The Checkpoint Mechanism in KeyKOS

Charles R. Landau

MACS Lab, Inc., 2454 Brannan Place, Santa Clara, CA 95050

KeyKOS is an object-oriented microkernel operating system. KeyKOS achieves persistence of objects by taking frequent system-wide checkpoints of the entire system state to disk. Checkpoints do not interrupt operation of the system for any significant period of time. On restart from a failure, the entire system is restored to the state at the last checkpoint, and processes resume execution as if there had been no interruption. This single mechanism ensures persistence of all objects over many years, maintains security, and achieves a high degree of fault-tolerance. The checkpointed state can be written to tape for off-site backup, without stopping the system. A journaling mechanism allows data to be committed between checkpoints. Paging of virtual memory is integrated with the checkpoint mechanism, allowing KeyKOS to achieve extremely high disk I/O performance.

1: Introduction

KeyKOS\(^1\) is a capability-based object-oriented portable microkernel operating system. Capability-based systems [7] lend themselves naturally to the object paradigm because of the close control they provide over the flow of information. Earlier papers [1, 3] have described how KeyKOS supports the full generality of objects by allowing object semantics to be completely determined by user-written programs. This paper describes in detail the mechanism KeyKOS uses to achieve persistence of objects.

It is desirable to make such general objects as useful and easily used as traditional objects such as files. In particular, such objects must be as reliable and secure as files. Clearly their state must be preserved over system restarts. Because the state of an object may be contained (temporarily or long term) in a program’s variables in its virtual memory or in its registers, these must be saved. An object’s state may also be contained in subobjects or peer objects or in its capability registers. In a secure environment, the security policy must be maintained across system restarts [2, 8].

KeyKOS addresses all these concerns by periodically taking a checkpoint. A checkpoint is a snapshot of the complete state of the system at one instant, saved to nonvolatile storage, i.e. disk. When the system is restarted, the state is restored from the most recent checkpoint. Because the complete state of the system is saved and restored, the state (including security state) of all objects is correct and consistent.

Development of KeyKOS began in 1975 at Tymshare, Inc. and was continued at Key Logic in Santa Clara, CA beginning in 1985. Techniques for application-level checkpoints were available before KeyKOS was designed (see for example [4] or [5]). These checkpoint schemes had restrictions. One might not be able to perform a checkpoint if there were resources locked, for example, or if there were multiple tasks active. These older checkpoint facilities also frequently required the same system configuration at restart that had been present at checkpoint (e.g. the same shared routines loaded at the same memory locations).

KeyKOS checkpoints are free of such restrictions and do not require application-specific logic. For example, if a resource is locked, the state of the lock is saved as part of the entire system state. Checkpoints are completely asynchronous with respect to application programs – they can occur at any time. The application is not aware of the taking of checkpoints or of restarting from a checkpoint.

2: System state

KeyKOS is able to ensure the complete state is saved by maintaining a very disciplined approach to storing state information. All state information in the system is stored in two types of primitive objects, pages and nodes.

Pages are the units of virtual memory and correspond to the host machine architecture’s page size. All data is stored in pages. (A page retains its identity when it is swapped in or out of main storage.) The KeyKOS file system is not separate from the virtual memory system. The file system

---

\(^1\) KeyKOS is a registered mark of Key Logic
is built on top of the facilities of the virtual memory system. All data, including data in files, is ultimately stored in pages of virtual memory.

Nodes are primitive objects that can store a fixed number of capabilities (16). They are implemented in software. All capabilities are stored only in nodes. Like pages, nodes are stored on the disk and are cached in main storage when referenced. (For efficiency, nodes are grouped into page-sized units on the disk.)

All other objects, even processes, are built out of pages and nodes. For example, an address space is implemented by a tree of nodes with capabilities to pages at the leaves. KeyKOS saves the complete state of the system by saving the state of all pages and nodes.

It is worth emphasizing that consistency is ensured by saving the state of processes as well as files. If an update of a file involves two different pages and only one of the pages has been modified at the time of the checkpoint, the file will not be damaged if the system is restarted. When the system is restarted the process that was performing the update is also restarted and the second page of the file is updated as if there had been no interruption. A power outage or hardware fault does not leave the system in some confused and damaged state.

2.1: Disk areas

Every page and every node is assigned a permanent home location on disk. When a page or node has not been referenced in a long time, it resides in its home location.

