Enables/disables checkpointing.
CM_on_chkpnt	Installs a checkpoint callback.
CM_on_restart	Installs a restart callback.
CM_restart	Restarts a checkpointed process; includes a timeout.
CM_restart_me	Restarts the checkpointed process in the process space of the calling process.
CMMD_chkpnt_me	Checkpoints the calling program from the nodes.
restart	Restarts a checkpointed process; POSIX-compliant.
rmchkpnt	Removes a checkpoint directory; POSIX-compliant.

5.8.1 POSIX Compliance

POSIX document P1003.10 No. 125 defines program interfaces for checkpointing and restarting. CPR provides POSIX-compliant versions of its basic checkpoint and restart calls. Other versions of these calls, beginning with **CM_**, provide additional features not available in the POSIX standard.

5.8.2 Checkpointing from the Partition Manager

CPR provides three functions for checkpointing a process from the partition manager:

- **chkpnt** is POSIX-compliant; it lets you checkpoint either the calling process or another process.

Copyright © 1993–1995 Thinking Machines Corporation

- **CM_chkpnt**, in addition to the features provided by **chkpnt**, provides a **timeout** argument that lets you escape from a checkpoint that is taking too long to execute. You can use this function only to checkpoint a process other than the calling process.
- **CM_chkpnt_me** is like **CM_chkpnt**, except that you use it to checkpoint the calling process.

Refer to Section 5.5 for limitations associated with checkpointing.

The chkpnt Function

Use the **chkpnt** function to checkpoint a program from the partition manager. This function is POSIX-compliant.

Here is the definition of **chkpnt**:

```
#include <cmsys/chkpnt.h>

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

where

class currently must be **CHKPNT_PROC**.

id is the UNIX process ID of the process to be checkpointed.

path is the pathname of the directory on the SDA in which the checkpointing information is to be stored. The directory must not already exist. **NOTE:** The target process must have sufficient permission to create the checkpoint directory.

flags can be either 0 or **CHKPNT_KILL**. If **flags** is set to 0, the process simply continues after the checkpoint. If it is set to **CHKPNT_KILL**, the process is killed after the checkpoint is completed.

Copyright © 1993–1995 Thinking Machines Corporation

The **chkpnt** function takes a snapshot of the process identified by **id** and stores the checkpoint information in the directory pointed to by **path**.

The **chkpnt** function can take an **id** argument of 0 or the process ID of the calling process. In these cases, the calling process will be checkpointed. This is active checkpointing (see Section 5.1.1).

If **chkpnt** is successful, it returns a 0. If it is unsuccessful, it returns a −1.

The CM_chkpnt Function

Use the **CM_chkpnt** function to checkpoint another process from the partition manager.

Here is the definition of **CM_chkpnt**:

```
#include <cmsys/chkpnt.h>

int CM_chkpnt(
    int class,
    pid_t id,
    const char *path,
    int flags,
    int timeout,
    int *errcode);
```

where **class**, **path**, and **flags** have the same meanings and restrictions as for **chkpnt**; see above. In addition,

id is the UNIX process ID of the process to be checkpointed. For **CM_chkpnt**, this cannot be 0 or the process ID of the calling process.

timeout specifies the number of seconds within which the checkpoint must take place. If **timeout** is set to 0, there is no timeout. If the target process specified by **id** cannot be checkpointed, or has not finished checkpointing in the number of seconds specified by **timeout**, **CM_chkpnt** sets ***errcode** to **BAD_CPR_TIMEOUT** and returns.

errcode points to an error code.

Copyright © 1993–1995 Thinking Machines Corporation

CM_chkpnt takes a snapshot of the process identified by **id** and stores the checkpoint information in the directory specified by **path**.

CM_chkpnt can be used only for passive checkpointing; that is, it can only checkpoint another process, and its **id** argument cannot be 0 or the process ID of the calling process. Use the **chkpnt** or **CM_chkpnt_me** function for active checkpointing from the partition manager.

If **CM_chkpnt** is successful, it returns **CM_SUCCESS** (0). If it is unsuccessful, it returns **CM_FAILURE** (−1) and puts an error code in the location pointed to by **errcode**; see Section 5.10 for a list of error codes.

If the target process has installed a callback via **CM_on_chkpnt** that takes too long to execute, the **CM_chkpnt**'s timeout argument provides an escape mechanism for the calling process. It can, for example, kill the target process, or simply go do something else. See Section 5.8.4 for information on **CM_on_chkpnt**.

Note that, except for the error code, calling **CM_chkpnt** with no timeout is equivalent to calling **chkpnt**.

The CM_chkpnt_me Function

Use the **CM_chkpnt_me** function to checkpoint the process issuing the call. Its definition is

