NAME

checkpoint_here, include_bytes, exclude_bytes - checkpointing functions

SYNOPSIS

```
#include <checkpoint.h>

int ckpt_target(argc, argv, envp)
int argc;
char **argv, **envp;

int checkpoint_here()

int exclude_bytes(addr, size, usage)
char *addr;
long size;
int usage;

int include_bytes(addr, size)
char *addr;
long size;
```

DESCRIPTION

libckpt.a is a library of checkpointing functions enabling application programmers to write fault tolerant code. To use libckpt, all you have to do is rename main() to ckpt_target(), recompile, and link with libckpt.a.

This enables libckpt to gain control of the application, perform some initializations, and begin checkpointing. By default, a sequential checkpoint will be taken every ten minutes. Should the system running the application fail, simply invoke the checkpointed program with the special command line option =recover (see OPTIONS below) to resume execution from most recent checkpoint.

checkpoint_here() forces libckpt to take a checkpoint. With clever placement of calls to checkpoint_here() along with calls to include_bytes() and exclude_bytes(), substantial improvements in the performance of libckpt are possible. Observe, however, that these calls are NOT necessary to make libckpt work well.

exclude_bytes(addr, size, usage) informs the checkpointing tool that the range [*addr,* *addr* *+* *size*] is to be excluded from all checkpoints until further notice. If *usage* is equal to the pre-defined constant CKPT_DEAD, exlusion begins immediately (that is, when the next checkpoint is taken). If *usage* is equal to the pre-defined constant CKPT_RDONLY, the range will not be excluded during the next checkpoint, but WILL be excluded from all subsequent checkpoints. This feature enables libckpt to deal correctly with read only memory, which should be checkpointed exactly once. Checkpointing read only memory multiple times is an inefficiency; Never checkpointing read only memory is an error that will yield results which are quite incorrect. See the EXAMPLES section below.

include_bytes() informs the checkpointing tool that the range [_addr_, _addr_+_size_] is to be included in all checkpoints until further notice. Initially, the entire address space from the beginning of the data segment to the end of the BSS segment is included. This is automatic and need not be specified by the user via an explicit call to **include_bytes()**. If the break is extended between checkpoints (via a call to malloc() or sbrk(), for example), the new area is automatically included.

A possible use for the **include_bytes()** and **exclude_bytes()** functions would be to exclude dead varibles from a checkpoint. A variable is said to be dead at a point in the code if for all possible execution paths the variable will have a new value written to it before it is subsequently read. Substantial savings on the size of the checkpoint file are possible if there are large areas of the heap excluded during a checkpoint.

OPTIONS

When invoking your application from the Unix command line, you may give one of two options (in addition to your own command line options):

=checkpoint

Enable checkpointing. This option allows the developer to determine how checkpointing with **libckpt** interacts with the application program without modifying the ._ckptrc_ file each time the application is run. See the PARAMETERS section below for more information on this parameter file.

When your program is invoked using the **=checkpoint** option, it must be the last option on the command line. _argc_ is decremented to hide the presence of this option from your application program.

=recover Recover from a system failure. When you invoke your application with this option, it must be the only option on the command line. **libckpt** will detect the presence of this option and enter a recovery function which restores the application's data space and stack to the state it was in at the time the most recent checkpoint was taken. This includes the command line options you used when the program was originally invoked. The system file table is restored, and the processor's registers are restored, ending with the restoration of the Program Counter. This in effect restarts your application from the point where the last checkpoint was taken.

Consider the following series of events:

You invoke your application (named a.out) as follows:

```
a.out     arg1 arg2 =checkpoint
```

Your application examines the value of _argc_ and finds it holds the value 3.

Your application runs for some time, taking occasional checkpoints, the system fails, and you restart your application with the command line:

```
a.out     =recover
```

Your application examines _argc_ and _argv_ and finds that once again _argc_ holds the value 3, and _argv_[2] holds the string arg2.

RETURN VALUES

checkpoint_here() returns 0 if returning normally (i.e. after taking a checkpoint), 1 when returning from a recovery. On failure, it returns -1 and sets errno to indicate the error.

include_bytes() and exclude_bytes() return 0 on success. On failure, they return -1 and set errno to indicate the error.

