Seamless Kernel Updates

by

Maxim Siniavine

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Seamless Kernel Updates

Maxim Siniavine

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2012

Kernel patches are frequently released to fix security vulnerabilities and bugs. However, users and system administrators often delay installing these updates because they require a system reboot, which results in disruption of service and the loss of application state. Unfortunately, the longer an out-of-date system remains operational, the higher is the likelihood of a system being exploited.

Approaches, such as dynamic patching and hot swapping, have been proposed for updating the kernel. All of them either limit the types of updates that are supported, or require significant programming effort to manage.

We have designed a system that checkpoints application-visible state, updates the kernel, and restores the application state. By checkpointing high-level state, our system no longer depends on the precise implementation of a patch and can apply all backward compatible patches. The results show that updates to major kernel releases can be applied with minimal changes.

Acknowledgements

I would like to express sincere gratitude to my advisor Prof. Ashvin Goel for his patience and invaluable advice. His guidance helped me all the time during research and writing of this thesis. I would also like to extend my thanks to all the committee members: Professor Angela Brown, Professor Michael Stumm and Professor Raviraj Adve for their insightful comments and hard questions. I thank my fellow graduate students Vladan Djeric, Zoe Chow, Isaac Good, Stan Kvasov for providing an intellectually stimulating and supportive environment. Finally I would like to thank my family for their encouragement and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Operating system maintainers release kernel patches regularly to fix security vulnerabilities and bugs, and to add features. However, users and system administrators often delay installing these updates because they require a system reboot, which results in disruption of service and the loss of application state. For example, updating the operating system for a game server requires all client users to stop playing the game, wait for the server to come back up, login to the server, and then generally play the game from the beginning, which is especially annoying for shooter and other real-time games.

Today, operating systems are updated infrequently because the updates have to be managed carefully. However, the longer an out-of-date system remains operational, the higher is the risk of a bug being triggered, or a system being exploited, since most exploits target existing vulnerabilities. In addition, users are unable to use the new features, e.g., performance optimizations, available in the kernel updates.

Realizing these problems, application programmers are increasingly designing programs that can be updated without significant disruption. For example, users of web applications are not aware when it is updated and can start using the new version automatically after they reload the page. In fact, users generally have no control over updates, which helps avoid the need to support several application versions. Similarly, many large

applications save and restore their state on an update (e.g., browsers restore web page tabs), thereby reducing disruption. Operating system kernels are the major remaining component of the software stack that require significant time for updates and lose state after an update.

Existing kernel update systems work at varying granularity. Dynamic patching performs updates at function granularity [6, 3], and hot swapping at object or module granularity [17, 4]. These techniques require significant programmer effort for implementing patch, object or module-specific state transfer functions that synchronize the state of an updated component with an existing component. For example, hot patching operates at function granularity and can be applied relatively easily to patches that only change code. However, carefully crafted state transfer functions are needed for patching updated data structures. Similarly, object and module granularity update systems require component-specific transfer functions for the updated stateful components, and must be designed to handle changes to the component interfaces [4].

None of these techniques handle cross-cutting changes due to major restructuring of code that occurs across major kernel revisions. For example, the Linux kernel is updated on average with five patches every hour, and developers release a major kernel release every 2-3 months [10]. Later in the paper, we show that each of these releases often consist of over a million lines of modified or new code. Requiring programmers to write state transfer functions for each of their patches or modules is simply impractical, especially when kernel patches occur so frequently and major revisions involve millions of lines of code.

Our goal is to reliably install major kernel updates, with minimal programmer effort, and without requiring user intervention or any changes to applications. The main insight is that applying updates at a courser granularity reduces the required programming effort. At higher level of abstraction implementation details are hidden which reduces the need to write state transfer functions for each patch. Say that a transfer function

exists for a stateful module. A patch that changes module internal state will not require an additional transfer function because this state is not exposed, making the patch easier to apply. For example, Swift et al. update device drivers at multi-module granularity by using common driver interfaces to automatically capture and transfer state between driver versions [19, 20].

Taking this idea to the limit, we have designed a system for the Linux kernel that checkpoints application-visible state, reboots and updates the entire kernel, and restores the application state. The checkpointed state consists of information exposed by the kernel to applications via system calls, such as memory layout and open files, and via the network, such as network protocol state. Our update system requires the least amount of additional programmer effort for installing a patch, because it hides most kernel implementation details, including interfaces between the kernel components. Furthermore, the kernel and the applications are strongly isolated from each other by memory management hardware and communicate by passing messages, i.e., system calls. As a result, there is no need to detect and update references from old to new data structures, or determine when this update process can terminate, which poses challenges in dynamic patching systems. Another significant benefit is that our system can handle all backward compatible patches because they do not affect application-visible state. Kernel patches generally provide such compatibility to minimize disruption. The main drawback of rebooting the kernel is that it is human perceptible, but we believe that the main impediment to applying updates today is loss of application state, rather than brief system unavailability.

Our focus on designing a reliable and practical update system raises several challenges. Ensuring that the system will restore applications reliably requires taking a consistent checkpoint. When kernel data structures are inconsistent, e.g., when a system call is in progress, a consistent checkpoint cannot be taken. Waiting for system calls to finish is unreasonable since many system calls can block in the kernel indefinitely. A third

solution is to interrupt system calls, but many applications are not designed to handle interrupted calls. Unlike dynamic patching and hot swapping methods, our solution guarantees quiescence, allowing consistent checkpoints to be taken for all updates. For system calls, we start with the POSIX specification for restarting system calls when a signal occurs, and provide a method for resuming system calls transparent to applications.

A practical system should require minimal programmer effort for applying kernel updates. To achieve this goal, the checkpoint format and the checkpoint/restore procedures must be made as independent of the kernel implementation as possible. We checkpoint data in the same format as exposed by the system call API and the network protocols. Both are standardized, and so our checkpoint format is independent of the kernel version, and we expect it to evolve slowly over time. An additional benefit of this approach is that we can use existing kernel functionality to convert the data to and from the kernel to the checkpoint, since this functionality is already needed to perform these conversions during system calls. When the kernel is updated, the updated functions will perform the conversion correctly. To minimize changes to the checkpoint procedures, we use kernel API functions as far as possible. These include system call functions and functions exported to kernel modules, both of which evolve slower than internal kernel functions.

This work makes three contributions. First, we design a reliable and practical kernel update procedure that allows taking a consistent checkpoint for all kernel updates, and requires minimal programmer effort for applying these updates. Second, we perform a detailed analysis of the effort needed to support updates across major kernel releases, representing more than a year and a half of changes to the kernel. During this time, six million lines of code were changed in 23,000 kernel files. We are not aware of any system that provides such extensive support for kernel updates. Finally, we evaluate our implementation and show that it works seamlessly for several, large, real-world applications, with no perceivable performance overhead, and without requiring any application modifications. The overall reboot time is reduced by a factor of 4 to 10 for several of

these applications.

Section 2 discusses related work in this area. Section 3 presents our approach and Section 4 describes the implementation of our system. Section 5 presents our analysis of the programmer effort needed to use our system and evaluates the performance of the system. Section 6 summarizes our work and provides conclusions.

Chapter 2

Related Work

Closest in goals to this work, Autopod [16, 15] uses a virtualization layer to decouple processes from their dependencies on the underlying operating system. Virtualization layer intercepts system calls and rewrites their inputs and outputs to enable consistency in the resource identifiers seen by the applications. The system calls are modified so that the identifiers they use, remain consistent on different machines after migration, to prevent conflicts with existing resources, and to isolate migrated processes. The virtualization layer creates a POD abstraction which gives a group of processes a private namespace for resource IDs and a private file systems.

