Hibernator™ - Checkpoint/Restart for Unix†

Chris Maltby
Peter Chubb

Softway Pty Ltd.

ABSTRACT

The emergence of Unix based workstation networks has rekindled the need for a comprehensive checkpoint/restart system as a means for balancing loads on network component machines. Making a valid checkpoint on a Unix system has a lot of challenges for the developer, as state information is scattered in many places, including the inside of pipes. Even defining what constitutes a 'job' is problematic.

This talk will describe some of the more interesting challenges we encountered in developing Softway's Hibernator technology, and give an overview of the related standards and alternatives. The talk will also discuss some possible ways Hibernator can interact with existing application environments and system management tools.

1. What's the problem

Those of us old enough to remember the batch oriented mainframe systems of older days will remember that these systems usually provided a means to "checkpoint" or suspend a running job, and allow it to be restarted later, perhaps after some system downtime, or even more than once if debugging. For big long-running programs, or when CPUs are slow or unreliable, this facility is very useful.

Unix, with its minicomputer heritage has never provided much in the way of checkpoint/restart. This leaves the application developer with the task of building in the required capability to each application with a resultant increase in code size and loss of portability. There are some library based tools such as Condor which have some limited ability to checkpoint processes, but they don't work if your process has done anything too hard for the checkpoint facility to understand.

The requirement for a generic checkpoint and restart for Unix has emerged now that Unix systems are replacing old-style mainframes. The need is fundamentally the same; the user or administrator needs more options than *kill* when faced with very long running or resource hungry jobs.

Also, the growth of networks of high-performance systems has seen system administrators and accountants seek ways to get more work out of them. It would be nice to be able to shift workload around the network from busy machines to more lightly loaded ones, assuming that the environment[1] is homogeneous enough to make this work.

Finally, it would be nice for jobs to be checkpointable by default, rather than by building or linking in some special way, without imposing a big performance penalty for the privilege.

By comparison with batch operating systems like MVS, Unix doesn't make checkpointing easy.

2. What is a checkpoint?

Before it is possible to perform a checkpoint and restart, it is important to define the full nature of the

---

† Unix is a trademark of X/Open

1. CPU type, mounted filesystems, shared library versions etc etc.

problem. Simply put, a checkpoint is a dump of enough of the volatile state information of a "job" to allow that "job" to be reconstructed.

Already we have run into a problem: what is a Unix "job"? A single process is clearly not sufficient. Users might want to checkpoint a job like the one in Exhibit 1 which is (at least) 4 processes.

```
$ grep name billrecords/* | sort +2n | awk -f process_script | pr -3
```

Exhibit 1. A possible long running Unix job

Not only is it hard to identify all the components of such a job (consider how you might kill it), but there is also a lot of hidden state information that must be saved if the job is to be restarted. Besides the (obvious) memory images of each of the processes, they may also have open temporary files[2], as well as devices such as the controlling terminal — and what about the pipes between them? Even more difficult is when the processes hold file locks, or have active connections to the network or to a DBMS.

3. Addressing the problems

3.1 Specifying a "job"

If your shell is co-operative, it might place all the components of a pipeline into a single process group. For example, shells that support job control when running in interactive mode put each command line into a separate process group. Alternatively, they may all be descended from one process (for some shells, that process is the last in the pipeline). Some previous implementations of Checkpoint (e.g., Cray's) allow process families (with a common ancestor) to be checkpointed as well as processes. An early draft of the POSIX standard for Checkpoint/Restart mandated process-family checkpoint; the current one mentions only sessions and process groups.

For the state information to be consistent, the processes must be quiescent before they can be checkpointed. However, if the processes are made not all made quiescent simultaneously then a deadlock or some possible user errors may result. Of course, some processes might like to know that a checkpoint is imminent so they can discard temporary resources or shut down network connections etc.

3.2 What is a file?

The old mainframes originally stored permanent information on tapes. Disks (or drums) were used for temporary storage while the job was running. Even when disks became predominant, the mainframe operating system would distinguish between permanent and temporary files. A checkpoint need only save the contents of the temporary files.

For Unix, this becomes much more problematic. All files are permanent[3] though it is clearly unwise to hope they will all be the same next time the job runs. It seems sensible to allow the user (perhaps when checkpointing the job) to specify templates which match the files to be saved.