```
#include <cmsys/chkpnt.h>

int CM_chkpnt_me(
    int class,
    const char *path,
    int flags,
    int *errcode);
```

where **class**, **path**, **flags**, and **errcode** have the same meanings and restrictions as for **CM_chkpnt**; see above.

CM_chkpnt_me takes a snapshot of the calling process and stores the checkpoint information in the directory specified by **path**.

CM_chkpnt_me provides only active checkpointing; that is, the calling function is always the process to be checkpointed. (Note that it has no **id** argument.)

Copyright © 1993–1995 Thinking Machines Corporation

The function has these return values:

CM_FAILURE indicates that the checkpoint could not be taken. An error code is returned in the location pointed to by **errcode**.

CM_SUCCESS indicates that the checkpoint was taken successfully.

CM_RESTART_SUCCESS
indicates that the process has just been restarted. This happens if the process takes a checkpoint with **CM_chkpnt_me** and is later restarted.

(**CM_chkpnt_me** is analogous to **setjmp** in that the return code tells you if the process has taken the checkpoint or just returned from it.)

CM_chkpnt_me has no timeout. If you have installed a callback via **CM_on_chkpnt** that contains an infinite loop, the callback will simply run uncontrolled. See Section 5.8.4 for a discussion of **CM_on_chkpnt**.

5.8.3 Checkpointing from a Node

Use the **CMMD_chkpnt_me** function in a C/CMMD program to request a checkpoint from a node. Its definition is

```
#include <cmsys/chkpnt.h>

int CMMD_chkpnt_me(
    const char *path,
    int flags,
    int *errcode);
```

where **path**, **flags**, and **errcode** have the same meanings and restrictions as for **CM_chkpnt**; see Section 5.8.2. The **path** and **flags** arguments provided must be the same for all nodes, or no checkpoint will take place, ***errcode** will be set to **BAD_CPR_CMMD**, and **errno** will be set to **ECMMD_INCONSISTENT**.

CMMD_chkpnt_me's return values are the same as for **CM_chkpnt_me**; see page 94.

You can call **CMMD_chkpnt_me** only in a CMMD program.

Copyright © 1993–1995 Thinking Machines Corporation

The function behaves like **CM_chkpnt_me**, except that it is called on the nodes, not on the partition manager. Note, however, that it checkpoints the entire calling process, including the partition-manager component, not just the component executing on the node on which the function is executed.

When a node calls **CMMD_chkpnt_me**, it blocks until all nodes for this process make the same call. When the last node makes the call, the process is checkpointed.

Like **CM_chkpnt_me**, **CMMD_chkpnt_me** has no timeout. If you have installed a callback via **CM_on_chkpnt** that contains an infinite loop, the callback will simply run uncontrolled. See Section 5.8.4 for a discussion of **CM_on_chkpnt**.

5.8.4 Installing a Checkpoint Callback Function

Use the function **CM_on_chkpnt** to install a callback function that is called on the partition manager before the checkpoint snapshot is made. The callback serves, in effect, as an early warning that the process is about to be checkpointed. The process that is about to be checkpointed can use the callback

- to clean up temporary files
- in general, to save whatever it needs that CPR doesn't handle automatically

See Section 5.2.5 for more information about callbacks.

CM_on_chkpnt has this definition:

```
#include <cmsys/chkpnt.h>

void CM_on_chkpnt(
    int (*funcp)(char *, int, void *),
    void *arg);
```

where

funcp points to the callback function to be executed before the process is checkpointed.

arg is an argument passed to the callback function.

Copyright © 1993–1995 Thinking Machines Corporation

Three arguments are passed to the callback function:

char * is a pointer to the pathname of the checkpoint directory. The application can save any information it wishes in this directory, but see the notes below for further information.

int is a "die" flag. If nonzero, this flag indicates that the application will die immediately upon completion of the checkpoint.

void * may be used by the callback function in any way you wish. The **void *** passed to the callback function is the **void *** passed to **CM_on_chkpnt** in the **arg** argument.

CM_on_chkpnt returns **CM_SUCCESS** (0) normally, and **CM_FAILURE** (−1) if the callback could not be installed.

The callback returns an integer. If a callback returns a nonzero value, the checkpoint is aborted. Since callbacks are user code, it is the programmer's responsibility to provide information (such as output to **stderr**) about the cause of the failure.

Note these points with regard to **CM_on_chkpnt**:

- You can call **CM_on_chkpnt** repeatedly, installing many callbacks. When a checkpoint occurs, they will be called in the reverse of the order in which they were installed.
- Callbacks installed via **CM_on_chkpnt** are called whether the checkpoint is active or passive.
- Any files in the checkpoint directory created by the application callbacks must begin with **user_**, and the callbacks may not touch any files that do not start with **user_**. CPR does not enforce these restrictions, but failure to adhere to them may result in unpredictable behavior.
- If the callbacks installed via **CM_on_chkpnt** take too long to execute (for example if blocked on I/O or in an infinite loop), the checkpointing program may time out, as provided for in the **CM_chkpnt** function. See Section 5.8.2.

Copyright © 1993–1995 Thinking Machines Corporation

- Calling **CM_on_chkpnt** from inside a signal handler may result in unpredictable behavior, because the function uses **malloc**, which is nonreentrant.

5.8.5 Restarting a Checkpointed Process

CPR provides these functions for restarting a checkpointed process:

- **restart**, which is POSIX-compliant.
- **CM_restart**, which lets you specify a timeout during which the restart must take place, and provides a mechanism that allows the caller to manipulate the environment to be inherited by the restarted process.
- **CM_restart_me**, which restarts the checkpointed process in the process space of the calling process.

Refer to Section 5.5 for limitations associated with restarting.

The restart Function

Use the **restart** function to restart a process that has been checkpointed.

Here is the definition of **restart**:

```
#include <cmsys/chkpnt.h>

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

where

path is a pointer to the directory on the SDA that contains the information for the checkpointed process.

flags can be either 0 or **RESTART_HOLD**. If **flags** is zero, then the process will continue execution after it is restarted. If **flags** is set to **RESTART_HOLD**, the process is placed in a waiting state, and will not run until a **SIGCONT** is sent to it.

Copyright © 1993–1995 Thinking Machines Corporation

The **restart** function restarts the process whose checkpoint information is stored in the directory pointed to by **path**. It returns the new process ID of the restarted process. If the process could not be restarted, it returns −1.

The CM_restart Function

Use the **CM_restart** function to restart a process. Its definition is

```
#include <cmsys/chkpnt.h>

pid_t CM_restart(
    const char *path,
    int flags,
    int timeout
    void (*funcp)(),
    int  *errcode);
```

where **path** and **flags** have the same meanings as for **restart**. In addition,

timeout specifies the number of seconds within which the restart must take place. If **timeout** is set to 0, there is no timeout. If the target process specified by **id** cannot be restarted in the number of seconds specified by **timeout**, **CM_restart** sets ***errcode** to **BAD_CPR_TIMEOUT** and returns.

funcp is a pointer to a function. If **funcp** is non-NULL, the function pointed to by **funcp** will be called in the restartee (the child process of the fork) before the exec. This callback lets the pre-exec restartee (which is the same code as the restarter) manipulate the environment (such as redirected I/O, pty/tty state, and uid) before the checkpointed process is restarted.

errcode points to an error code.

CM_restart restarts the process whose checkpoint information is stored in the directory pointed to by **path**. It returns the new process ID of the restarted process. If the process could not be restarted within the timeout period, it returns **CM_FAILURE** (−1) and puts an error code in the location pointed to by **errcode**; see Section 5.10 for a list of error codes.

Copyright © 1993−1995 Thinking Machines Corporation

The CM_restart_me Function

Use the **CM_restart_me** function to restart the checkpointed process in the process space of the calling process. Its definition is

```
#include <chkpnt.h>
int CM_restart_me(
    const char *path,
    int flags,
    int *errcode);
```

where **path**, **flags**, and **errcode** have the same meanings and restrictions as for **CM_restart**; see above.

This function restarts the checkpointed application in the process space of the caller. When a process calls **CM_restart_me**, the calling process is overlaid with the restarted process. (In this way, it is similar to **exec**.) If it succeeds, it does not return. If it does return, it returns **CM_FAILURE** (−1) and sets **errcode** to an error code; see Section 5.10 for a list of error codes.

Because **CM_restart_me** behaves like **exec**, it lets the restarting process recreate the restartee's inherited environment as follows:

1. The restarting process forks.
2. The child recreates the inherited environment (file descriptors, pty/tty state, uid, etc.).
3. The child executes **CM_restart_me**.