The migration is performed by sending a SIGSTOP to all the processes in a POD to stop them, and then making a copy of all their resources, their memory and their private file systems. This copy is then transfered to another machine and the saved processes are once again started in another POD at the destination.

Similar to our system, Autopod uses a checkpoint-restart mechanism, with a high-level checkpoint format, for migrating processes across machines running different kernel versions. The focus on migration has several consequences. First, virtualization introduces performance overheads, and the virtualization layer itself needs to be maintained to keep up with kernel changes. This layer is not needed in our system, designed purely for

kernel updates. Second, the checkpoint requires copying all memory pages, making the checkpoint much larger than required in our system. Third, Autopod requires migrating state across machines, which may not be a viable option for desktop environments and stateful servers such as databases running on large local disks. Furthermore, Autopod exposes applications to interrupted system calls, making applications susceptible to crashes and data loss. By migrating state across machines that are already running operating systems, Autopod does not seem to address quiescence issues such as caused by non-interruptible system calls and interrupts. Finally, we evaluate our system across major kernel updates and provide a detailed analysis of our checkpoint code and format, and all the code changes required.

Otherworld [9] is designed to recover from kernel failures. In addition to the main kernel it maintains a secondary crash kernel. When a fault is detected in the main kernel, execution transfers to the crash kernel. The crash kernel initializes itself and then examines the data structures of the main kernel. Using the main kernel's data structures, the crash kernel attempts recover the state of the running applications and to resurrect them, so they can function after the crash.

The design of Otherworld has inspired this work, but our goals are different, namely applying kernel updates reliably. After a kernel failure, Otherworld does not have the liberty to achieve quiescence. For example, a thread may have acquired a lock and partially updated a critical kernel data structure when a crash occurs, making the checkpoint and resurrection process failure prone. Similarly, as we show later, restarting system calls transparently without achieving quiescence is impractical. Given that the kernel has crashed, a best-effort resurrection process is acceptable, while our aim is to make the update process as reliable as possible. Also, we support sharing of kernel resources between threads, do not require any modifications to applications, and evaluate the feasibility of the approach for kernel updates.

Several researchers have proposed dynamic patching at function granularity for ap-

plying kernel updates [6, 13, 3]. LUCOS [6] uses the Xen VMM to stop and dynamically update functions in Linux. When a kernel data structure is updated, LUCOS maintains both versions in memory, until it can determine that the old version is no longer in use. This quiescence step requiring walking the stack of every kernel thread looking for updated code. However, this assumes that code that accesses an updated data structure must have been updated. Programmers have to analyze a Linux patch, decide whether any data structures have been updated, and then write a LUCOS-specific state transfer function, a tedious engineering effort [6]. LUCOS uses page protection and the transfer function to maintain coherence between the data structure versions.

DynAMOS [13] patches functions, similar to LUCOS, after disabling interrupts. DynAMOS supports updates to non-quiescent functions as well as changes to the data structures. Non-quiescent functions are updated via a series of transformations to the code which allows both old and new versions of the same function to exist at the same time. As the old instances terminate they gradually get replaced by the new updated code. Adding new fields to data structures are done using shadow structures. New field is added to the shadow structure which is maintained along side the original data structures, and all the users of the new field are changed to access the shadow structure. Kernel threads are updated when they enter a sleep state. When a thread enters sleep it is terminated and the new threads running updated code is started instead.

Ksplice [3] generates and applies binary function patches for security updates, while requiring minimal or no programmer effort. It inspects binary object files to detect the changes to the code and automatically generate patches from those changes. To apply the patch it overwrites the memory if the running kernel to insert new updated functions and remove old ones. Security patches tend to be localized, generally only changing code, and hence the approach works well for this domain.

At the application level, several systems use compiler support for updating programs [7, 14]. Polus [7] finds all changed types, global variables and modified functions

by comparing the syntax tree of both the old and the new versions of files, and builds relations between the changed variables and functions to generate patches. Then, it applies techniques similar to LUCOS for patch injection. Ginseng [14] introduces sufficient room in data types to update them with new fields in the future. It also adds indirections for types and functions, to provide type-safe updates for C code. Using these techniques for kernels is challenging because of the frequent use of type unsafe code, including assembly code, as well as optimized and inline functions for which it is hard to ensure safety guarantees. For example, Polus is unable to generate patches correctly in the presence of pointer aliases and void pointer casts.

The K42 operating system is designed to allow hot swapping and updating components at the object and module granularity [17]. In K42 function calls are made through an indirection table. Updating the entry in the indirection table allows all the callers to automatically start using the new version of the function. Issues with quiescence are handled by restructuring the kernel so that all kernel threads handle requests quickly in a non-blocking manner. By restructuring the kernel K42 supports data structure updates within modules as well as updates to module interfaces [4]. However, it cannot update code outside its object system including low-level exception-handling code, parts of the scheduler, and its message-passing IPC mechanism. More importantly, state transfer functions need to be written manually for the updated objects and modules.

Updating the entire kernel has several other advantages over dynamic patching and hot swapping systems, including handling updates to boot code and non-quiescent code such as long-running kernel threads, and fixing non-critical failures such as memory leaks.

Several systems use checkpoints for recovering from failures in applications or kernel components. Linux-CR [11] aims to add a general-purpose checkpoint and restart mechanism to the Linux kernel. It relies on the Linux containers to isolate the processes that are meant to be checkpointed from others running on the system, and give them a private namespace for resource identifiers. To stop the processes in a quiescent state

Linux-CR uses freezer control groups. Freezer control allows to stop all the processes in the control group in the same way as sending them the SIGSTOP signal, but silently, without making signal visible to the applications. The Linux-CR addresses the difficulty of keeping the checkpoint code synchronized with kernel developments by keeping the generic code in its own subsystem and the code to save specific object types close to the native code for those objects. While it addresses the difficulty of maintaining the checkpoint code, it does not attempt to tackle the problem of migrating applications to new versions of the kernel.

CuriOS [8] recovers a failed service transparently to clients in a microkernel OS by isolating and checkpointing the client state associated with the service. Membrane [18] restarts failed file systems transparent to applications by using a lightweight logging and checkpoint mechanism.

Complementary to this work, virtual machines can be used to speed the kernel reboot process by running the existing and the updated kernel together [12]. Primary-backup schemes can be used for rolling kernel upgrades in high availability environments, but they require application-specific support [1, 2].

Chapter 3

Approach

Our goal is to perform seamless kernel updates, without requiring user intervention or any changes to applications. Figure 3.1 shows the timeline of events for regular updates and seamless updates. During a regular update, applications are closed manually after saving work, the kernel is rebooted, and then applications need to be restarted manually. Our seamless update approach is based on treating the entire kernel as a replaceable component [9].

Our update system operates in five steps as shown below and in Figure 3.1.

1. Load new kernel: When a kernel update arrives, we use the kexec facility in Linux to load the new kernel image and to start executing it. We modified kexec so that it performs the next two steps before starting the new kernel.

2. Wait for quiescence: We ensure that the kernel reaches quiescence as described in Section 3.2.

3. Save checkpoint: The checkpoint code walks the kernel data structures associated with application-visible state and converts them to a high-level format that is independent of the kernel version.

