User-level Checkpointing Through Exportable Kernel State

Patrick Tullmann Jay Lepreau Bryan Ford Mike Hibler
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
{tullmann, lepreau, baford, mike}@cs.utah.edu
http://www.cs.utah.edu/projects/flux/

Abstract

Checkpointing, process migration, and similar services need to have access not only to the memory of the constituent processes, but also to the complete state of all kernel provided objects (e.g., threads and ports) involved. Traditionally, a major stumbling block in these operations is acquiring and re-creating the state in the operating system.

We have implemented a transparent user-mode checkpointer as an application on our Fluke microkernel. This microkernel consistently and cleanly supports the importing and exporting of fundamental kernel state safely to and from user applications. Implementing a transparent checkpointing facility with this sort of kernel support simplifies the implementation, and expands its flexibility and power.

1. Introduction

Checkpointing is a technique for applications to recover from transient failures through rollback recovery. A complete image of the application is created by the checkpointer. The image must contain enough "state" to enable the checkpointer to reconstruct the application. After a transient failure, the application can be restarted from the image. The required state consists of two distinct parts: the application's memory and its kernel state (e.g., its threads, signal state, ports, etc.)

The memory image of the environment is just a byte copy of the application's memory. Kernel state is not so "visible" and must be extracted and explicitly reconstructed when the application is restored.

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army, under grant number DABT63–94–C–0058. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

We handle memory in the same manner as most checkpointers—by copying it out to stable storage. All user-level checkpointers that we know of infer kernel state by interposing on system calls. By recording information about requests and their results, a checkpointer can retain enough information to re-create the equivalent state when the process is restored. For example, by recording the name of an open file, its file descriptor, and finding the current offset in the file, a checkpointer can re-create kernel state by re-opening the file and "seeking" to the right place in that file.

Given a kernel that imports and exports the state of its objects in a clear manner, we can directly query and restore kernel state. The checkpointer must still convert kernel object state into some externalized, usually linearized, form. This process is termed "pickling."

The ability to import and export kernel state is not a requirement unique to checkpointing. Process migration and distributed memory systems also must be capable of manipulating kernel state.

2. Completely Exportable and Settable Kernel State

The Fluke microkernel architecture [6] provides nine kernel supported object types, or *flobs*, which are needed for memory, synchronization, communication, and control. Flobs encapsulate all user-visible kernel state in well defined objects. For example, a **thread** encapsulates the flow of control in an address **space**. A **mapping** defines a range of memory imported to a space, while a **region** defines a range of memory that can be exported to other spaces. A **mutex** provides one synchronization primitive. Fluke represents pointers to flobs with **references**, so a mapping contains a reference to the region.

2.1. Pickling

A fundamental property of the Fluke kernel is that *all* state in user-visible kernel objects (flobs) is cleanly exportable, at any time. There are two parts to a flob's pickled state: a simple C structure, and a short list of references. The simple structure comprises the raw information in the object. For example, a mapping's state consists of an integer offset in the corresponding region (one can think of a region as a Mach-like "memory object"); a virtual address which the mapping is to represent; the size of the memory represented by the mapping; the access protections for the memory represented by the mapping; and two references. The first reference in the mapping points to the region object exporting the memory. The second reference points to the space in which this mapping is providing the memory.

The state of a flob can be set in exactly the same way: by providing a simple structure and a short list of references. If an application passes inconsistent state into a flob, it may corrupt its own execution, but not the kernel or other processes. Only a few kernel checks of flob state have turned out to be necessary. Fluke port and mutex operations, for example, do no verification of imported state, except to ensure that pointers point to valid, available memory—it is not necessary for the kernel to check the contents of that memory. Some flob types do require a few simple checks of the state. For example, mappings must assure that addresses are valid and page-aligned.

2.2. Atomicity and Restartability of Kernel Operations

Providing consistent kernel object state at arbitrary times requires a fundamental property of all kernel operations: every kernel operation must be either transparently *atomic* or *restartable*. If they were not, an object could have state stored in internal kernel data structures that is not exported by normal kernel operations. The Fluke kernel provides this transparent interruption and restart. Therefore, at no time is an object in a situation where its state cannot be extracted: the complete state of any object is always valid and accessible to user-mode software. Implementing transparent interruption and restart requires some careful design. All Fluke internal functions abort execution with a well defined set of error codes. An error generally unwinds the call stack, undoing state changes. If the error signals an interrupted operation, changes are undone and the system is left in a consistent state. The call will be transparently restarted later. All other error codes roll the kernel state back and dispatch the error as appropriate.