5.8.6 Installing a Restart Callback Function

Use the **CM_on_restart** function to specify a callback function that is called on the partition manager when a process is restarted. See Section 5.2.5 for more discussion of callback functions. **CM_on_restart**'s definition is

```
#include <cmsys/chkpnt.h>

int CM_on_restart(
    int (*funcp)(char *, int, void *),
    void * arg);
```


Copyright © 1993−1995 Thinking Machines Corporation

where

funcp points to the callback function to be executed after the process is restarted, but before it returns to the point where the process was checkpointed.

arg is an argument passed to the callback function.

Three arguments are passed to the callback function:

char * is a pointer to the pathname of the checkpoint directory. This is provided so that the application can retrieve any information saved in the checkpoint directory by the checkpoint callbacks.

int is a "die" flag. If nonzero, this flag indicates that the application died immediately after creating the checkpoint that is now being restarted. If the flag is 0, the application had the opportunity to perform further computation after the checkpoint was taken, such as changing the contents of data files.

void * may be used by the callback function in any way you wish. The **void *** passed to the callback function is the **void *** passed to **CM_on_restart** in the **arg** argument.

The function pointed to by **funcp** is called when the process is restarted. The callback is executed after the process is restored, but before any other user code is executed.

The callback allows the process to manually restore whatever it may need. For example, you can use this callback to get the new process ID of the restarted process if the process uses its own process ID for anything.

The callback returns an integer. If a callback returns a nonzero value, the restart is aborted. Since callbacks are user code, it is the programmer's responsibility to provide information (such as output to **stderr**) about the cause of the failure.

Note these points with regard to **CM_on_restart**:

- You can call **CM_on_restart** repeatedly, installing many callbacks. When a restart occurs, they will be called in the reverse of the order in which they were installed.

CMOST Version 7.4, April 1995
Copyright © 1993–1995 Thinking Machines Corporation

- Any files in the checkpoint directory created by the application callbacks must being with **user_**, and the callbacks may not touch any files that do not start with **user_**. CPR does not enforce these restrictions, but failure to adhere to them may result in unpredictable behavior.
- Calling **CM_on_restart** from inside a signal handler may result in unpredictable behavior, because the function uses **malloc**, which is nonreentrant.

5.8.7 Removing the Checkpoint Directory

Use the **rmchkpnt** function to remove a checkpoint directory. Its definition is

```
#include <cmsys/chkpnt.h>

int rmchkpnt(const char *path);
```

where **path** is the pathname of the checkpoint directory that **rmchkpnt** is to remove.

5.8.8 Disabling and Enabling Checkpointing

By default, checkpointing is enabled for a process. Use the **CM_enable_chkpnt** function to disable and subsequently enable checkpointing. This is useful if you want to prevent a checkpoint from occurring during a particularly sensitive portion of a program. The function's definition is

```
#include <cmsys/chkpnt.h>

int CM_enable_chkpnt(int);
```

Pass in 0 to disable checkpointing. If a passive checkpoint is attempted while checkpointing is disabled (for example, from the command line or from an external process), the checkpoint will happen when checkpointing is subsequently enabled. If an active checkpoint is attempted from within the program itself, it is ignored.

Pass in 1 to enable checkpointing.

Copyright © 1993–1995 Thinking Machines Corporation

The function returns its previous state: 0 if disabled, 1 if enabled.

5.8.9 Handling Inherited File Descriptors

CPR provides two functions you can use to find out which file descriptors are inherited. You can then use this information to restore these file descriptors, for example via restart callbacks.

Use the **CM_cpr_get_num_inherited_fds** function to find out the number of file descriptors that were inherited by the process. You can use this information to allocate space for a table of records describing these inherited file descriptors; use the function **CM_cpr_get_inherited_fds** to fill in this table.

The syntax of **CM_cpr_get_num_inherited_fds** is

```
#include <cmsys/chkpnt.h>

int CM_cpr_get_num_inherited_fds(void);
```

The function returns the number of inherited file descriptors, or −1 if there is an error.

The syntax of **CM_cpr_get_inherited_fds** is

```
#include <cmsys/chkpnt.h>

int CM_cpr_get_inherited_fds (
                        inherited_fd_rec_t *, int);
```

where

inherited_fd_rec_t * is a pointer to a table of structures provided by the caller; see below.

int is the number of elements in the table.