ERRORS

Note that the errors ENOCKPT and ETOOSOON are exclusive to _libckpt_ and are unknown to the standard error routines such as perror(). If errno is ENOCKPT or ETOOSOON and you call perror(0) the message "Unkown error" is displayed on the stderr.

If a checkpoint is not taken because not enough time has expired (ETOOSOON) or because a previous checkpoint is still in progress (ECHILD), then subsequent checkpoints may still be taken. In all other cases, if a checkpoint fails (because, for example, the disk is full), then checkpointing is disabled, and all future checkpointing attempts, whether by explicit calls to checkpoint_here() or by timer interupts, will set errno to ENOCKPT and return -1.

EFAULT A call to include_bytes() or exclude_bytes() specified an address range not entirely within the data or BSS segments of user memory.

ENOCKPT
checkpoint_here(), include_bytes(), or exclude_bytes() was called without enabling checkpointing via the use of the checkpointing parameters. See the PARAMETERS section below. This error flag will also be set if a checkpoint is attempted after a previous attempt to take a checkpoint failed.

ETOOSOON
checkpoint_here() was called before _mintime_ seconds had expired since the previous checkpoint. See the PARAMETERS section below.

ECHILD An attempt was made to take a checkpoint while the child forked by a previous checkpoint is still in progress. This error may only occur if the _fork_ parameter is enabled. See the PARAMETERS section below.

PARAMETERS

Several parameters are available to fine tune the operation of _libckpt_. You may enable or disable checkpointing, you may enable or disable incremental or forked checkpointing, and you may specify a minimum and maximum time. These times are used to ensure that enough, but not too many, checkpoints are taken. You may also specify a directory in which checkpoint files are to be created, turn verbose mode on or off, and specify the maximum number of checkpoint files which can be created before they are coalesced. These parameters may be set in a special parameter file, ._ckptrc_, which may appear in either your home directory or the current directory. If both exist, the version in the current directory takes precedence.

All these paramters have defaults. If you wish to accept the defaults, no action is required. The possilbe parameters, their values, and their defaults are as follows:

parameter	range	default
checkpointing	on/off	on

incremental	on/off	off
fork	on/off	off
mintime	[number]	0
maxtime	[number]	600
directory	[directory]	.
verbose	on/off	off
maxfiles	[number]	1

The checkpointing parameter turns checkpointing on or off. If off, the other parameters are irrelevant, and any calls to **checkpoint_here()**, **include_bytes()**, or **exclude_bytes()** will return -1 and set *errno* to **ENOCKPT**.

The incremental parameter enables or disables automatic incremental checkpointing. This type of incremental checkpointing makes use of the **mprotect()** system call and may not be entirely reliable on all systems. Manual incremental checkpointing may be accomplished by turning the *incremental* parameter off and placing calls to **include()** and **exclude()** properly so that sections of the heap which have not changed since the previous checkpoint will not be included. Turning the *incremental* parameter on does this automatically, but since the **mprotect()** system call is flaky, so is automatic incremental checkpointing.

The fork parameter enables or disables forked checkpointing. If disabled, a sequential checkpoint is taken; that is, execution of the application is suspended while the checkpoint file is written to disk. If enabled, a child process is forked which takes the checkpoint while the parent process resumes execution of the application. If an attempt is made to take another checkpoint while this child is still executing, the new checkpoint is NOT taken and **errno** is set to **ECHILD**. If this attempted checkpoint is the result of a call to **checkpoint_here()** (as opposed to a timer interupt), then **checkpoint_here() returns -1.**

You may specify a minimum time which *must* pass before a new checkpoint may be taken using the *mintime* parameter. If **checkpoint_here()** is called before *mintime* seconds have passed since the previous checkpoint, no checkpoint is taken; **checkpoint_here() returns -1 and** sets *errno* to **ETOOSOON**. You may disable this timer by setting *mintime* to 0.

You may specify a maximum time which may pass between checkpoints using the *maxtime* parameter. If *maxtime* seconds expire without **checkpoint_here()** being called, it is called automatically. You may disable this timer by setting *maxtime* to 0.