Except for threads, which are automatically stopped upon a `get_state` call, exporting the state does not affect the kernel object itself.

Mach 3.0 [1] attempted to provide interruption and rollback of kernel operations, by providing the `thread_abort()` operation. However, `thread_abort()` will abort a non-atomic operation in a non-restartable way. For example, a multi-page IPC transfer may have transferred an arbitrary amount of data at the time of the abort. `Thread_abort()` returns indicating only that the thread was aborted, not where. The newer `thread_abort_safely()` will return an error if the thread is engaged in a non-atomic operation. By way of contrast, non-atomic and non-restartable operations do not exist in Fluke.

2.3. Nested Process Model

There is one other feature of the Fluke environment that simplifies the task of a checkpointer and broadens its scope. Briefly, in Fluke, a parent process can have tight control over the environment of its child process. Compare this with a traditional Unix environment where a parent sets up a child process and retains almost no relation and even less control over that process. Specifically, a Unix child process can allocate resources independently of the parent, and then may persist after the parent exits. In Fluke, all requests for services initially go to the parent process. As the parent process, a checkpointer can know what references were granted to the child, and more importantly, it can know what they *logically* point to. Understanding the logical connection is what allows the checkpointer to correctly re-establish external connections upon restart.

This nesting model also extends the scope of the checkpointer. The immediate child of a checkpointer may, in turn, manage its own child, or even multiple children. A checkpoint will cover the state of the entire child environment, including children and "grandchildren". If the immediate child is a full multi-process POSIX system manager, then the entire system—PIDs, signal state, everything kept track of by the manager—is included in the checkpoint.

The nested process model is the focus of a paper [5] published in OSDI '96. This paper concentrates on the issues associated with the exportability of kernel state.

3. Related work

The V++ Cache Kernel [3] supports loading and unloading threads, address spaces, and "kernels" between the Cache Kernel and special user process "application kernels." The state in these objects may be changed when they are unloaded from the Cache Kernel. In theory, a checkpointing application-kernel could take advantage of this exportable state in much the same way that the Fluke checkpointer does, although to our knowledge it has not been done.

By contrast, in Fluke, *any* application can get and set the state of its associated kernel objects, not just privileged processes. Additionally, the Cache Kernel imposes a strict ordering to getting and setting the state of its objects—for example, one must unload all of the threads in an address space before unloading the address space. (A Fluke space object is roughly equivalent to a Cache Kernel address space.) Fluke imposes no ordering restrictions.

Note that traditional Unix kernels can export some kernel state, via `/proc`, or by using signal stacks to get process state, or with the `ptrace()` system call used by debuggers.

To our knowledge there isn't any other work on systematically exportable (and importable) kernel state. There is indirect evidence that *not* having systematically exportable kernel state makes checkpointing difficult. When others implemented support for task migration (which also needs to acquire relevant kernel state), on Mach 3.0 [10] to migrate a thread they effectively cleared all kernel thread state with a `thread_abort()`, so that when the a new thread is created it has equivalent kernel state (none). Since `thread_abort()` is not a transparent operation, but can impact IPC state, the Mach 3.0 task migration is not guaranteed to function correctly in all situations.

User-level checkpointers have been implemented in a variety of operating systems with varying levels of service and transparency [2, 4, 11]. Most of them require re-linking with the target application in order to intercept appropriate system calls [11], or use shared libraries. Kernel state is *inferred* from calls to and results of kernel system calls.

KeyKOS [8] and L3 [9] have transparent multi-process checkpointing, but it is an integral part of both kernels, which makes extracting kernel state simpler. In Fluke, checkpointing is a regular application, and runs only when needed. These systems do more for fault tolerance than our current checkpointer, with which we have not yet demonstrated whole-system persistence. However, our system provides more flexibility.

Cray Research's UNICOS [7] can checkpoint processes and related collections of processes—the majority of checkpointers for single machines can checkpoint only a single process image.

4. Design of the Fluke Checkpointer

The Fluke checkpointer is, in Fluke-ese, a "nester." It controls the environment and resources of its child. All requests for services initially come to the checkpointer; in responding to these requests generally a reference to a server port is returned. The checkpointer records what external references it gives to its child.