4. Initialize new kernel: The kexec jumps execution to the beginning of the new kernel,

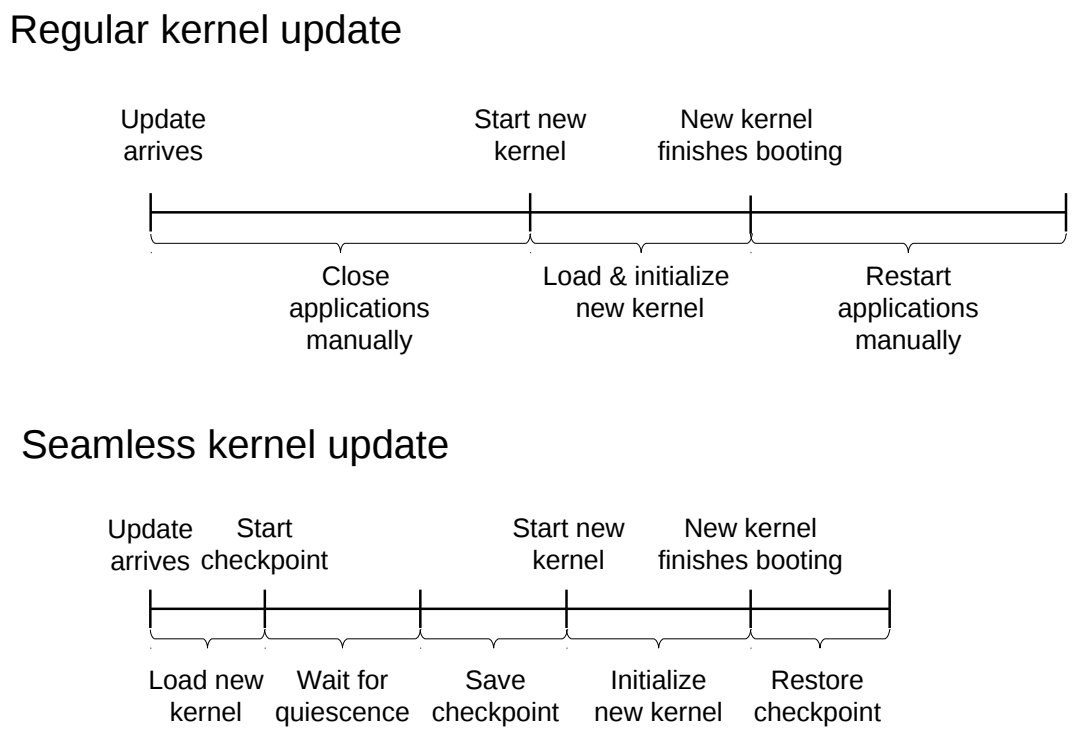

Figure 3.1: Timeline for regular and seamless kernel update

and the new kernel initializes itself.

5. Restore checkpoint: After kexec has initialized the new kernel, it reads the checkpoint and recreates applications using the checkpoint information. Then it restarts these applications which may require restarting blocked system calls as described in Section 3.3.

3.1 Implementation Overview

Our update system checkpoints and restores processes by capturing application-visible kernel state from kernel data structures. The Linux kernel stores all process related in-

formation in the task_struct data structure. This structure contains process information such as the PID of the process, the parent and the children of the process, scheduling parameters and accounting information. The task_structure contains pointers to other data structures that describe the resources currently being used by the process, such as memory management information, open files, etc. Thus the state of a process can be checkpointed by traversing the graph rooted at the task_struct associated with the process. Our checkpoint typically saves the fields in the data structures that are visible to applications through system calls. These fields also allow us to restore these data structures during the restore process.

The checkpoint information and the memory pages of all the processes need to be preserved during the reboot process. When the Linux kernel starts executing, it uses a bootstrap memory manager to dynamically allocate memory that is needed during the boot process before the memory management system has been initialized. After the bootstrap memory manager is initialized, we read the checkpoint and mark all the pages used by the checkpoint and the process pages as reserved so that the memory manager cannot immediately reuse these pages. After the restore operation, the process pages are marked as allocated and can be freed when a process terminates or as a result of demand paging.

When multiple processes share a resource, eg. a memory region, they keep pointers to the same structure, e.g., a memory region descriptor. We implement this sharing by saving each resource separately in the checkpoint, and using pointers within the checkpoint to indicate the sharing. The implementation tracks each encountered resource in a Save hash table. The key of this hash table is the memory address of the kernel resource, and the value is the memory address of the corresponding entry in the checkpoint. When checkpointing any resource, we check if its address exists in the Save hash table, and if so, we use the value in the Save hash table to create a pointer to the existing checkpoint entry. During restore, we create a Restore hash table with keys that are the values from

the Save hash table. As each resource is restored, its memory address is filled in as the value in the Restore hash table. Looking up the Restore hash table as each resource is created ensures that the resource sharing relationships are setup correctly.

The restore process runs with root privileges and hence care must be exercised when restoring the state of the OS resources. The kernel uses two types of credentials, one set for processes and another for files. We set the the various user and group IDs for each restored process thus ensuring that the restored process runs with the correct credentials. The restore process does not create files and hence we do not need to set up or modify any file credentials. However, there is one exception to this rule with listening Unix sockets which we describe later.

Our implementation currently checkpoints the following OS resources: 1) thread state, 2) memory state, 3) open files, 4) network sockets, 5) pipes, 6) Unix sockets, and 7) terminals. It also checkpoints the state of the following simple hardware devices: 1) framebuffer, 2) mouse, and 3) keyboard. Our aim has been to implement features that enable supporting as many commonly used applications as possible, and especially server-side applications. While we have tested several simple desktop programs using the Xfbdev X server, adding support for live driver updates would make our approach more compelling for updating device-rich client machines [19].

3.2 Quiescence

Ensuring that our system will restore applications reliably requires taking a consistent checkpoint, with two conditions: 1) all threads are stopped, and 2) the kernel data structures are consistent. The first ensures that thread state remains consistent with its checkpoint state. For example, if a thread continues execution during or after checkpointing, it can affect the state of other threads with which it shares any resources. When both these conditions are met, we say that the kernel is quiescent, and a checkpoint is

taken.

The first condition can be met easily by pausing all processors other than the one running the checkpoint thread.[1] For the second condition, data structures can be inconsistent when any kernel code is executing, including in system calls, exception handlers and interrupt handlers. We need to let all kernel code finish executing and stop further entry into the kernel. However, system calls and exception handlers can block or sleep while waiting for events. A thread can sleep in one of two states, uninterruptible sleep and interruptible sleep, in the Linux kernel. During an uninterruptible sleep, the thread can hold locks and modify data structures, so we need to let this code continue executing. Fortunately, an uninterruptible sleep is used where an operation is expected to take a relatively short time, such as a disk access for paging and memory allocation. Very long uninterruptible sleep is considered a bug and we do not expect to encounter it during normal operation. Recovery from this type of failure is beyond the scope of our project. Waiting for this code to finish executing does not significantly impact the overall update time because it is much faster than the time needed to initialize the new kernel.

A thread in an interruptible sleep can block indefinitely, e.g., waiting for user or network input. In this case, the kernel releases locks so that the thread does not block other threads from making progress, and ensures that its data structures are consistent before putting a thread in an interruptible sleep state. Only system call threads can sleep in an interruptible state, and they can be interrupted by sending a signal to the thread, in which case, the system call returns immediately with an EINTR error code. To avoid blocking indefinitely, we save the state of system calls blocked in an interruptible sleep (below, we will call these blocked calls) and handle them during restore, as described in Section 3.3.