This decision exposes yet another problem, which has bothered system administrators for a long time. That is, how do you determine the file name of an open file? Remember that the process (or something else) may well have unlinked the file or renamed it after it was opened. Three approaches could be taken:

1. Replace the shared library[4] with one that captures the file name at *open* time. This doesn't help for file descriptors inherited or obtained through IPC.
2. Make the user specify a list of directories to search for the file. There is an obvious performance penalty for this.

---

2. which may also be mapped into memory
3. even the ones in */tmp* (unless */tmp* is on a RAM-disk)
4. But then what about statically linked processes?

3. Modify the kernel to make it easier to find the name given an open file descriptor.

The next problem is selecting some sort of default checkpoint action for files in case the user doesn't offer one. Perhaps the file's contents should only be saved if the file is open for writing. If the *O_APPEND* attribute is set, then maybe it's only necessary to save the file position so that log files can be truncated to the right place on restart.

But what of a file which is mapped into memory? A writable map can have either a *MAP_SHARED* or a *MAP_PRIVATE* attribute. If it's the latter, then it becomes necessary to work out which parts of the memory image have been changed from the file's original contents so that they can be saved separately.

Some implementations avoid this by only supporting sequentially accessed files.

3.3 When a file is not a file.

A Unix process's open files can also be things like devices, fifos, sockets and pipes. Most processes have at least one device special file open, even if it is only the standard error stream.

The general device case is too hard to implement, but particular devices are possible. The *controlling terminal* device is one that should be recognised and redirected on restart; it's probably standard error as well. With help from device drivers it may be possible to deal with non-streams devices such as tapes or frame buffers, but this problem rapidly becomes too difficult (as well as exponential in size!)

Pipes may look simple but also have hidden complexities. Clearly, unless all the participants in a pipe are included in the checkpoint it will be impossible to make it consistent. Even given that we can identify all the pipe endpoints and determine that all are covered, we must still extract the data in those pipes (and possibly a record structure as well, for SVr4 pipes) and then re-insert it at restart.

But it gets worse. Suppose in the example in Exhibit 1 that the *grep* and *sort* commands have finished, but that the *pr*, the *awk* and its possible children are still running. The pipe between the (now dead) *sort* and the *awk* may has only one end, and it may also contain data (now followed by an end-of-file marker).

If the system runs SVR4[5], then its pipes will be implemented using streams. It's even possible to push streams modules or multiplex drivers onto these stream pipes. The general case is again too difficult to handle; there is probably sufficient dynamic state information to prevent successful restart in this case anyway.

A fifo file is a special case of a pipe, and should present no special problems, provided you can identify all the possible endpoints. One extra difficulty is that a fifo may be in a 'waiting to open' state as well as partially closed.

3.4 File Locks

The POSIX standard, and some existing checkpoint/restart implementations, save file locks in the checkpoint image, and reestablish all locks at restart time. The implementations that do this are generally used in a 'system checkpoint' mode, where all processes that can be checkpointed are checkpointed just before a shutdown, and restart is valid only immediately after a reboot.

In any other circumstance, a process holding a file lock is not a good candidate for checkpoint, because the act of taking a checkpoint breaks the lock semantics.

The eight cases to consider are when a process holds a read or a write lock, when it is checkpointed and continued, or checkpointed and killed, and whether the file is saved and restored or just reopened at restart time. I do not propose to do a full case analysis here, but a few examples will give the idea.

Consider a process holding a file or record lock for writing, that's going to be checkpointed and killed. It's typically holding the lock so that other processes can't see the file in an inconsistent state, and to lock out other writers while it is doing its update.

At checkpoint time, because it is killed, its lock is lost, possibly leaving the file in an inconsistent internal

---

5. and some other recent flavours of Unix

state for later file users.

If it is not killed, but continued, and the file is not saved at checkpoint time and restored at restart time, then when the restart happens, the file lock is reestablished, but the record that the target process was working on will not be in the state that it remembered.

Any implementation could support lock restoration if it can be known that the locks need only protect between processes in the checkpointed set but without detailed knowledge of the process's internal behaviour there seems to be no way to ensure this is true or to police it.

Our implementation refuses to checkpoint processes holding file locks.

3.5 Signals and PIDs

Each Unix process has its own unique process ID, as well as sharing a process group ID and session ID. Process group and session IDs are not returned to the system until all the members have exited. When a checkpointed process or group is restarted, it may need to retain its original PID and group IDs. This is because it may be using saved copies of these IDs for sending signals to other processes both within the checkpoint and outside it. Of course, if processes outside the checkpoint have exited or been restarted this may be a problem.