You may specify a directory in which **libckpt** will write checkpoint files using the *directory* parameter. The default is the current directory.

You may enable verbose mode by setting the _verbose_ parameter to on. If enabled, this causes **libckpt** to write diagnostic messages to the **stderr** when applicable. In particular, messages will be displayed at the beginning of the application, and every time **libckpt** regains control of the process. This is useful for fine tuning **libckpt** using various values in the parameter file. The format of the messages are

CKP [_number_] :_message_

where _number_ is the return value of the **time(0)** function call. An example of such a message (the message displayed when a checkpoint is begun) is:

CKP 774906022 : beginning

The default for _verbose_ is off.

You may specify the maximum number of checkpoints retained by **libckpt** using the _maxfiles_ parameter. Since **libckpt** may use incremental checkpointing, each checkpoint file must be retained during the lifetime of the application. After _maxfiles_ checkpoint files have been created, **libckpt** will coalesce them into a single file. If _incremental_ is off and _maxfiles_ is 1, **libckpt** not only knows that automatic incremental checkpointing is disabled, but also assumes that you have NOT coded incremental checkpointing by hand using **checkpoint_here()**, **exclude_bytes()**, and **include_bytes()**. In this case, only one checkpoint file is ever kept, and no coalescing is performed. This is the default.

Finally, you may enable checkpointing on the command line, (even if the file _.ckptrc_ exists and has _checkpointing_ off). To enable checkpointing on the command line, include the flag **=checkpoint** as the last command line argument. _argc_ is decremented before **ckpt_target()** is called in order to hide the use of the **=checkpoint** flag from the application. Setting the _checkpointing_ parameter on the command line overides whatever settings are found in the ._ckptrc_ files.

EXAMPLES

The simplest example is to simply rename **main()** to **ckpt_target()** and recompile, linking with **libckpt**. This must be done regardless of whether other modifications are included. **libckpt** includes it’s own **main()** function, which simply does some initialization, then calls **ckpt_target()**.

If renaming main() to ckpt_target() is the only modification made, then in the absence of the *.ckptrc* parameter file, libckpt will take a checkpoint every 10 minutes. Several optimizations are available simply by creating or modifying the *.ckptrc* parameter file.

An example *.ckptrc* file appears below:

```
checkpointing on
incremental on
fork on
directory .
verbose on
maxfiles 5
maxtime 300
mintime 0
```

This parameter file will yield excellent performance for many applications. It forces a forked, incremental checkpoint every 300 seconds (5 minutes). Since verbose is on, diagnostic messages will be written to the stderr so performance may be measured.

```
                        #include <checkpoint.h>
                        ckpt_target(argc, argv)
                        int argc;
                        char **argv;
                        {
                                     printf("beginning program\n");
                                     if ( checkpoint_here() )
                                                 printf("returning from a recovery\n");
                                     else
                                                 printf("returning from a simple checkpoint\n")
;
                                }
```

If the a.out resulting from this code is run with no .*ckptrc* file and with =checkpoint as the last command line argument, the output is

```
beginning program
returning from a simple checkpoint
```

If the a.out is then run with =recover as the only command line argument, the output is

```
returning from a recovery
```

The return value of checkpoint_here() is often ignored, since the program state upon a normal return from checkpoint_here() is identical to the program state resulting from the use of the =recover flag. The example is merely indended to illustrate flow of control caused by use of the =checkpoint or =recover flags.

Observe that if the above code is run with <u>neither</u> =checkpoint nor =recover, the output is

```
       beginning program
       returning from a recover
```

since checkpoint_here() will return a -1 in this case and set <u>errno</u> to <u>ENOCKPT</u> to indicate that the checkpoint_here() function was called even though checkpointing was currently disabled.