We use the following steps to ensure that the kernel is quiescent before taking a

[1] The quiescence conditions don't apply to the checkpoint thread because it is not checkpointed and it doesn't modify any kernel data structures

checkpoint:

1. Stop other processors: We start the process by using the stop_machine function in Linux that allows waiting for interrupt handlers to finish executing on all processors, disables interrupts and pauses execution all other processors, and then returns control back to the calling processor where this code can continue running.

2. Quiesce user threads: In this step, we wait until *all* user threads are quiesced. If a user thread is currently in user mode or is blocked, then it is not running any kernel code, and we say that it is quiesced. When all user threads are quiesced, we can proceed to Step 3, otherwise we perform the following steps:

   (a) Disable all further system calls: We need to let threads running in the kernel or sleeping in uninterruptible sleep continue execution. By disabling further system calls, we can guarantee quiescence of user threads. If a thread issues a system call, we block it in a special blocked state, so that we know that it simply needs to be restarted on restore.

   (b) Wait briefly for quiescence: We enable the processors again by returning from stop_machine, wait for 20 ms, and restart the process by returning to step 1. While waiting, the kernel operates normally, allowing interrupts to occur and threads to exit the kernel or to block in interruptible sleep.

3. Quiesce kernel threads: In this step, we wait until *all* kernel threads are quiesced. Kernel threads are used to perform deferred processing tasks, such as writing dirty file buffers to disk. We separate quiescing user threads from quiescing kernel threads so that the existing deferred tasks can be completed before taking a checkpoint. Quiescing kernel threads proceeds in four steps:

   (a) Deferred processing: At this point, we have returned from the stop_machine function and all interrupts are enabled, and we can perform various deferred

processing tasks. We create hard links to temporary files and then save all dirty file buffers by calling the system-wide sync operation to complete any buffered IO, as described later in Section 4.1.3.

(b) Reboot notification: Then, we send a standard reboot notification to all kernel threads so that these threads can prepare for devices to be shutdown. In particular, after the notification returns, the threads do not access any devices. For example, fast devices, such as the hard drive, will not serve any further requests because users threads are quiescent and the kernel threads have been notified about the reboot in this step.

(c) Shutdown devices: Next, we shutdown all devices because many drivers assume upon startup that devices have previously been reset [5]. This shutdown process may wake up certain kernel threads (e.g., when the hard drive cache is flushed) and hence we cannot disable interrupts or suspend kernel threads until this point. However, since interrupts are enabled, certain slow devices, such as the keyboard, mouse, timer and the network may generate exogenous interrupts until the devices have been shutdown, but we prevent these interrupts from waking up sleeping user threads. Losing these exogenous interrupts will resemble a short system freeze, but without significantly affecting applications. For example, TCP packets will be retransmitted since we restore TCP state.

(d) Wait briefly for quiescence: After all the user threads are quiescent and all the devices have been shutdown, the kernel threads should not have any further work and they should not be running, i.e., they should be quiescent. At this point, we invoke stop_machine to disable interrupts again. For safety, we check if all the kernel threads are blocked, and if so, we can proceed to the next step. Otherwise, we return from stop_machine, wait for 20 ms, and repeat this step.

4. Take checkpoint: Now that the kernel is quiescent and all interrupts are disabled, we can checkpoint application-visible state. The checkpoint and restore process is described in Section 4. We ensure that the checkpoint accesses memory but not the disk, which has been shutdown in the previous step.

3.3 Restarting System Calls

After restoring the checkpoint in the new kernel, we need to resume thread execution, which requires handling system calls that were blocked. There are over 300 system calls in Linux, but only 57 of them are interruptible. [2] It is fortunate that we do not need to consider the uninterruptible calls because many of them modify kernel data structures and are not idempotent and thus are not easily restartable.

For the interruptible calls, a simple solution would be to return the EINTR error code to the user thread, since this return value is part of the specification of what happens when a signal is sent to the thread. However, most applications do not handle interrupted system calls, specially if they don't use signals.

Instead, we reissue the system calls that were blocked after the application is restored. To ensure correct behavior, we looked at the POSIX specification for restarting system calls upon a signal, since this specification is implemented by the Linux kernel. The kernel can already automatically restart some system calls when they are interrupted by a signal. It can restart these calls because they are idempotent when they are blocked, even if they are not idempotent otherwise. The system calls block in order to wait for external events. However, if the event has not occurred, then the system call has not done any work, and so it can be safely reissued. The POSIX specification disallows restarting some of the system calls on a signal, for two reasons: 1) a timeout is associated with the system call, or 2) the system call is used to wait for signals. In our case, we still

[2] We found this number by manual analysis, and by cross correlating with the manual pages of the calls. Of the 57, several are variants of each other, such as the 32 and 64 bit versions of the call.

wish to restart these calls to avoid failing the application. For timeout related calls, they can be reissued after taking the timeout period into account, as discussed below. For signal related calls, they can be restarted, because a signal was never delivered in our system. However, they require adjusting the signal mask, as described below. We use five methods to transparently resume all the blocked system calls after restoring a user thread. A few system calls require multiple methods.

1. Restart: The system calls that are idempotent when they are blocked are restarted. This is exactly the same behavior the kernel already implements for these system calls when they are interrupted by a signal. Examples are open, wait and its variants, futex, socket calls such as accept, connect, etc. There are 19 such restartable calls.

2. Track progress: System calls that perform IO operations like read and write keep track of how much progress they have made. When these calls are interrupted by a signal, their current progress is returned to the user. We implement the same behavior, and after restoring the thread, we return the progress that the system call had made before the checkpoint was taken. We do not reissue this call to complete the operation (e.g., finish a partial read) because input may never arrive. It may appear that returning a partially-completed IO operation may cause certain applications to malfunction. However, system call semantics require correctly designed applications to handle short reads and writes. For example, on a read, fewer bytes may be available than requested because the read is close to end-of-file, or because the read is from a pipe or a terminal. Similarly, network applications communicate with messages of predetermined length and reissue the calls until the full message is processed [21]. As a result, we have not observed any problems with returning partial results for reads and writes for the applications that we have tested. When no progress has occurred, we restart the system call, because returning a zero indicates that the communication has terminated (EOF) after which the application

may fail, when in fact our checkpoint maintains the communication channel. In this case, we can still safely reissue the call since it had not made any progress before being blocked. There are 23 calls that require progress tracking.

3. Return success: System calls that close file descriptors like close or dup2 invalidate the descriptors if they block. For these calls, we return success because when the checkpoint is taken, the file descriptor is already invalidated and the resource will be reset once the kernel is restarted. There are 3 such calls.

4. Update timeout: If the system call has a timeout associated with it, e.g., select, and it uses a short timeout compared to the time it takes to restart the kernel, we simply reissue the system call to avoid returning a spurious timeout. For long timeouts, we restart the system call after calculating the remaining time and subtracting the total time it took for the kernel to reboot and restore the thread. There are 11 calls that require timeout handling.

5. Undo modifications: Certain system calls, like pselect and ppoll, make a copy of the process signal mask, and then temporarily modify it. Before restarting these system calls, the signal mask has to be restored from the copy to the original state. The pselect and ppoll calls also require timeout handling. There are 7 calls that require undo modifications.

