A system for Process Checkpointing and Restarting* (Using a core dump)

Asim Shankar

April 19, 2003

Abstract

This report describes a system for checkpointing and restarting UNIX processes. It differs from some existing implementations in that (a) It does not require the executables to be linked with library, so processes can be checkpointed without change and more interestingly, (b) the manner in which a checkpointed process is restarted. Other systems such as [ckpt, esky] have a complex mechanism of restoring the stack and register state of the checkpointed process as both are also used by the restoration code. This system seems to be simpler as the restarted process and the restoration code are in independent address spaces. The system runs only on user-level code and requires no modifications to the kernel.

1 Objective

The core file contains a complete memory dump of the process, thus in theory it should be possible to restore the process to the same state it was in when the core was dumped.

However, there are many unanswered questions when it comes to restarting from this state. What happens to the open file descriptors? Files may have changed, how do you handle sockets, pipes, seeks etc. Then there are issues with process ids - does the process id have to be the same as before? What about the parent-child relationship? Signal handling state - what signals are blocked? How does the process see the time that has elapsed since the checkpoint?

Answers to these questions would affect what exactly does one mean by "restarting" a process from the checkpointed state and how to go about it. However, one would notice that for jobs that are essentially compute intensive, where inter-process communication and signal handling aren't the major point of concern - the process address space has all the information necessary. The point is that restarting from the address-space dump in the core *can* serve a worthwhile purpose.

2 Result

The result so far is a system that can checkpoint and then restart any process along with file descriptors, with the following caveats:

- File descriptors of only regular files, directories and symbolic links can be checkpointed. No character/block devices, sockets or pipes
- Signal handlers are not restored (default ones are used)
- Processes that have used `dlopen()` to open a dynamic library are not restarted successfully
- Programs must be single threaded
- Only a single process will be checkpointed, thus programs that use `fork()`, `exec()` (or other things like `system()` and `popen()`) are in trouble

*`http://www.geocities.com/asimshankar/checkpointing/`

- Programs that use the `mmap()` call to map files to the process' address space cannot be restarted

Given these limitations, which some other checkpointing systems share, it seems that things are done much more simply here than in other systems. Section 4 explains why.

3 Other Systems

Other user-level restarting systems include [libckpt], which requires that programs be linked to the libckpt library before they can be checkpointed. This system looks into details of making the checkpointing process efficient and less time-consuming.

[ckpt] developed at the University of Wisconsin is another user-level library that uses the environment variable LD_PRELOAD to install a signal handler which checkpoints the process. The data dumped by this handler is almost exactly what would be found in a core dump file. [ckpt] does not require the process to be linked to any special library and hence processes can be checkpointed without having to recompile/link them. Restarting is done by having a restart process read the checkpoint file and overwrite its own address-space with the data in the checkpoint file. This is a tricky and involved process and while restoring the code and data of the checkpointed process one must ensure that the code and data of the restart process is not corrupted. The system doesn't provide support for file descriptors.

[esky] is another user-level system that supports file descriptors, `mmap()` and `dlopen()`. Though, it appears that it also has to handle restoration of the stack and registers with care as they are in use by the restoration code itself.

Versions of SGI's IRIX and Cray's UNICOS operating systems provide some kernel support that make possible their checkpointing and restarting utilities. In this implementation, we use the core dump (done by the kernel) as the checkpoint of the executing process.

4 Methodology

Here's an overview of the steps the restart utility takes in order to restart a process given the executable file and the core dump file:

1. Open the executable and core files and read their ELF headers
2. From the NOTES program header of the core file, get the PR_STATUS structure (this has the register values) of the checkpointed process.
3. `fork()`, we now have a CHILD process (which will be the restarted image) and the PARENT process (which sets up the child)
4. CHILD: `ptrace(PTRACE_TRACEME,...)` and then `exec()` the executable file
5. PARENT: Setup a breakpoint in the child
   This is done as follows: Store the instruction in the child at the entry point of the executable and replace it with the `INT3` instruction (opcode `0xCC`). Then do a `ptrace(PTRACE_CONT,...)`. This allows the child process to run till it reaches the entry point (normally the address of the `_start` function). Once here, it will execute the `INT3` instruction which causes a `SIGTRAP` to be generated and returns control to the parent process. (Allowing the child to run till the entry point allows the address space to be initialized and code to be loaded). In the case of statically compiled binaries (e.g.: `gcc` with `-static`), instead of the entry point, we would want to break at the address of `main()`).
6. PARENT: With the help of the LOAD sections in the core file, restore the address space of the child.
   (The program headers with type LOAD specify the virtual address and the offset in the core file where the contents of that address can be found)
7. PARENT: Restore the registers of the child (part of the PR_STATUS structure present in the NOTES in the core file).
8. PARENT: Detach the child (`ptrace(PTRACE_DETACH,...)`)

9. **The CHILD process is now the restored image of the checkpointed process**

The use of the `exec()` call and breaking at the entry point of the program handles the initialization of the process' address space and loading the executable code of the program and the used dynamic libraries (except those explicitly mapped by `dlopen()`). [ckpt, esky] handle the restart by making the restart process overwrite its own address space. This can be quite complicated as one must make sure that the code of the restart process remains intact and there are a host of related issues that must be carefully dealt with. The methodology above is much simpler as the address space of the restart process and the restarted process are completely independent.

File Descriptors - File descriptors are handled with the help of a dynamic library that must be put into the LD_PRELOAD environment variable. This library installs a special signal handler for the `SIGQUIT` signal which dumps information on the open file descriptors to a text file. This text file is then read in during the restart process mentioned above after the `fork()` and before the `exec()` and file descriptors are restored with their offsets.