Copyright © 1993–1995 Thinking Machines Corporation

Each entry in the table has this form:

```
typedef struct
{
    int fd;
    int dup_of;
} CM_inherited_fd_rec_t;
```

where

fd is the inherited file descriptor.

dup_of indicates whether this file descriptor is a duplicate of another file descriptor. If it is a duplicate, it specifies the file descriptor of which it is a duplicate. If it isn't a duplicate, it is set to −1.

The function returns the number of **CM_inherited_fd_rec_t** structures filled in, or −1 if there is an error.

5.8.10 Finding Out the Revision Number of the CPR Library

Use the **CM_cpr_get_lib_revision** function to determine the revision number of the CPR library to which the application program is linked. Its syntax is

```
#include <cmsys/chkpnt.h>

int CM_cpr_get_lib_revision(void);
```

The function returns the revision number of the CPR library.

You can call this function only from the partition manager.

5.8.11 Tentative Definitions of CPR Function Pointers

CPR provides pointers to all CPR users functions, so that a program that sometimes uses checkpointing can be compiled correctly whether or not it is linked to **libcpr**. A user program or library that is sometimes linked to **libcpr** can use these calls through the pointers. (This takes advantage of the ANSI C feature called *tentative definition*; the fundamental purpose of

Copyright © 1993−1995 Thinking Machines Corporation

tentative definitions is to allow a program to determine at runtime whether it is linked to another program.)

The function pointers for the CPR user functions are

Function Pointer	Function
`pid_t (*restart_p)()`	`restart`
`pid_t (*CM_restart_p)()`	`CM_restart`
`int (*CM_restart_me_p)()`	`CM_restart_me`
`int (*chkpnt_p)()`	`chkpnt`
`int (*CM_chkpnt_p)()`	`CM_chkpnt`
`int (*CM_chkpnt_me_p()`	`CM_chkpnt_me`
`int (*CM_on_chkpnt_p)()`	`CM_on_chkpnt`
`int (*CM_on_restart_p)()`	`CM_on_restart`
`int (*rmchkpnt_p)()`	`rmchkpnt`
`int (*CM_enable_chkpnt_p)()`	`CM_enable_chkpnt`
`int (*CM_cpr_get_num_inherited_fds_p)()`	`CM_cpr_get_num_inherited_fds`
`int (*CM_cpr_get_inherited_fds_p)()`	`CM_cpr_get_inherited_fds`
`int (*CM_cpr_get_lib_revision_p)()`	`CM_cpr_get_lib_revision`

For example, you could include the following code in an ANSI C program:

```
int (*CM_chkpnt_me_p)(); /be sure to leave uninitialized */

void maybe_chkpnt()
{
    int errcode;
    char *chkpnt_dirname;

    chkpnt_dirname = set_chkpnt_dirname();
    if (CM_chkpnt_me_p)      /* if linked with libcpr... */
         {                   /* ... then take the checkpoint */
              rval = (*CM_chkpnt_me_p)(CHKPNT_PROC,
                                       chkpnt_dirname,
                                       0,
                                       &errcode);
              ...
         }
...
}
```


Copyright © 1993–1995 Thinking Machines Corporation

If the program was linked with **libcpr**, the checkpoint would be taken when **maybe_chkpnt** was called. If the program was not linked with **libcpr**, the symbol **CM_chkpnt_me_p** would be 0, and the program would branch around the checkpoint logic.

5.8.12 Creating a User Initialization Function

The CPR library contains a tentative definition of a pointer to a function that the user program (such as a user library) can provide. If you set this function pointer, the function will be called before the **main** of the user program. The definition in **libcpr** is

```
void (*CMCPR_user_init_hook_p)(void);
```

A user library can set this function pointer, so that **libcpr** can call an initialization function in the library, which can then install callbacks if needed.

Example

Suppose you always want to restore a device called *dingus* (CPR does not automatically restore devices). You could create a library, **libdingus**, that would be linked to all the relevant applications. The library could include this code:

```
/*
 * Set CMCPR_user_init_hook_p pointer so that CPR library will
 * call libdingus_init() before calling user's main().
 */
void (*CMCPR_user_init_hook_p)() = libdingus_init;

/*
 * Note: testing function pointers for non-zero values not
 * necessary if we only get to libdingus_init
 * via CMCPR_user_init_hook_p pointer, because then call to
 * libdingus_init is only possible from libcpr, so we must
 * be linked to libcpr. Still, it's good practice.
 */
void libdingus_init(void)
{
```