In addition, there are two areas on disk called swap areas. The swap areas are used alternately to temporarily hold copies of changed pages and nodes. A small area on disk called the checkpoint header designates one of the swap areas, called the checkpoint area. The other swap area is called the working area.

The checkpoint mechanism is integrated with the paging mechanism. In the normal course of demand paging, pages and nodes which are swapped out are written to the working area. The working area contains all swapped-out pages from the current checkpoint cycle; the checkpoint area contains swapped-out pages from the previous checkpoint cycle; and the home location contains older versions. To swap in a page, the system looks for the page first in the working area. If not found there, the system looks in the checkpoint area. If not there, the page is fetched from its home location. This ensures that the most recent version of the page is obtained.

Pages and nodes are written to consecutive locations in the working area. The KeyKOS I/O system takes advantage of this and can write consecutive disk locations in a single revolution. As a result, KeyKOS achieves extremely high disk bandwidth utilization (in excess of 90% on all channels).

3: Checkpoint mechanism

Taking a checkpoint can be conceptually understood as performing the following logical steps:

1. All processes in the system are stopped.
2. All changed pages and nodes are written to the working area.
3. The roles of the swap areas are reversed; the working area becomes the checkpoint area and vice versa. The checkpoint header is updated to designate the new checkpoint area.
4. All processes are restarted.
5. The pages and nodes in the new checkpoint area are copied to their home locations. This process is called migration. This makes the checkpoint area available to become the working area at the next checkpoint.

Operation proceeds in a cycle, with a checkpoint phase, followed by a migration phase, followed by a phase with just normal paging activity, followed by another checkpoint phase, etc. In a typical KeyKOS system, a checkpoint is taken every five minutes. Under moderate load, the checkpoint phase takes ten to twenty seconds, and the migration phase takes somewhat longer. The majority of time, the system is neither checkpointing nor migrating. The system administrator can adjust the checkpoint frequency at any time without interruption of service. The system administrator can also force a checkpoint to be taken, for example just before a scheduled shutdown.

The actual checkpoint mechanism is optimized in several respects in order to reduce as much as possible the period of time during which processes are stopped. In normal operation, no I/O takes place while processes are stopped.

In the course of operation, KeyKOS maintains two or three versions of every page and node:

The Current version is the most recent state of the page or node. When a process stores into a page or node, it changes the Current version.
The **Checkpoint** version is the state of the page or node as of the most recent committed checkpoint. If the system were restarted at this point, the Checkpoint version would become the Current version. The Checkpoint version of a page or node is found by looking in the checkpoint area; if it is not found there, it is in the home location.

The **Next Checkpoint** version exists only during the checkpoint phase. It is the state of the page or node as of the beginning of the checkpoint in progress. If the current checkpoint completes and the system is then restarted, the Next Checkpoint version will become the Current version.

At the beginning of a checkpoint, all changed pages in memory are logically copied. This creates a Next Checkpoint version which is the same as the Current version as of the start of the checkpoint. The actual copying is deferred, using a copy-on-write mechanism. If any change is made to the page, a real copy is made. One copy is the Next Checkpoint version; the other is the Current version, which may then be changed. If a page frame is not available to make a copy, the page is written to the working area immediately.

The actual steps of the checkpointing cycle can now be described in detail.

1. The checkpoint phase begins. This is the instant as of which the system state will be saved. All processes in the system are stopped. (This ensures that steps 2 and 3 take place in an “instant” as viewed by processes.)

2. All changed pages are put into a copy-on-write state.

3. All changed nodes are copied to buffers in main memory.

4. The processes are restarted.

5. The buffered nodes and changed pages are saved to the working area. It is the Next Checkpoint versions that are written. If a Current version exists that is different from the Next Checkpoint version (as a result of writing on a copy-on-write page), the Current version is not written out at this time. (It is never necessary to page out Current versions to make room in main storage; Next Checkpoint versions or buffered nodes can be paged out instead.)

6. The roles of the swap areas are reversed. The old working area becomes the checkpoint area. The checkpoint header is updated to designate the new checkpoint area. This action commits the checkpoint. The effect is to discard the (old) Checkpoint state, and the Next Checkpoint state becomes the (new) Checkpoint state. The contents of the old checkpoint area are no longer needed. The old checkpoint area is emptied and is re-used as the working area for the next checkpoint cycle.