Unfortunately, there's no way to guarantee that those PIDs haven't been already reassigned to new processes. Only the application developer knows whether this will cause a problem for any given program. At first glance you might consider aliasing processes inside the checkpoint in some special way, so that they can assume new real PIDs, but this could become arbitrarily complicated if the processes are checkpointed again.

A 32 bit PID would at least reduce the chance of a collision, especially if part of it contained a user token. As it was, we needed to add a new *pidfork()* system call to make PID restarts feasible.

Signals and their use present other problems. Signal handling in a modern UNIX system is very complex. Signals can be queued, blocked, held temporarily (during a *sigpause*), have extra information associated with them, etc. All this has to be saved and restored.

3.6 Miscellaneous process things

There are other process attributes that need special care. When a process exits (perhaps when it is checkpointed) a record may be written to the process accounting file containing information about CPU usage etc. To avoid massive problems of the kind which triggered Clifford Stoll on his search for hackers[6], when the process is restarted it must record only the accounting information relating to that instance from restart to exit/re-checkpoint. However, if the process itself asks for its CPU usage, it must be told the cumulative amount, including all previous periods of execution.

Also, the process may have a CPU time resource limit. The limit must be transparently adjusted to provide for CPU time already used. And these are just a few of the little bits of state information that must be saved and restored to make a checkpoint work.

As well, the process may be in the middle of executing a "slow" system call (say *read()* from a pipe). It should transparently resume that system call when it is restarted. This is a hard problem, as for many modern Unices the information as to how far a slow system call has got is held on the kernel stack (rather than in the u area), so it cannot be obtained.

Processes that use floating point are also a problem. Many modern unices optimise use of the floating point unit by not saving its state at context switch time, and by attempting to reschedule a process on the same processor as before. This means that the FP state may not be accessible to the checkpoint process.

3.7 The restart object

Assuming that it is possible to save sufficient state information about the processes we care about, the restarting of those processes is also non-trivial. First, the checkpoint itself needs to be protected against

---

6. a trivial discrepancy in an account

malicious fiddling, especially if it contains setuid processes. Even a small amount of change (or corruption) could result in unpleasant consequences. The checkpoint needs to be stored somewhere safe, and protected against change with both file permissions and secure checksums. Checksums (or at least datestamps) may also be required on some of the data files not saved as part of the checkpoint such as shared libraries.

Another issue is whether it is possible to make the various process checkpoint images executable, or otherwise yet another kernel change is required to load and restart them. Even so, the restart program must do as least as much as the checkpoint program to reconstruct the state from the saved information.

4. Does it work?

In spite of the difficulties mentioned above, it is possible to make a powerful checkpoint and restart for Unix. Between 1990 and 1994 Softway worked on a multi-stage process to develop this capability for Fujitsu's SVR4 based mainframe systems. Since then we have continued to develop it for other more generic targets and to minimise its impact on the Unix kernel. As it stands, Hibernator is capable of checkpointing and restarting processes and process groups with only a small set of restrictions.

It supports multiple options for dealing with open files, which can be specified at checkpoint time as well as when the job begins. It will attempt to reassign the old process IDs when possible and either abort the restart or warn if they cannot be assigned. It handles mapped files, pipes and the controlling TTY device. A job may be checkpointed and killed, or allowed to continue. And finally, the job can ask to be notified of an imminent checkpoint so that it can clean up uncheckpointable attributes and resources such as network connections etc., before indicating it is ready for the checkpoint to proceed.

5. What's next?

There is still plenty of work to be done to establish Hibernator and the concept of Unix checkpoint and restart. Only when the checkpoint/restart capability becomes widespread will it attract the support of the various database vendors. Without this, the DBMS (batch) application needs to have special case disconnect/reconnect code to support checkpointing.

The load balancing problem is easier to work on: it just needs a multi-cpu job scheduler along the lines of NQS. The extension of signal actions to support checkpoint instead of kill would also make the load balancer's task easier. If a job is causing problems or if it exceeds some resource limits, it could be checkpointed by just signalling it.

Graphical user interface environments such as X-Windows are also capable of checkpoint, provided the X server can be persuaded to reveal all the information it holds on behalf of its clients. You could walk up to your desktop display, log in and recover your exact working state from when you were last there — minus a few telnets.

6. Acknowledgements

Thanks to Greg James, Jeremy Fitzhardinge and the whole Hibernator team for endless lunchtime sessions on ways to extract kernel state information and also to re-insert it.