Copyright © 1993–1995 Thinking Machines Corporation

```
  if (CM_on_chkpnt_p)  (*CM_on_chkpnt_p) (chkpnt_callback);
  if (CM_on_restart_p) (*CM_on_restart_p)(restart_callback);
}

int chkpnt_callback()
{
  /*
   * Insert logic here to capture state of dingus device.
   */
}

int restart_callback()
{
  /*
   * Insert logic here to restore state of dingus device.
   */
}
```

Because of the tentative definitions, it is possible to link **libdingus** both to applications that are not linked to **libcpr** (where it will have no effect) and to applications that are linked to **libcpr**. If the **libdingus** in this example is linked to **libcpr**, **libdingus_init** will be called before the user's **main** and the callback functions will be called on checkpoint and restart.

5.8.13 Forcing a Non-Checkpointable Program to Run

You may want a program to be able to use CPR to checkpoint and restart other programs, but not be checkpointable itself. This may be particularly important if the program is normally linked dynamically. Dynamic linking makes a program noncheckpointable, and such a program, when linked with **libcpr**, is prevented from executing unless the check is overridden via the **CMCPR_ERROR_OPTIONS** environment variable; see Section 5.2.6. ANSI C programs can force a non-checkpointable program to run by including this statement in the program linked to **libcpr**:

```
int CMCPR_force_noncheckpointable = 1;
```


Copyright © 1993–1995 Thinking Machines Corporation

5.9 Checkpointing and Restarting from within a Program (CM Fortran)

CPR provides functions for checkpointing, restarting, and enabling or disabling checkpointing from within a CM Fortran program. Their arguments are identical to those of the corresponding C routines, discussed in Section 5.8. If you use any of these routines, you must include the file **/usr/include/cmsys/CMF_chkpnt.h**.

To gain access to other CPR functionality (for example, callbacks), CM Fortran programmers should include C modules that contain the appropriate functions discussed in Section 5.8.

Refer to Section 5.5 for limitations associated with checkpointing and restarting.

5.9.1 Checkpointing a Process

Use the **CM_chkpnt_me** function to checkpoint the calling CM Fortran process. Its syntax is

```
INCLUDE '/usr/include/cmsys/CMF_chkpnt.h'

integer rval
rval = CM_chkpnt_me( class, path, flags, errcode )
```

where

class is an **INTEGER**. Its value must currently be **CHKPNT_PROC**.

path is a **CHARACTER** string that specifies the pathname of the directory on the SDA in which the checkpointing information is to be stored. The directory must not already exist. The process must have sufficient permission to create the checkpoint directory.

flags is an **INTEGER**. Its value can be either 0 or **CHKPNT_KILL**. If **flags** is set to 0, the process simply continues after the checkpoint. If it is set to **CHKPNT_KILL**, the process is killed after the checkpoint is completed.

Copyright © 1993–1995 Thinking Machines Corporation

errcode is an **INTEGER** that specifies an error code.

CM_chkpnt_me takes a snapshot of the calling process and stores the checkpoint information in the directory specified by **path**.

Note that **CM_chkpnt_me** provides only active checkpointing; that is, the calling function is always the process to be checkpointed.

The function has these return values:

CM_FAILURE indicates that the checkpoint could not be taken for some reason. An error code is returned in **errcode**; see Section 5.10 for a list.

CM_SUCCESS indicates that the checkpoint was taken successfully.

CM_RESTART_SUCCESS
indicates that the process has just been restarted. This happens if the process takes a checkpoint with **CM_chkpnt_me** and is later restarted.

5.9.2 Restarting a Process

Use the **CM_restart_me** function to restart the calling CM Fortran process. Its syntax is

```
INCLUDE '/usr/include/cmsys/CMF_chkpnt.h'

integer rval
rval = CM_restart_me( path, flags, errcode )
```

where

path is a **CHARACTER** string that specifies the directory on the SDA that contains the information for the checkpointed process.

flags is an **INTEGER**. Its value can be either 0 or **RESTART_HOLD**. If **flags** is 0, the process will continue execution after it is restarted. If **flags** is set to **RESTART_HOLD**, the process is placed in a waiting state, and will not run until a **SIGCONT** is sent to it.

Copyright © 1993–1995 Thinking Machines Corporation

errcode is an **INTEGER** that specifies an error code; see Section 5.10 for a list.