3.4 Checkpoint Format and Code

We checkpoint application-visible state, consisting of information exposed by the kernel to applications via system calls, such as memory layout and open files, and via the network, such as protocol state for network protocols implemented in the kernel. Checkpointing this state requires programmer effort proportional to the system call API rather than the size of the kernel implementation or the number of kernel updates. Furthermore, since

Structure Fields	**Notes**
In checkpoint	
vm_begin, vm_end	Region of address space controlled by this vm_area_struct
vm_page_prot	Address space is writable, readable or executable
vm_flags	Special attributes: for example direction the stack grows
vm_file	Name of the file mapped by a vm_area_struct
vm_pgoff	Offset from the beginning of the file
vm_private	Used for mmap operations
anon_vm	Specifies the type of reverse mapping used
Not in checkpoint	
mm_struct	Pointer to memory descriptor
vm_next	Pointer to the next vm_area of the process
vm_rb	Tree node used to find vm_area based on virtual address
vm_ops	Pointer to functions operating on vm_area_struct
vm_set, prio_tree_node	Used to implement reverse mapping
anon_vma_node, anon_vm	Used to implement reverse mapping

Table 3.1: Analysis of vm_area_struct

the system call API and the network protocols are standardized and change relatively slowly over time, we expect that a carefully designed checkpoint format will evolve slowly.

Our approach raises several issues: 1) what state should be saved, 2) the format in which it should be saved, 3) how sharing relationships between threads and their resources are expressed, and 4) how the code should be implemented. We save information available to the user space through system calls and via special filesystems like /proc and /sysfs. We also save network protocol state, including buffered data, to ensure that a kernel update is transparent to network peers. For example, we store port numbers, sequence numbers and the contents of the retransmit queue for the TCP protocol.

The checkpoint consists of a list of entries, representing either a thread or a resource owned by the thread, such as open files and sockets, with each resource using a unique format. As an example, Table 3.1 shows all the fields in the vm_area_struct structure in the kernel and the fields that are saved in our checkpoint. This structure represents a region of an application's address space, and the fields saved in our checkpoint are exposed to applications via the smaps file in the /proc file system or when accessing memory. For example, this information determines whether a memory access will cause an exception or a memory mapped file to be read from disk. The data in the checkpoint allows recreating the virtual memory region correctly, while the rest of the fields relate to the data structures used to implement the regions. The implementation dependent fields are not visible to the user, and thus not included in the checkpoint. We expect that while these fields may change (and have changed) over time, the checkpoint fields are unlikely to change significantly for backward compatibility.

Since we are saving state visible at the system call API, we save it in the same format. Internally, the kernel may store this state in any implementation-dependent way, but it needs to convert it when communicating with user applications. For example, a file path is a string in user space, but the kernel represents it by a sequence of dentry, qstr and mnt_point structures. By using a string for a file, we expect that the checkpoint will not depend on the kernel version, and we can use existing kernel functions to convert to the correct implementation-dependent kernel versions of the file-related structures. For example, the do_filp_open function will convert a path name to a file descriptor, and since it is the same function used to implement the open system call, we expect it to perform any implementation dependent work required when opening a file. Besides various data structures that are stored in the checkpoint, as described later in Section 5.1, we also need to store the virtual memory state for each thread. To reuse existing pages and page tables, we only explicitly store the user-visible contents of the per-thread global page directory in the checkpoint, and we ensure that the new kernel does not clobber any

user-level pages and page tables. To do so, we also need to store the physical address of every used page.

We represent sharing of resources with pointer relationships in the checkpoint. We use a single hash table to represent sharing of all resources between threads during checkpointing. The address of a resource is used as the key, and the address of its corresponding checkpoint entry as the value, which makes it simple to set up the pointer relationships in the checkpoint. During restore, we use a similar hash table, but the key and value are inverted, so that the address of the checkpoint entry is the key, and the address of the restored resource is used as the value, which makes it simple to assign a pointer to a shared recreated resource.

Beside a portable checkpoint format, the checkpoint code must also be easy to port across different kernel versions for our update system to be practical. Ideally, the checkpoint mechanism would be implemented entirely in user space, relying only on the stable system call API. Unfortunately, some of the required functionality, such as page table information, and resource sharing relationships are only available in the kernel. Our code mostly uses functions exported to kernel modules, which evolve slower than internal kernel functions. We use as high-level functions available in the kernel as possible for saving and restoring state. For example to restore a pipe between two processes we call a high-level function do_pipe_flags which performs all the implementation dependent work needed to create a pipe. Afterwards, we use another high-level function to assign the newly created pipe to the two processes we are restoring. The high-level API takes care of all the details involved with maintaining the file descriptor tables of the two processes. Also, updates to the implementation of these functions will not affect our code. We also do not rely on any virtualization or any indirection mechanism,which would itself need to be maintained across kernel updates. Section 5.1 analyzes our checkpoint format and code in more detail.

Chapter 4

Implementation

This section describes the implementation of our kernel update system. First, we describe the process checkpoint-restore mechanism used for updating kernels. Then, we present the interface to our update system that can be used by system-level utilities to interact with our system. Finally, we describe the limitations of our current implementation and methods for addressing the limitations.

4.1 Implementation of checkpoint and restore

The checkpoint save operation involves saving data structure values and is relatively simple. The restore operation is more complicated because it requires recreating custom processes from the checkpoint information, similar to creating the initial user process. Hence, much of the description below focuses on the restore operation.

4.1.1 Threads

The stop_machine kernel function used for quiescing the system (see Section 3.2). It schedules a thread on each CPU and each thread disables interrupts. This process waits until all other threads go through a context switch. At which point, the kernel stores the

register values and segment descriptors table entries in the thread_struct structure. We store this context switch data in our checkpoint. To restore a thread, we spawn a kernel thread for each thread stored in the checkpoint. Within the context of each spawned thread, we invoke a function, that we created, similar to the execve system call. The execve system call replaces the state of the calling process with a new process whose state is obtained by reading an executable file from disk. Our function converts the kernel thread into a user thread by loading the state from the checkpoint in memory. We restore the saved register values and segment descriptors for the thread so that the new kernel's context switch code can use these values to resume thread execution.

We restore the saved task_struct fields and reestablish the parent-child process hierarchy by changing the parent and real_parent pointers so that they point to their restored parents. We also make sure that all restored children are added to the list of children associated with their parent, and for multi-threaded processes we add threads to their thread group lists. After this setup, the kernel starts identifying the spawned kernel thread as a regular user process.

4.1.2 Address Space

An address space consists of a set of memory mapping regions and page tables. Each memory mapping region describes a region of virtual addresses and stores information about the mapping such as protection bits and the backing store. The page table stores the mapping from virtual pages to physical pages. Currently, our implementation supports the x86 architecture where the page table structure is specified by the hardware and thus will not change across kernel versions.

Linux manages memory mapping regions using a top-level memory descriptor data structure (mm_struct) and one or more memory region descriptors (vm_area_struct). We store various fields associated with these data structures, including the start and end addresses of each memory region, protection flags, whether or not the region is

anonymous, and the backing store, as shown in Figure 3.1. For memory mapped files, we store the file name and the offset of the file for the virtual memory region. We restore these data structures by using the same functions that the kernel uses for allocating them during the execve (mm_struct), mmap (vm_area_struct) and mprotect system calls. These functions allow us to handle both anonymous regions and memory-mapped files. For example, we restore a memory-mapped file region by reopening the backing file and mapping it to the address associated with the region. The memory region structures can be shared and we handle any such sharing as described earlier.

