Abstract

Determinator is a distributed, multiprocessor OS that guarantees that applications can exactly repeat their computations. Its practical usefulness has been limited, however, by the lack of a method for storing and retrieving system state between machine power cycles. In this project, a checkpointing system was developed as a persistent storage mechanism for Determinator.

The checkpointing system makes use of some of the ideas introduced in the KeyKOS checkpoint mechanism. It preserves system state rather than simply file state, thereby allowing a user to restart computation with only a minimal loss of work. It uses multiple disk regions in order to assure continual checkpoint reliability after the first checkpoint is taken. A major difference between the two systems is in their understanding of pages. In KeyKOS, disk pages are tightly integrated with the virtual memory system; in Determinator, memory pages map to disk pages, but disk pages do not participate in the virtual memory system. This allows Determinator to function equally well with or without persistent storage.

The checkpointing system developed is both a complete checkpoint system and a robust backup solution. It uses an IDE device as a persistent storage medium, and saves the state of the system automatically at a user-modifiable time interval or manually at a user's request. A system that is power cycled after a checkpoint will transparently resume processing with only the work performed since the last checkpoint lost. Further, once an initial valid checkpoint is taken, there is no moment during normal operation when there is not a valid checkpoint.

I. Introduction

Determinator is a distributed, multiprocessor OS that insures determinicity of computation [1]. One of its current weaknesses is that it is unable to store any state when it is shut down. This limits the utility of Determinator in real-world scenarios, where machine failures often occur without warning. An ideal persistent storage system for Determinator would be a checkpointing system, allowing the machine to resume processing without wasted computing effort. Furthermore, this system would ideally be very robust in the face of failure, never losing more than one checkpoint’s worth of work after taking the first checkpoint.
The system developed is a complete checkpointing system that can also serve as a backup system, thereby easing the implementation of backup solutions for Determinator. Furthermore, rather than integrating tightly with the virtual memory system (as in the KeyKOS checkpoint system), the system developed here remains as modular as possible with respect to the rest of the kernel, making Determinator usable and useful on systems without persistent storage.

Saving and restoring the state of a machine so as to make the intervening power cycle invisible to processes and users requires careful consideration. Certain machine state needs to be saved between power cycles, while other machine state can safely and easily be regenerated on-the-fly at start-up. It is not enough to simply grab the entire memory space and write it to the disk. Besides the obvious performance implications, other types of machine state cannot be blindly saved if the machine is expected to be able to restart. It is much simpler and much more sensible to regenerate structures such as the process queue and the network driver on restart, rather than to try to fix their state.

The checkpoint system introduced here uses a number of ideas from the checkpoint system designed for the KeyKOS operating system [2]. The most significant difference between the Determinator and KeyKOS checkpointing systems stems from the significantly different understanding of “pages” in the two operating systems. In KeyKOS, every page has a “home” location on a persistent store, from which it is on-demand loaded into memory. The KeyKOS paging system uses the disk as an extension of its standard memory. In Determinator, all active pages exist in memory at all times. The disk is only an archive of these pages, and is only changed during a checkpoint. Each addressable page of memory has a designated storage location on the disk.

II. Implementation

The checkpointing system establishes a few storage paradigms in order to facilitate its work. The disk layout for the checkpointing system can be seen in Figure 2. It consists of a superblock containing metadata used by the checkpointing mechanism, and two checkpoint regions. Each checkpoint region is approximately 4GB in size, and comprises a page file and a page bitmap. The page file contains the checkpointed memory pages, organized as an array indexed by each page’s location in memory. The page bitmap is an array that specifies which memory pages are present in the associated checkpoint region.

The disk is formatted on first boot to create a superblock. If there is no checkpoint before the machine is power cycled, the superblock is harmlessly recreated. There is no need to modify any other location on disk except over the course of standard checkpointing operations.

The use of two seemingly functionally-identical regions requires double the space, but
Figure 2: Checkpoint disk layout

<table>
<thead>
<tr>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>128KB</td>
</tr>
<tr>
<td>4GB</td>
</tr>
<tr>
<td>128KB</td>
</tr>
<tr>
<td>4GB</td>
</tr>
<tr>
<td>1 sector</td>
</tr>
</tbody>
</table>

Figure 2: Checkpoint disk layout

has significant advantages for reliability. The two regions are in fact used differently. While both can potentially serve as checkpoint regions, a checkpoint is always first written to region 1, followed by an incremental update to region 2. When region 2 contains an incremental update, this update is copied to region 1 before the new update is written to region 2. This approach requires extra disk transfers, but has significant reliability advantages. After the first checkpoint, there is no moment under correct operation when a Determinator checkpoint volume will be corrupted. The disk access code expects that the disk will have at least $512 + 512 + 2 \times (2^{32} + 2^{20})$ bytes of storage space (about 8.6GB) which can be addressed using logical block addressing. Very few modern workstations have less than 50GB of persistent storage, so the storage requirement of Determinator is very reasonable. The checkpointing system will fail unexpectedly if there is not enough hard drive space.

Since Determinator cannot know in advance how many pages of memory the machine will have, it creates 4GB page areas on the disk. This usage of space is inconsequential in the face of 2TB harddisks and economical 128GB SSDs; in return, it greatly simplifies the mapping between memory pages and their checkpointed versions.