This function restarts the checkpointed application in the process space of the caller. When a process calls **CM_restart_me**, the calling process is overlaid with the restarted process. If it succeeds, it does not return. If it does return, it returns **CM_FAILURE** (−1) and sets **errcode** to an error code; see Section 5.10 for a list of error codes.

See page 99 for further information about **CM_restart_me**.

5.9.3 Enabling and Disabling Checkpointing

Use the **CM_enable_chkpnt** function to enable and disable checkpointing from within a CM Fortran program.

By default, checkpointing is enabled for a process. Use the **CM_enable_chkpnt** function to disable and subsequently enable checkpointing. This is useful if you want to prevent a checkpoint from occurring during a particularly sensitive portion of a program. Its syntax is

```
INCLUDE '/usr/include/cmsys/CMF_chkpnt.h'

integer rval
rval = CM_enable_chkpnt ( int )
```

where **int** is an **INTEGER** that specifies whether checkpointing is to be enabled or disabled.

Pass in 0 to disable checkpointing. If a checkpoint signal is issued while checkpointing is disabled, the checkpoint will happen when checkpointing is subsequently enabled.

Pass in 1 to enable checkpointing.

The function returns its previous state: 0 if disabled, 1 if enabled.

Copyright © 1993–1995 Thinking Machines Corporation

5.10 Errors and Error Codes

CPR posts warning and error messages to the UNIX system logger. Warnings are posted via **syslog(LOG_WARNING)** and errors are posted via **syslog(LOG_ERR)**.

Errors encountered during checkpointing are never fatal, but always abort the checkpoint. Errors encountered during restart are always fatal, resulting in the death of the process.

Table 2 lists the error codes set by functions in the CPR library.

Table 2. CPR error codes.

Error Code	Meaning
BAD_CPR_CALLBACKS	A user callback returned a nonzero value.
BAD_CPR_CLASS	Bad value in **class** argument.
BAD_CPR_CM	Problem with CMTS part of checkpoint.
BAD_CPR_CM5NOMEM	There isn't enough memory available on the nodes to restart the process.
BAD_CPR_CMMD	Illegal use of CMMD; consult **errno**.
BAD_CPR_COREFILE	Couldn't create or read core file.
BAD_CPR_DATA	Problem restoring data section.
BAD_CPR_DIR	Can't access or create directory of checkpoint files.
BAD_CPR_DIR_EXISTS	Checkpoint directory already exists.
BAD_CPR_DISABLED	Checkpointing disabled by **CM_enable_chkpnt**.
BAD_CPR_DRIVER	Can't open **chkpnt** driver.
BAD_CPR_EXECUTABLE	Executable doesn't match checkpointed information.
BAD_CPR_EXTERNFILE	Couldn't create or read CPR file.
BAD_CPR_INHERITED	Inherited state doesn't match checkpointed information.
BAD_CPR_IOCTL	Checkpoint driver **ioctl** failed.
BAD_CPR_NOMEM	**malloc** returns zero.
BAD_CPR_NOPROC	No such checkpointable process.

Copyright © 1993–1995 Thinking Machines Corporation

Error Code	Meaning
BAD_CPR_PARAMS	Parameters passed in by user are illegal.
BAD_CPR_PATHNAME	Bad value in **path** argument.
BAD_CPR_PERM	Insufficient permission (normally caused by an attempt to checkpoint a process to which the caller has no permission to send a signal).
BAD_CPR_REQUEST	Illegal value in **cpr_request** structure.
BAD_CPR_RESTARTEE	Restartee died prior to finishing restart.
BAD_CPR_STACK	Problem restoring stack section.
BAD_CPR_STARTUP	CPR initialization code didn't set up information correctly.
BAD_CPR_SYSCALLS	Some system call to checkpoint or restart failed.
BAD_CPR_SYSTEMSTATE	A regular system call failed.
BAD_CPR_TIMEOUT	Unable to complete checkpoint in time.
BAD_CPR_USERFILES	Problem saving/restoring open files.

5.11 Sample Programs

5.11.1 C

The C program below demonstrates the use of checkpoint and restart callbacks. It simply loops, printing out its process ID every two seconds. When checkpointed or restarted (via a command, or a call from another process), the callback prints out various information.

Note the extra code required because the program does not make use of the CM-5's nodes, and therefore does not automatically link with **libcmna**. This code is not required in C*, CM Fortran, or C/CMMD programs, since they automatically link with **libcmna**.