The x86 architecture uses multi-level pages tables, and the top-level page table is called the page table directory. This page table format will not change across kernel versions and so we do not copy page tables or user pages during the checkpoint and restore. As a result, a process accesses the same page tables and physical pages before and after the kernel update. However, one complication with restoring page tables is that the Linux kernel executes in the address space context of the current user thread, and it is mapped at the top of the virtual address space of all processes. The corresponding page table entries for each process need to be updated after the kernel update. These page table entries are located in the page table directory. The function that creates the memory descriptor data structure (mm_struct) also initializes the page table directory with the appropriate kernel page table entries. We initialize the rest of this new page table directory from its pre-reboot version and then release the latter. At this point, we notify the memory manager to switch all process pages from being reserved to allocated to the new process.

4.1.3 Files

The Linux kernel uses three main data structures to track the files being used by a process. The top-level fs_struct structure keeps track of the current working directory and the root directory of a process (the root directory can be changed with the chroot

system call). The file descriptor table contains the list of allocated file descriptors and the corresponding open files. Finally, the file descriptor structure stores information about each open file. All three structures can be independently shared between several processes. For example, two processes might share the same working directory, may have different file descriptor tables, and yet share one or more opened files.

For each process, we store its root and current working directory, list of open file descriptors, and information about open files. Linux stores the current root and current working directory of a process as dentry structures. In the checkpoint, we store them as full path names. For files, we store its full path, inode number, current file position, access flags, and file type. The full path of each file is obtained by traversing the linked list of dentry structures. Each file structure has a pointer to a dentry structure, which stores the file name. In turn, each dentry structure has a pointer to another dentry which stores the name of the parent directory. For non-regular files (e.g., sockets, terminals), we store additional data needed to restore them, as discussed in later sections.

When restoring each process, we call chroot to restore the current root and chdir to restore the current working directory. Restoring open files requires calling functions that together implement the open and dup system calls. We do not use these system calls directly because our code has to be flexible enough to handle restoring shared data structures. For example, when the entire file descriptor table is shared between threads (or processes), once the table is setup for a thread, files do not need to be opened or duped in the second thread. To restore an open file, we first call a function that creates a file descriptor structure. Then we open the file using the flags, such as read/write, non-blocking I/O, etc., that were saved in the checkpoint. Next, we use the lseek system call to set the current file position. Then we dup the file descriptor so that it uses the correct descriptor number, and finally, we install this descriptor in the file descriptor table.

Temporary files require some additional steps before they can be restored. A tem-

porary file is created when unlink is called on a file in use (an open file) and the link count of the file reaches zero. In this case, all directory entries referencing the file have been removed and the file cannot be opened again. However, the contents of the file exist until all applications using the file release their reference to it, at which point the file is deleted permanently. If a system crash occurs, temporary files need to be removed (garbage collected) since they are not accessible in the system (no directory entries point to these files and the processes that were using the file before the crash do not exist any longer). File systems remove temporary files (when an open file's link count reaches zero) by adding a reference to the file to an orphan list that is kept on disk. At mount time, the orphan list is traversed and the temporary files are deleted. We do not want the temporary files to be deleted so that the restored processes can continue using these files. To do so, we create a hardlink to the temporary file when taking the checkpoint. We use a function used by the hardlink system call to create a link to the file, but instead of using the file name we use the inode of the temporary file as the source of the link. We perform this step as part of deferred processing when the interrupts are enabled and the disk can be accessed (See Section 3.2). During restore, a temporary file is handled similar to a regular file, except that after it is opened, we invoke the unlink system call to remove the reference to the file from the filesystem.

We ensure that all dirty file system buffers are committed to disk by calling the file-system wide sync operation as part of deferred kernel processing, as described in Section 3.2. As a result, we do not need to save and restore the contents of the file-system buffer cache.

4.1.4 Network Sockets

A network socket provides the interface to the different Internet protocols supported by the Linux kernel. Our update system currently supports UDP and TCP protocols because these are the most common protocols used by applications. Other types of network

protocols such as ICMP and raw IP sockets are typically used by utility applications and have little state. As a result, they do not benefit from our update approach and were not implemented. The socket interface allows reading and writing packets using the read and write system calls similar to files but it also provides operations such as bind, connect and accept. The kernel represents sockets by file descriptors that store protocol state associated with the socket.

Network applications and protocols must already handle network failures that cause packets to be lost, duplicated or re-ordered. We rely on this behavior to simplify restoring network connections. Once the application (and/or protocol) state is restored, the application can handle any problems that arise as a result of packets being dropped during the update. For example, TCP handles lost packets using retransmissions transparent to TCP applications. From an application's perspective, the update process will seem like a temporary network delay.

4.1.4.1 UDP Sockets

UDP is a stateless protocol for sending messages over the network. It does not provide reliability, integrity or ordering, so this makes restoring UDP sockets straight forward. When creating a checkpoint, we store the source and destination IP addresses and port numbers of the socket. To restore a UDP socket, we call the socket function to create the socket descriptor and then optionally call the bind function to assign a port number to the socket. We discard any sent or received data that was still being processed by the kernel and let the application handle packet loss. We rely on user utilities to set up the IP address of the machine and the routing tables so that network communication is possible after the update.

4.1.4.2 TCP Sockets

The TCP protocol provides a reliable, stream-oriented network connection to socket endpoints. It guarantees delivery without packet loss, reordering or duplication. TCP runs above the unreliable IP protocol and makes these guarantees by requiring the receiver to send acknowledgments for the data it has received to the sender. The sender assigns a sequence number to each byte of data that it has sent. Depending on the received acknowledgments, it decides to transmit the next packet in the sequence or it retransmits already sent packets. The sender also keeps a timer for each packet sent and retransmits packets if an acknowledgment is not received for the packet before the timer expires or if it receives too many duplicate acknowledgments. TCP is bidirectional so each socket acts as both a sender and a receiver.

Internally, when an application writes to a TCP socket, the user data is either split or combined (Nagle algorithm) into segments of a certain size (e.g., maximum segment size). After each segment is created it is added to the sender's write queue. TCP adds a header to each segment in the write queue and passes it on to the lower layer protocol (IP) that transmits the segment on the network. After the segment is transmitted, it is moved to a retransmit queue in case it is needed for retransmission. A segment is taken off the queue after an acknowledgment is received.

There are two types of TCP sockets, listen and communication sockets. Listen sockets wait for incoming connection attempts and perform the TCP three-way handshake to establish a connection. For these sockets, we save the local address and the port the socket was listening on, as well as the maximum number of pending connections. This state is specified by applications when creating sockets using the system call interface. Restoring a listen socket involves issuing the same system calls as needed to create the listen socket. A listening Unix socket creates a file name for the socket, but this name does not get removed on a kernel reboot. We delete the name during restore, and the socket restore code recreates this name. This file is owned by the restore process (root)

and we use chown system call to change the ownership to the original owner.

Communication sockets are used to transmit data. These sockets maintain much more state than the listen sockets. Some of this state, such as the last received sequence number, acknowledged sequence number and the packets in the write and retransmit queues, must be checkpointed because it is essential for correct TCP operation. Failing to restore such state will result in lost data and termination of the restored TCP connection. Other TCP state, such as the congestion avoidance state, is performance related and does not need to be checkpointed for correct TCP operation. This state can be reset to default values and the TCP implementation will automatically adjust them to reflect the network conditions. However, in some cases, ignoring the performance related state has a significant impact on TCP throughput as discussed below.