7. The checkpoint phase ends and the migration phase begins.

The purpose of migration is to ready the checkpoint area for the next checkpoint, while losing neither the Current version nor the Checkpoint version of any page or node. (The Next Checkpoint version only exists during the checkpoint phase.) If the Checkpoint version is the same as the Current version (i.e. the page has not been changed since the beginning of the last checkpoint), migration saves them by copying the page from the checkpoint area to its home location. The version will be available in the home location after the checkpoint area is re-used at the next checkpoint.

If the page has been changed since the beginning of the last checkpoint, the Checkpoint version is different from the Current version. The Current version is either in main storage or in the working area; discarding the contents of the checkpoint area will not affect it. The Checkpoint version will be discarded at the end of the next checkpoint when the checkpoint area is re-used, so the version in the checkpoint area does not need to be saved. Thus, pages which are frequently changed do not present a load to the migration phase.

Migration of pages and nodes to their home locations is performed at low priority while normal system operation continues. Migration disk I/O is done in an order which decreases delays due to disk arm latencies.

The next checkpoint cannot start until the migration completes. If the migration has not completed when the time for the next checkpoint arrives, migration priority is increased, and the checkpoint is started when the migration does complete.

The system administrator defines the size of the swap areas so as to be adequate for the normal operation of the system. If the working area becomes full, a checkpoint is taken immediately. More precisely, a checkpoint is taken when the working area has just enough space left to hold the pages and nodes to be written by the checkpoint. The priority of migration is automatically adjusted by a mechanism designed to ensure that the migration completes before the swap area fills up. In the very
unlikely event that the working area becomes full before the migration has completed, all processes are stopped while the migration completes.

Checkpointing and paging are implemented entirely within the KeyKOS kernel. Part of the migration logic is performed by a process outside the kernel, to simplify data management.

4: Restart mechanism

When KeyKOS is restarted, the checkpoint header is read to determine which swap area is the checkpoint area. The other swap area, the working area, is considered to be empty. Thus, the system restarts using the Checkpoint version of all pages and nodes. The system state is in effect rolled back to the state as of the last checkpoint.

The complete state of all processes is saved in the checkpoint. This includes the registers, the program counter, and the information as to whether the process is runnable. Processes which were runnable at the time of checkpoint are runnable at the time of restart, and are scheduled and run just as they would be if the system outage had never occurred.

It may be difficult to appreciate the scope of the system state that is saved on checkpoint and restored on restart. To give an example, if a user had a window open on his screen and was editing a file at the time the system went down, on restart his screen will reappear as it was at the last checkpoint and he will still be in the middle of his editing session, with all his changes preserved in the editor. The editor is not aware of either the checkpoint or the restart, and has no special logic to deal with these events. We believe KeyKOS is unique in its ability to checkpoint the complete state of all applications without any knowledge or cooperation from the application programs.

The kernel does contain data that is not checkpointed, such as page tables. This data is always derived from information in nodes and pages, and is reconstructed when the system restarts. For example, page tables are reconstructed, when needed, from the information in the tree of nodes and pages that describes the address space. Restarting the system from a checkpoint does not destroy any information that is essential to the state of the system.

Because the system may have gone down during migration, on restart the migration phase is restarted from the beginning. (There is no harm if part or all of the migration is done twice.)

The reliability of the checkpoint/restart mechanism eliminates the need for other means of consistency checking when the system is rebooted. As a result, system restart times are very short compared to systems such as UNIX.\(^1\)

Some state information cannot be saved with the checkpoint, either because it is external to the KeyKOS system, such as network peers, or because it is impractical to save, such as the contents and position of tapes. On restart, ad hoc mechanisms are used to resynchronize with such external systems. The journaling mechanism (below) provides one way to recover state information that was rolled back on restart.

5: Journaling mechanism

In a typical KeyKOS system, a checkpoint is taken every five minutes. In an unscheduled outage, at most five minutes of work may be lost. That is adequate for many applications, such as word processing. However, transaction processing applications need to be able to commit a transaction and guarantee that the information will not be lost. To support such needs, KeyKOS provides a journaling mechanism.

The journaling mechanism allows an authorized program to update the Checkpoint version of a selected page. The kernel writes the Current version of the page to the home location and to the checkpoint area, if the page exists there. If the system is then restarted, the data in the page at that time will be preserved.