5 Implementation

5.1 Preamble

Based on the methodology described above, a system was implemented. Some things regarding the implementation:

- The system works on Linux and requires kernel 2.4 or above
  (The `mmap2()` system call is used to allocate pages to the process after the program was `exec()`ed. Kernel 2.2 doesn't seem to have this call implemented)
- The "checkpoint" file used is an ELF core file with type ET_CORE. This implementation works on the IA32 architecture (The architecture affects, among other things, the registers available etc.).
- In such a system, the stack starts at `0xbfffffff` and "grows" to lower addresses
- The `.text`, `.data` and `.bss` segments of the executable are loaded at `0x0804000`. Dynamic libraries are loaded by `ld` at `0x4000000` onwards

5.2 Checkpointing

Checkpointing in this system simply means generating a core dump. Here we describe ways to do that and the slightly different methodology used to checkpoint file descriptors (which are **not** checkpointed in the core dump).

5.2.1 Using a signal

There are some signals (`SIGSEGV`, `SIGQUIT` among others) whose default disposition is to cause the process to dump core and quit. Thus, one way of creating a checkpoint for a running process is to send it the `SIGQUIT` signal. There is a limit to the allowable size of this core dump and many times the default setting is to not allow the core file to be created. To remedy this, before running the process type the following in your shell (bash):

```
ulimit -c unlimited
```

5.2.2 Using *gdb*'s *gcore* command

NOTE: For this we require `gdb` version 5.2 or greater (which implement the `gcore` command).
A debugger can be attached to a running process and then used to manipulate it. `gdb` has a command "`gcore`" that creates a core dump of the process. In fact, with the debugger you can bring a process to a *safe* state before dumping core. For example, if the process opens sockets, does some processing and then closes the sockets then you can use gdb to set a break point where all sockets are closed and then create a core dump. Thus, when the process is resumed from the core file, there were no open socket fds to worry about. To attach gdb to a running process use

```
gdb <executable filename> <process id>
```

5.2.3 Checkpointing file descriptors

The file descriptor table is maintained by the kernel and thus doesn't lie in the process' address space. Therefore, information on open file descriptors doesn't seem to be present in the core file.

Furthermore, various issues arise when trying to restore them, for example, what do you do with sockets and pipes? What happens if the file is moved? etc. This system however, provides rudimentary support for regular files (regular meaning files/directories as opposed to sockets or pipes). On receipt of a `SIGQUIT` signal, we store for each open file descriptor - its descriptor, filename, offset and flags and write all this information into another file. The default signal handler is then restored and the process is sent another `SIGQUIT` signal which forces a core dump. Information on open file descriptors is taken from `/proc/self/fd`.

Use of this special signal handler does not require any relinking, we use the environment variable LD_PRELOAD to load our library (`libsavefds.so`) which installs the special signal handler.

In summary, to checkpoint a process with file descriptors, ensure that libsavefds.so is present in LD_PRELOAD before starting the process and then when you need to checkpoint it, send the process a `SIGQUIT` signal.

6 The *restart* utility

The core component of this system is the `restart` program. Not much had to be done for checkpointing as we basically ask the kernel for a core dump to create the checkpoint (checkpointing file descriptors uses a special library in `LD_PRELOAD`). This program essentially implements the methodology explained in Section 4).

The usage of this utility is shown in Fig. 1. Special mention must go to the -b option which is useful when it comes to statically linked executables. The -b option takes an address as argument, which is the address at which the `exec()`ed process is paused and the state of the checkpointed process is restored. The system requires that by executing all instructions in the program till this breakpoint, the program code and code of required dynamic libraries are loaded into the address space of the process (`ld` does it's job). Most executables are dynamically linked to `libc` and `ld` and the entry point of these executables (`_start` function) has the characteristics required of the breakpoint address. However, in the case of statically linked executables, the entry point is often 0 and at this address even the program code has not been loaded. Hence, for such executables an acceptable breakpoint would be the address of the `main()` function. One could look up the symbol table and determine this address, however in case symbols have been stripped, the -b option can be used to specify it.

Acknowledgments

We would like to thank Dr. Deepak Gupta[1], our course instructor, for his valuable guidance and the creators of Google for the search engine.

References

[ckpt] Victor C. Zandy. ckpt: User-level checkpointing. (University of Wisconsin), `http://www.cs.wisc.edu/~zandy/ckpt/`.

[esky] David Gibson. `http://esky.sourceforge.net`.

[libckpt] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. libckpt: A portable checkpointer for unix. (Appeared in USENIX Winter 1995), `http://www.cs.utk.edu/~plank/plank/www/libckpt.html`.

[1] `http://www.cse.iitk.ac.in/users/deepak`

```
Usage: restart [options] <executable filename> <core filename>

Options:
  -b, --breakpoint=ADDRESS   When execing the program to be restarted then run till given
                             instruction ADDRESS before restoring address space and registers
                             (Default is the entry point of the executable, which is generally
                             the address of the _start function, thus all dynamic libraries are
                             loaded by this time. Specifying this is useful for statically linked
                             executables (Compiled with the --static flag in gcc)).
  -f, --filedes[=FILENAME]   Restore file descriptors from FILENAME created by
                             libsavefds.so (Default FILENAME is "filedescriptors")
  -n, --nostop               Do not pause the restarted process
                             (By default the process must be sent a SIGCONT to continue)
  -s, --select               Make detailed selections while the address space is restored
  -V, --verbose              Be a bit verbose about what is being done while restarting
  -w, --wait                 Wait for restarted process to finish execution
  -h, --help                 Display this help and exit
  -v, --version              Display version information and exit
```

Figure 1: Usage of the restart utility