When restoring a connection, we allocate the socket descriptor and then restore the connection state. This state consists of the source and destination addresses, port numbers, and sequence numbers of the received and sent data. Then we restore the contents of the write and retransmit queues while preserving TCP headers so that segmentation does not need to be performed again.

At this point, we can resume sending packets. However, which packet should be sent initially? The sending side maintains two counters, *snd.una*, the first unacknowledged sequence number (sender knows that all octets or bytes smaller than this sequence number are acknowledged and does not keep them in its retransmit queue), and *snd.nxt*, the next sequence number that the sender should send. Similarly, the receiving side maintains a counter, *rcv.nxt*, the next sequence number expected on an incoming segment. Initially, we started sending packets starting from the *snd.una* sequence number because the packets between *snd.una* and *snd.nxt* may not have arrived at the receiver during the update and this would quickly perform retransmissions. However, this implementation would deadlock occasionally. Upon inspection, we found that this deadlock would occur when $rcv.nxt > snd.una$. In this case, the receiver had acknowledged packets between *snd.una*

and *rcv.nxt*, but these acknowledgments had been dropped during the update. As a result, the sending side would observe acknowledgments for packets that it believed that it had not yet sent. Linux kernel versions 2.6.28 and 2.6.29 discard these future acknowledgments and our TCP connection would make no further progress at this point (later Linux versions accept future acknowledgments). To fix this problem, we started sending packets from the *snd.nxt* sequence number, which is the same packet that TCP would have sent before the update. The next acknowledgment would trigger retransmissions if packets had been lost during the update.

During normal operation, TCP relies on incoming acknowledgments to advance its transmit window and increase the window size. However, if acknowledgments are dropped during the update, it may take a long time before they are retransmitted again. The long retransmission timeout at the receiver becomes an issue when the flow control or the congestion control window is so small that it does not allow the sender to send any packets. To jump start the sending process, we set these windows so that at least one packet can be sent after the connection is restored. After this packet is received, the receiver sends an acknowledgment in response, which resumes communication quickly.

We found that TCP throughput declines significantly (by almost 40%) if the timestamp and selective acknowledgment extensions are disabled. So even though they are not essential for reestablishing the connection, our implementation saves and restores both the timestamp state and selective acknowledgment state as described below.

The timestamp extension allows TCP to estimate round trip time more accurately, resulting in more efficient transmissions. The timestamp extension adds two fields to the TCP header. The first field contains the current timestamp value at the sender. The second field echoes the last timestamp seen by the sender from the receiver side. These fields allow TCP to determine the roundtrip time by taking the difference between the current time and the echoed time. Timestamps must increase monotonically, which also serves to protect again wrap around of sequence numbers (PAWS). If a packet has

a higher sequence number than the previous one but an earlier timestamp, it indicates that the sequence number has wrapped around and the packet is a duplicate and must be discarded.

We checkpoint the timestamp values in the packet headers to support the timestamp extension. Linux uses the number of timer interrupts since the kernel was started (jiffies) to set the current timestamp value. However, a kernel update resets the jiffies counter and so the timestamps do not increase monotonically after the update. As a result, the receiver discards all packets that are sent after the kernel update. We fixed this issue by adding an extra field to the TCP socket structure that holds the offset that must be added to the jiffies counter to obtain the correct timestamp number after the update. All timestamp calculations have been updated to use this offset as well. An alternative implementation is to set the jiffies counter to the same value as before the reboot. The advantage of our solution is that it keeps the change isolated to the TCP subsystem rather than affecting all uses of the jiffies counter, who might rely on the counter being initialized to a certain value.

Selective acknowledgment is another common extension that increases TCP throughput by making retransmissions more efficient. In addition to regular cumulative acknowledgments, the TCP receiver can add extra fields to the TCP header that allow the receiver to acknowledge discontiguous blocks of data. The sender then only has to retransmit just enough data to fill in the gaps. To restore selective acknowledgment on the receiver side, its is only necessary to checkpoint whether it was enabled in the first place. The sender stores an extra bit for each packet in the retransmit queue and sets this bit if it receives a selective acknowledgment for the packet. When retransmission is triggered, segments with the bit set are not retransmitted. Since our checkpoint preserves segment boundaries, we added the selective acknowledgment bit to the saved segments in the checkpoint and set this bit when restoring the retransmit queue.

Before the TCP subsystem is started by the kernel, all incoming TCP packets are

dropped by the kernel. These dropped packets will be eventually retransmitted by the sender. However, there is a time period when the TCP subsystem has started but the TCP connections have not been restored. During this time, if a packet arrives at a port with an unrestored socket, then TCP sends a reset packet to the sender, which closes the connection. To avoid this issue, we use the netfilter API to drop packets meant for applications which have not yet been restored from the checkpoint. The netfilter API allows inserting hooks at different layers of the network stack. We insert a hook that drops all TCP packets destined to any of the sockets stored in the checkpoint. The packet is dropped before the TCP subsystem receives the packet and so the reset is not sent. Any other network communication is unaffected, and after all the applications are restored, the hook is removed.

We do not save the state of the receive queue because we observed that it only contained unacknowledged packets. In particular, the Linux kernel sends acknowledgments only after packets have been copied to the user space. If these packets have been copied, then their contents are restored when the application is restored. Otherwise, the sender will retransmit the unacknowledged packets in the receive queue. The TCP protocol allows the kernel to acknowledge received packets before they are copied to the application. The receive queue is structured similar to the send queue, and so if future kernel versions send acknowledgments before copying data to the user space, then the receive queue must be restored in the same way as we restore the send queue.

4.1.5 Pipes

A Unix pipe is a unidirectional communication channel used for interprocess communication. There are two types of pipes, an unnamed pipe and a named pipe. An unnamed pipe is typically used to communicate between a parent and a forked child process. It is created using the pipe system call that returns two file descriptors, one for reading and one for writing. Data written to the write descriptor can be read from the read descrip-

tor. After a fork, the file descriptors (but not the pipe itself) get copied, and the parent and child communicate with one writing to one descriptor and the other reading from the other descriptor. A named pipe (also known as FIFO) is created using the mknod system call and has a name in the file system. Processes use named pipes by using the open system call and then reading or writing to them.

Internally, a pipe is represented by two file descriptors that point to a shared memory buffer consisting of a fixed number of pages. The memory buffer is used to store data that has been written and is ready to be read. If multiple processes uses a pipe (e.g., after a fork), they share references to the same file descriptors. There is no difference in the unnamed and named pipe implementation other than the way they are created.

To save a pipe in the checkpoint, we create two entries, one for each end of the pipe. When a process uses a pipe, we create a reference from the process to the entry for the pipe in the checkpoint. The memory buffers are not copied. Instead the checkpoint stores the pointers to the data pages.

Restoring pipes is tricky because each end of the pipe might be referenced by several processes and sometimes a process can close one end while the other end is open. For example, a writer may have exited (which closes the write side of the pipe) while the reader has not yet read all the data. For each pipe entry in the checkpoint, we initially create both ends of the pipe (for simplicity), even if there is only one end saved in the checkpoint. To create unnamed pipes, we use the pipe system call, and for named pipe, we call open on the pipe file name. Then we use the dup system call to assign the original file descriptor numbers.

We implement sharing of pipe file descriptors in the same way as we handle all shared resources (see Section **??**). We also keep track of which processes use which end of the pipe. After all processes have been restored, we consult the pipe usage count to see if there are any pipe ends that are not being used. Unused pipe ends represent pipe descriptors that were closed before the update. If any unused pipe ends are found, we

close them before allowing processes to resume execution. The pipe buffer pages are restored similar to user pages, as described in Section 4.1.2.