The checkpointer interposes on the memory management of the child environment so it can "see" the memory the child environment is using. In addition to being part of the checkpoint, the memory image of the child is important for finding and manipulating flobs. (Each flob is associated directly with a piece of memory in its task.) The checkpointer uses a kernel call to find all of the flobs in a region. It then pickles and enters them into internal tables for representing the inter-object references.

While the kernel provides the mechanism to pickle the state of a solitary flob, the checkpointer must provide the mechanism for pickling inter-object links. The current checkpointer simply assigns flobs unique identifiers by which inter-object links are represented.

The references pointing to flobs external to the child environment fall into two classes. The first class consists of references the checkpointer recorded and passed into the child. When the child is pickled, these references are tagged by what they *logically* point to. For example, any reference which matches the `stdin` port reference is pickled with a static identifier that stands for `stdin`, when the process is restored, the restorer will replace these port references with references to the new environment's logical `stdin`. Most external references will have equivalent logical connections in a new environment.

The second category of external references consists of those references that don't have logical equivalents in the new environment, or at least, they don't have equivalents that the checkpointer understands. These external references can become arbitrarily complex, but they are not unique to our kernel—consider checkpointing a running Web browser.

When restoring a child environment from a checkpoint image, the memory image is restored as it would be in a standard checkpointer. To reconstruct the kernel state, flobs are re-created at the appropriate address in the child environment. In a second pass the pickled flob state is injected, and the inter-object references are restored.

There is a lot of room for improvement in the policy our checkpointer implements. There are well known optimizations [4, 11] which are orthogonal to the use of exportable kernel state; we plan to integrate several of these features into our checkpointer in the future.

5. Status and Results

Currently running on the x86 platform are the Fluke kernel and enough services to provide a subset of the POSIX environment, including simple file I/O; process management such as fork, wait, and some signals; and demand-paged memory management. All of the kernel object types (region, mapping, mutex, condition variable, reference, port, port set, thread, and space) are fully implemented.

Before the kernel was fully implemented a concern was that allowing the setting of arbitrary kernel object state might require excessive checking by the kernel, to assure its

robustness. This has not proven to be the case. There are two major reasons for this. First, the majority of the state is opaque to the kernel. Threads, for example, have a lot of register state and the kernel is not concerned with what is in those registers. Second, much of the state is encapsulated in the references. A mapping references both the region exporting memory and the task into which the memory is being mapped. Checking the integrity of a reference is a simple operation for the kernel. Through implementing a user-level checkpointer, as well as test programs, we have demonstrated that the state of all of these kernel object types can be exported to user processes and safely and accurately regenerated.

6. Conclusions

Kernels which support exportable state make transparent, comprehensive checkpointing flexible and simple. Because it can directly query kernel state, our checkpointer does not need to make any link-time modifications to the checkpointed application. It can also run as a regular user mode process and requires no special hooks or backdoors into the kernel, while still retaining the ability to checkpoint complicated multi-process sub-environments. To our knowledge this is the first checkpointer that can operate over arbitrary domains in this manner.

Availability

We plan to make the first release of Fluke before the end of 1996. The checkpointer will be included in this package, along with other nesters, including a virtual memory manager, a process manager, and a transparent debugger.

Acknowledgements

The authors would like to thank Michelle Miller, Colette Mullenhoff, Greg Thoenen, and the anonymous reviewers who provided helpful comments and criticism. Additionally the entire Flux project team at the University of Utah provided substantial help getting us to where we are.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conference*, pages 93–112, June 1986.

[2] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[3] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, Nov. 1994.

[4] E. N. Elnoxzahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[5] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Association.

[6] B. Ford, M. Hibler, and F. P. Members. Fluke: Flexible μ-kernel Environment (draft documents). University of Utah. Postscript and html available under http://www.cs.utah.edu/-projects/flux/fluke/html/, 1996.

[7] B. A. Kingsbury and J. T. Kline. Job and Process Recovery in a UNIX-based Operating System. In *Proc. of the Winter 1991 USENIX Conference*, pages 355–364, 1989.

[8] C. Landau. The checkpoint mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, September 1992.

[9] J. Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[10] D. S. Milojičić, W. Zint, A. Dangel, and P. Giese. Task migration on the top of the Mach microkernel. In *Proc. of the Third USENIX Mach Symposium*, pages 273–289, Santa Fe, NM, Apr. 1993.

[11] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proc. of the Winter 1995 USENIX Technical Conference*, January 1995.