The next example shows how checkpoint_here() would normally be called:

```
       #include <checkpoint.h>

       ckpt_target(argc, argv)
       int argc;
       char **argv;
       {

                    while(1) {
                                get_input();
                                if (done) break;
                                do_calculation();
                                write_output();
                                checkpoint_here();

                    }

       }
```

In this example, performace improvements may be possible if the user is able to determine that large portions of the data space are dead at the call to checkpoint_here(). In this case, calls to exclude_bytes() and include_bytes() may be used to inform libckpt that these memory locations need not be written to the checkpoint file. If get_input reads to a large global array A of size ARRAYSIZE, then the code could be modified to look like the following:

```
       #include <checkpoint.h>
```

```
ckpt_target(argc, argv)
int argc;
char **argv;
{

        while(1) {
                get_input();
                if (done) break;
                do_calculation();
                write_output();
                exclude_bytes(A, ARRAYSIZE, CKPT_DEAD);
                checkpoint_here();
                include_bytes(A, ARRAYSIZE);

        }

}
```

The third argument to exclude_bytes() forces libckpt to exclude the indicated range of memory immediately rather than after the next checkpoint. You should also observe that in the above example the array A is only dead at the bottom of the loop; thus we must call include_bytes() as shown to ensure that a correct checkpoint will be taken even if a checkpoint is taken as a result of a timer interupt. Such a timer interupt can occur at any point in the code, so we must be careful to make sure that our series of calls to include_bytes() and exclude_bytes() yields AT ALL TIMES a correct list of memory locations to checkpoint.

If we disable timer based checkpoints by setting the *maxtime* parameter to 0, then such care need not be taken, since we know that no such interupt driven checkpoints will occur. The example code may then look like:

```
#include <checkpoint.h>

ckpt_target(argc, argv)
int argc;
char **argv;
{

        exclude_bytes(A, ARRAYSIZE, CKPT_DEAD);
        while(1) {
                get_input();
                if (done) break;
                do_calculation();
                write_output();
                checkpoint_here();
```

```
                        }

                }
```

As a final example, observe how libckpt handles read only data through the use of the *usage* argument to exclude_bytes(). If data is read from disk into an array B of size ARRAYSIZE, and array B is never changed, then B should be included in exactly one checkpoint, the first. The *usage* argument to exclude_bytes() should be CKPT_RDONLY in this case to inform libckpt that the indicated region should be exluded from all checkpoints AFTER THE NEXT, but that this region SHOULD be included in the next checkpoint.

```
#include <checkpoint.h>

ckpt_target(argc, argv)
int argc;
char **argv;
{

        /* the next 3 statements could appear in any order */

        exclude_bytes(B, ARRAYSIZE, CKPT_RDONLY);
        exclude_bytes(A, ARRAYSIZE, CKPT_DEAD);
        read_array_B();

        while(1) {
                get_input();
                if (done) break;
                do_calculation();
                write_output();
                checkpoint_here();

        }

}
```

FILES

ckpt.temp.?	temporary checkpoint file
ckpt.?	checkpoint files
.ckptrc	parameter file

NOTES

If checkpointing is *not* enabled via the parameter file or the =*checkpoint* flag, calls to checkpoint_here(), include_bytes(), or exclude_bytes() all return -1 and set *errno* to NOCKPT. This need not be considered an error.

These return values and error codes are provided for the convenience of the user.

If an application which uses checkpointing and reads from **stdin** is begun by redirecting **stdin** via the shell’s redirection capabilities, and the application is interupted and restarted with the =**recover** flag, you must redirect **stdin** again from the same file. All other open files will be re-opened by **libckpt** You may not redirect **stdout** or **stderr**, nor use pipes, in any application which uses this checkpointing tool.

All old checkpoint files, **ckpt.?**, must be removed from the checkpoint directory before beginning an application which uses checkpointing.

Your application should not change the current working directory (or make other changes to the system state). Doing so will prevent **libckpt** from writing checkpoint files to the correct directory.

BUGS

This tool cannot operate if the application reads from or writes to a pipe.

This tool cannot operate if the application redirects **stdout** or **stderr**.

This tool does not restore calls to **signal()**, (and uses **signal()** itself), so that applications attempting to catch signals using the system call **signal()** cannot be checkpointed.

The only system state restored by **libckpt** is the open file table. Thus your application should not assume that system calls it has made (other than for I/0) are still in effect upon recovery.