It may be difficult to appreciate the scope of the system state that is saved on checkpoint and restored on restart. To give an example, if a user had a window open on his screen and was editing a file at the time the system went down, on restart his screen will reappear as it was at the last checkpoint and he will still be in the middle of his editing session, with all his changes preserved in the editor. The editor is not aware of either the checkpoint or the restart, and has no special logic to deal with these events. We believe KeyKOS is unique in its ability to checkpoint the complete state of all applications without any knowledge or cooperation from the application programs.

The kernel does contain data that is not checkpointed, such as page tables. This data is always derived from information in nodes and pages, and is reconstructed when the system restarts. For example, page tables are reconstructed, when needed, from the information in the tree of nodes and pages that describes the address space. Restarting the system from a checkpoint does not destroy any information that is essential to the state of the system.

Because the system may have gone down during migration, on restart the migration phase is restarted from the beginning. (There is no harm if part or all of the migration is done twice.)

The reliability of the checkpoint/restart mechanism eliminates the need for other means of consistency checking when the system is rebooted. As a result, system restart times are very short compared to systems such as UNIX.\(^1\)

Some state information cannot be saved with the checkpoint, either because it is external to the KeyKOS system, such as network peers, or because it is impractical to save, such as the contents and position of tapes. On restart, ad hoc mechanisms are used to resynchronize with such external systems. The journaling mechanism (below) provides one way to recover state information that was rolled back on restart.

5: Journaling mechanism

In a typical KeyKOS system, a checkpoint is taken every five minutes. In an unscheduled outage, at most five minutes of work may be lost. That is adequate for many applications, such as word processing. However, transaction processing applications need to be able to commit a transaction and guarantee that the information will not be lost. To support such needs, KeyKOS provides a journaling mechanism.

The journaling mechanism allows an authorized program to update the Checkpoint version of a selected page. The kernel writes the Current version of the page to the home location and to the checkpoint area, if the page exists there. If the system is then restarted, the data in the page at that time will be preserved.

This simple mechanism supports a range of recovery policies. It is up to the application to examine the journaled data on restart and recover appropriately. KeyKOS provides a facility that applications can use to implement full commit/abort transaction processing.

To ensure consistency of the security-related state of the system, journaling of nodes is not allowed.

6: Mirroring

To further increase reliability, KeyKOS stores each node and (optionally) page on two separate disks at all times. This allows KeyKOS to recover from any single disk failure without losing any data and without interrupting service. If a disk is repaired and brought back on line, it is automatically brought up to date, again without interrupting service. Care is taken that at every point in time a valid checkpoint state exists that can be used to restart the system if a failure occurs at that point.

While mirroring doubles the amount of disk space required, the penalty in performance is not severe. Disk

\(^1\) UNIX is a registered mark of AT&T Bell Laboratories, Inc.
writes occur in the background. Disk reads are actually more efficient because the system chooses the copy that can be most quickly accessed.

7: Tape checkpoint

KeyKOS has a facility for writing a complete checkpointed state of a system onto tape, while the system runs. Because writing the entire system state takes longer than the time between checkpoints, the tape checkpoint mechanism synchronizes with the checkpoint/restart mechanism to keep track of pages and nodes which are changed. The Checkpoint versions of all pages and nodes are written to tape in an initial pass. Then pages and nodes which have been changed since they were written are appended to the tape. This process repeats until a consistent state of all pages and nodes has been saved. The tape checkpoint is independent of machine configuration and can be restored and restarted on any system with sufficient disk space. In fact, this is the way KeyKOS is distributed.

8: Practical experience

KeyKOS has been in production use since 1983. Because the KeyKOS checkpoint/restart mechanism introduces a significant departure from the traditional reliability paradigm, it is worth examining the experience gained in using the system over this period.

KeyKOS systems have run for periods of as long as three years. Processes have existed and run over that entire period, through power outages, hardware failures, and software failures.

Key Logic developed a prototype UNIX-compatible system implemented on top of KeyKOS. At UNIForum '90, we demonstrated this system by literally pulling the plug on the computer at random. Within 30 seconds of power restoration, the system had resumed processing, complete with all windows and state that had been on the display. We are aware of no other UNIX implementation with this feature today.