The Determinator checkpointing system is designed to work only with 32-bit Determinator kernels. On a 64-bit system, the disk layout would need to be significantly changed in order to accommodate the maximum potential addressable space in a 64-bit system. The current checkpointing system relies on the ability to create a sparsely-populated array on disk that is indexed by page locations in memory. A 64-bit checkpointing system would require a more complicated form of memory-to-disk mapping.

III. Saving a checkpoint

The checkpoint mechanism is bootstrapped off of the timer handling code. The timer interrupt handler calls the timer code in the checkpoint system, which fires a checkpoint after a pre-set delay. This delay interval is set at compile time; however, there is a system call and associated userland utility that allow the user to set the checkpoint interval at will. This utility and other associated system calls also allow the user to enable or disable automatic checkpointing, perform a manual checkpoint, and check the status of the checkpoint system.

Any checkpoint present on the disk will be loaded automatically, and the system will continue to perform checkpoints if automatic check-
pointing was not stopped before the restored checkpoint. The manual checkpointing functionality allows programs (or users) to save data regularly, mimicking part of the behavior of a traditional file system. Performing a manual checkpoint resets the automatic checkpoint counter, but does not disable automatic checkpointing.

When a checkpoint is about to begin, all processors enter a spin-loop except the boot processor. Once the boot processor has detected that all other processors are spinning, it begins the checkpoint. The page directory and page table space is scanned to determine which pages have been written to since the last checkpoint. This is done recursively over all the processes in a running system (since each process in Determinator has its own page directory and virtual address space). The page directory/page table scanning routine creates a list of memory pages that need to be written out to disk. The pages that represent processes in the process tree are also marked dirty – it is assumed that their state will have changed. This special case is needed because processes will not necessarily be marked dirty by the page directory/page table scanning code, as they do not necessarily exist in any page table. All pages allocated with Determinator’s memory allocator will also be marked dirty when they are allocated. This is necessary because certain kernel components allocate memory, but do not associate the memory they allocate with the virtual memory hierarchy.

With a complete list of pages to write to disk, the checkpoint proper begins. This is the most time-intensive part of the checkpointing process by far, since it requires writing to the storage device. A better IDE driver would significantly improve the speed of checkpointing, likely by at least two orders of magnitude. The checkpointing system first writes out all of the modified pages to disk. While it is doing this, it updates an in-memory version of the page bitmap.

After all modified pages are written to disk, the page bitmap is written to disk. The superblock is then updated to refer to the new or updated checkpoint. The checkpointing operation is very robust – the superblock is the final disk data structure updated.

IV. Restoring from a checkpoint

When the system is power cycled, the checkpoint system examines the superblock to see if there is a valid checkpoint on the disk. If there is not, the checkpoint system initializes the hard disk with a new superblock. If there is a valid checkpoint, the system enters the restoration mechanism. Depending on whether the first or second region contains the most recent portion of the checkpoint, the system copies the first or the first and second page bitmaps into memory. The pages on disk that correspond to pages in the checkpoint are not added to the freelist. Since every page that is not added to the freelist will be restored from a page on disk, this operation does not cause the machine to leak memory over the course of many checkpoints. The checkpointing system checks each page in the page bitmaps and copies all the pages on the disk into main memory. After all pages are in main memory, the superblock is read in order to find the memory address of the root process of the checkpointed system. Since pages on disk are loaded back to their pre-checkpoint memory locations after a restore operation, this memory location will still contain the root process. The system resumes normal operation by adding any processes in the process tree that are either in a ready or a run-
ning state (that is, that were in a ready or running state when the machine was checkpointed) to the ready queue. Once this is complete, the restoration is done and the system continues the boot process.

After a restore operation, certain elements of the start-up process are skipped. The most significant part of the start-up process that is skipped is the loading of userland binaries. These binaries already exist in the checkpoint. Since Determinator processes contain their file systems entirely in paged virtual memory, there is currently no provision for updating Determinator’s userland in a checkpoint.

V. Testing

The system was tested in single and multiprocessor configurations on a QEMU 0.13.91 virtual machine with a 9GB virtual disk for the checkpoint system. The amount of RAM was varied between 96MB and 2GB (the maximum amount allowable by this version of QEMU) over the course of testing. Some unit tests were written to insure the functionality of certain key elements of the system, such as the code to manipulate page bitmaps. A userland test utility was created to facilitate testing the rest of the system. The remainder of the testing was performed ad-hoc, running the system and insuring that it functioned as expected.

VI. Future work

There are many enhancements that could be made to the checkpoint mechanism in order to make it a more robust and practically-useful solution. The IDE driver is of paramount importance – a better IDE driver would have the potential to speed up the operation of the system by at least two orders of magnitude. Assuming a 20MB/s write speed, a complete 4GB system checkpoint would take on the order of four minutes. It might even be possible to use more than one processor in conjunction with direct memory access (DMA) to increase the rate of transfer and processing. The checkpointing system usually operates eagerly, copying immediately and deferring little to background processes. A tighter integration of the virtual memory and checkpointing systems would likely allow for more background disk transfers, at the potential expense of simplicity of disk data structure. The checkpoint system could be integrated with the networking system, allowing nodes to drop out of a distributed system, return, and not bring down the network. It would also be interesting to explore system upgrades, and how to design checkpoint data so that both kernel and userland upgrades would be possible.

References