Copyright © 1993–1995 Thinking Machines Corporation

```
/*
 * loop.c
 *
 * loop.c is a simple program that is to be the target of a checkpoint.
 *
 */

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

#include "cmsys/chkpnt.h"

/*
 * Following line of code is ONLY for use by a program not linked with
 * libcmna.  It should not exist in a CM-5 program.
 *
 * Tentative definition of pointer to cpr initialization function.
 * See "tentative definition" in ANSI-C.  It is a way to determine
 * at runtime if we are linked with the cpr library.
 *
 */
void (*CMCPR_init_p)(char *);

static int chkpnt_callback_0 (char * dirname, int die, void * myarg);
static int restart_callback_0(char * dirname, int die, void * myarg);

static int pid;

void main(int argc, char ** argv)
{
  int rval;
  int i = 0;

  pid = (int) getpid();

  printf("%s executed as pid %d\n", argv[0], pid);

  /*
   * NOTE: Following code is exclusively for use by a UNIX-only process
   *       that is not linked to libcmna - one that does not use the
   *       processesing nodes (PNs) of the CM-5.
   */

  if (CMCPR_init_p)             /* if linked to libcpr ... */
    (*CMCPR_init_p)(argv[0]);   /* ... then call cpr init function */

  else
```


Copyright © 1993–1995 Thinking Machines Corporation

```
    {
      printf("*** not linked to libcpr! \n");
      exit(1);
    }

  rval = CM_on_chkpnt(chkpnt_callback_0, (void *) &i);
  printf("%s: just installed chkpnt callback 0, rval = %d\n",
         argv[0], rval);

  rval = CM_on_restart(restart_callback_0, (void *) &i);
  printf("%s: just installed restart callback 0, rval = %d\n",
         argv[0], rval);

  while (1)
    {
      sleep(2);
      printf("target program (pid = %d) running: %d\n", pid, i++);
    }

}


static int chkpnt_callback_0(char * dirname, int die, void * myarg)
{
  int * ip;

  ip = (int *) myarg;
  printf("chkpnt_callback_0 called\n");
  printf("\t chkpnt dirname = %s\n", dirname);
  printf("\t die flag = %d\n", die);
  printf("\t pid = %d, loop index = %d\n", pid, *ip);
  printf("\t exit chkpnt_callback_0\n");

  return 0;
}


static int restart_callback_0(char * dirname, int die, void * myarg)
{
  int * ip;

  ip = (int *) myarg;
  pid = getpid();
  printf("restart_callback_0 called\n");
  printf("\t chkpnt dirname = %s\n", dirname);
  printf("\t die flag = %d\n", die);
  printf("\t new pid = %d, loop index = %d\n", pid, *ip);
  printf("\t exit restart_callback_0\n");
```


Copyright © 1993–1995 Thinking Machines Corporation

```
  return 0;
}
```

5.11.2 CM Fortran

The code below does a checkpoint in CM Fortran.

```
      include '/usr/include/cmsys/CMF_chkpnt.h'

      integer numsec,i,j,errcode
      parameter (numsec=60)
      integer rval


      i = 10

      print *,"before checkpoint_0: i =", i

      rval = cm_chkpnt_me(
     &   chkpnt_proc,
     &   "/test1/cpr/fortrantest_0",
     &   0,
     &   errcode)

      write (6,6001) rval,errcode,i
6001  format ('after checkpoint_0: rval = ',I10,' errcode = ',
     &   I10,' i = ',I10)

      i = 20

      print *, "before checkpoint_1: i = ", i

      rval = cm_chkpnt_me(

     &   chkpnt_proc,
     &   "/test1/cpr/fortrantest_1",
     &   chkpnt_kill,
     &   errcode)
```


Copyright © 1993–1995 Thinking Machines Corporation

```
      write (6,6002) rval,errcode,i
6002  format ('after checkpoint_1: rval = ',i10,' errcode = ',
     &   I10,' i = ',I10)

      i = 30

      print *,"about to disable checkpoint, reset i = ", i

      rval = cm_enable_chkpnt(0)

      print *,"disabled checkpoint, rval (should be 1) = ", rval

      do  j = 1,numsec
         call sleep (1)
         print *,"seconds until resume = ", numsec - j
      enddo

      print *,"about to re-enable checkpoint, i =", i

      rval = cm_enable_chkpnt(1)

      print *,"checkpoint reenabled, i = ", i

      print *,"checkpoint reenabled, rval (should be 0) = ", rval

      stop
      end
```


Copyright © 1993–1995 Thinking Machines Corporation