8.1: Fault tolerance

The checkpoint/restart mechanism provides a simple and reliable way to recover from hardware failures. Rather than attempt an orderly shutdown in the presence of essentially arbitrary hardware malfunction, while maintaining system security and integrity, the kernel simply performs a software "crash", stopping execution with the anticipation of restarting from the last checkpoint after the hardware is repaired.

The checkpoint/restart mechanism provides a measure of recovery even from kernel software failures. A restart serves to reinitialize the kernel while preserving the user state. If the failure was caused by a single event, or by circumstances that are irreproducible, the system recovers and continues to run. While the KeyKOS kernel is highly reliable, the failures that have been observed nearly always fall into this category. To increase system reliability and integrity, the kernel performs a thorough consistency check of all its data structures. This check is performed at the beginning of each checkpoint, to reduce the possibility of checkpointing an erroneous system state.

Because the important state of a KeyKOS system is held not just in files, but in a complex network of objects and capabilities, it is particularly important to ensure that the entire state of the system can be safely recovered under any conditions. In the 17-year history of KeyKOS, the only cases in which a damaged system state was checkpointed occurred as a result of kernel development and testing. The KeyKOS kernel is now a mature technology, and failures of this type have not been seen for years. There has never been a need for a data recovery utility such as UNIX's fsck. For off-site backup and disaster recovery, tape checkpoints are recommended.

8.2: Upgrading software

A significant benefit of the KeyKOS architecture is the ease with which the kernel software can be upgraded. The new kernel can simply restart using the data checkpointed by the old kernel.

The other side of this coin is that user software upgrades must be managed more carefully in KeyKOS than in less reliable architectures. Non-kernel software does not stop running when the system is rebooted. It requires explicit action to stop such software. Furthermore, the convenience of storing permanent object data in virtual memory structures or even process registers must be weighed against the need to separate the program from its data when it is desired to upgrade the program while preserving the data. In the applications where KeyKOS has been used, we have found fairly simple upgrade strategies to be adequate, but they must be planned in advance.

8.3: Performance characteristics

Because data in files is ultimately stored in pages of virtual memory, and the virtual memory manager keeps recently used data in main storage, KeyKOS achieves file I/O performance that is often spectacular compared with systems which do not cache disk file I/O. For example, the KeyKOS UNIX prototype can execute an exec()
system call (which loads a program from a file) eleven times faster than MACH 2.5 on the same hardware (an Omron Luna 88K) [1].

Transaction processing performance benefits from the KeyKOS architecture. In one benchmark, transaction processing speed was at least five times faster than comparable systems [6]. Under load, the KeyKOS commit/abort facility will commit more than one transaction with a single journal write, reducing the average number of disk I/O operations per transaction. CICS, for example, is unable to commit multiple transactions in a single write.

On the other hand, two issues present themselves as potential performance problems; the interruption of service at the time of checkpoint, and the disk I/O load presented by migration. At the beginning of a checkpoint, all processes are stopped while changed pages are noted and changed nodes are copied. The consistency check is also run at this time. The period of time in which processes are stopped is typically one second on a loaded system. This time is relatively independent of the host hardware, because faster machines typically have larger memories to be checked. The majority of this time is devoted to the consistency check. Depending on the relative importance of integrity and consistent performance, the consistency check could be eliminated. There is also a short period of reduced performance after processes are restarted, as copy-on-write pages are modified. A checkpoint generally will not be noticed by a developer editing at a terminal or by a point-of-sale user of a transaction processing system. However, checkpointing would be incompatible with most real-time requirements.

When a page is swapped out in KeyKOS, it is first written to a swap area; then, during migration, it is read back in, and written to the home location. This has the potential for tripling the number of I/O operations compared with systems which simply swap out the page. If the page is mirrored, as pages usually are, even more I/O is needed. In practice this has not been a problem. First, disk I/O bandwidth is greatly improved in KeyKOS, often by an order of magnitude, for reasons mentioned above. Second, pages which are frequently changed do not need to be migrated, as discussed above. Finally, migration I/O usually takes place at low priority.

9.1: Acknowledgements

Norman Hardy and William S. Frantz, along with the author, developed the KeyKOS checkpoint/restart software from conception to implementation, at Key Logic. Alan Bomberger contributed to a draft of this paper.

10: References

[4] DOS System Control and Service, IBM Corp., GC24-5036
[5] IBM System 360 Operating System Supervisor and Data Management Services, IBM Corp., C21-6646