4.1.6 Unix Sockets

Unix sockets are another interprocess communication channel. Similar to pipes, Unix sockets can be unnamed or they can be bound to a file name (a listening socket) in the filesystem. The major differences between Unix sockets and pipes are that Unix sockets allow bidirectional communication and they can be created using the socket interface in addition to the regular file interface used for pipes.

Internally, Unix sockets are represented by a pair of socket descriptors, and both descriptors can be used to read and write data. Each socket keeps a reference to its peer, and when the data is written, it is placed on the peer's receive queue. Unlike TCP sockets, Unix sockets don't use a send queue.

For each socket descriptor, we create an entry in our checkpoint. The checkpoint entry contains a a reference to the peer descriptor, the type of the socket, connection state (connected or disconnected), file name for named sockets and the contents of the receive queue. For listening sockets, we also store whether the socket is accepting incoming connections. All processes using the socket keep a reference to the socket entry.

To restore Unix sockets, we first remove the existing file for named sockets. Then we create both the ends of the socket using standard Unix socket creation code. Then we set the socket connection state, file name and restore the contents of the receive queue. Shared sockets are handled as described previously in Section **??**.

4.1.7 Terminals and Keyboard

The Linux kernel uses a terminal emulator to provide a simple interface to the keyboard and text-mode display. An application accesses these devices by reading and writing to terminal device files located in the /dev directory. The kernel implements multiple

consoles by virtualizing the single keyboard and display. When switching from one console to another, it saves the hardware terminal state of the current console (i.e., screen contents) in memory, and restores the terminal state of the next console from memory.

When a process is using a virtual console (a file descriptor points to the terminal device), we first force the kernel to switch to a different console, which updates the terminal state stored in kernel memory, and then save this memory state in the checkpoint. Besides the screen contents, we also save the mode of the terminal (text or graphical), and the way input is processed, e.g., as lines or character-by-character and how the escape codes are processed.

To restore the terminal state, we open the terminal device that the application was using, and then again switch to a different virtual console. We restore the screen contents by copying them from the checkpoint and then we use the ioctl system call to set the correct terminal mode. At this point, we switch back to the original console, which synchronizes the hardware with the updated terminal state. Any status messages during boot are overwritten and should be logged separately for debugging purposes.

4.1.8 Framebuffer

The framebuffer device is used by graphical applications to access the video card. Applications access the framebuffer by opening the /dev/fb file and issuing ioctl system calls to set the desired resolution and bits per pixel. The framebuffer interface provides a simple bitmap display. Applications update the bitmap display by memory mapping the /dev/fb file and writing to the mmapped region. The kernel implements a framebuffer per virtual console (discussed in Section 4.1.7) by virtualizing the hardware framebuffer.

To add checkpointing support for the framebuffer, we have to save the framebuffer contents and the display mode settings. The contents of the framebuffer are saved by copying them into the checkpoint. A full copy is necessary because the framebuffer contents are modified when the new kernel is changing display setting and outputting

status messages during its initialization. To restore the framebuffer, we recreate the memory map that it was using originally and copy the contents from the checkpoint into the video memory.

The display settings are stored in the fb_var_screeninfo structure and control things like resolution, pixel size (16-bit vs 32-bit) and pixel format (RGB vs BGR). This data is modified by applications via the ioctl system call, so our checkpoint stores this information in the format used by this system call. This format is not dependent on the kernel version or the framebuffer driver and can be restored by calling the ioctl function.

We added framebuffer support for the Xfbdev X server. Xfbdev was chosen because it does not rely on more advanced hardware dependent features, like hardware acceleration or DRI. To add support for a more full featured X server like X11 would require a more comprehensive solution for saving and restoring state of hardware devices and drivers. Our implementation could successfully save and restore the state of the Xfbdev X server, the window manager (twm) and some simple applications like xclock, xcalc and xedit without requiring any changes to these applications.

4.1.9 Mouse

Graphical applications use the mouse in addition to the framebuffer (see Section 4.1.8) and the keyboard (see Section 4.1.7). We added support for saving and restoring the /dev/input/mice mouse device because this device is used by the Xfbdev X server. The mouse driver uses two structures mousedev and mousedev_client. The mousedev structure is shared among all the clients using the mouse. It is used to hold globally shared mouse state, including the client list and for interfacing with the lower level mouse subsystems. The mousedev_client structure maintains per-client information and queues packets received from the mouse. When a mouse moves, it sends a packet which contains the state of the buttons and displacement since the last packet. We saved and restored both of those structures, which ensures that the mouse operates correctly after

the update.

4.2 System Call Interface

We have added two system calls for executing the update process. These system calls 1) enable checkpointing specific processes, and 2) restoring all checkpointed processes. We have also added two debugging system calls that 1) determine whether a checkpoint is available, 2) help distinguish between a process that was started normally or was restored from a checkpoint. We do not require changes to existing programs. These system calls are intended to be used by user-level utilities and scripts to manage the update process after the kernel is initialized.

1. Enable_save_state: Takes a pid as an argument and sets a flag that indicates that a process with the given pid, all its children and all processes in the same thread group will be checkpointed. Without this flag, the process is not checkpointed.

2. Load_saved_state: Restores all the processes stored in the checkpoint.

3. Is_state_present: Checks if a checkpoint is available.

4. Was_state_restored: Used by a processes to check if it was started normally from scratch or if the process was created from a checkpoint. This system call can be used if some action needs to be taken by a process after it is restored.

4.3 Limitations

In this section, we discuss several limitations of our current implementation. Currently, we preallocate physically contiguous memory for the checkpoint when the kernel is first booted at a fixed location. After an update, we do not release this memory so that the kernel can be updated again from this fixed location. In the future, we plan to remove

this limitation by allocating discontiguous physical memory on demand when creating a checkpoint. One approach is to allocate this memory to the checkpointing process and save the state of the checkpointing process itself. After restoring the page tables of the checkpoint process, the rest of the checkpoint would be read from virtually contiguous memory. When this process exits, the checkpoint memory would be released.

Our implementation for handling temporary files currently only supports Linux ext3 file systems. The reason is that when we create a hardlink to a temporary file during checkpointing, we do not remove the file from the file-system specific orphan list. As a result, the file system still attempts to remove the file during the mount process. To avoid this problem after a kernel update, when an ext3 file system is mounted, we disable orphan list processing for the files associated with the checkpointed processes. A full implementation should remove the temporary file from the file-system specific orphan list when the checkpoint is taken.

We have added support for a wide variety of kernel features to enable supporting many types of commonly used applications. Some of the feature we do not implement include SYSV IPC, pseudo terminals, message queues, and the epoll system call. Our implementation supports Unix stream sockets, but we do not support Unix datagram sockets, which preserve message boundaries or the ability to pass file descriptors. We preserve boundaries for TCP segments and this code could be used for Unix datagram sockets also. Adding support for passing file descriptors would require supporting the sendmsg/recvmsg system calls. Applications using these kernel features cannot be restored in our system.

We do not see any fundamental issues that would prevent the missing functionality from being added in future versions because some of the features that we do implement already provide similar alternatives. For example, we provide Unix sockets and pipes for IPC, shared memory via threads and synchronization via the futex system call.

Since our implementation does not support all kernel features, we are not able to save

all processes running on the system. In particular, system or administrative applications like cron, udevd or getty are not saved and restored, and so these programs are shutdown and restarted normally on reboot. However, these programs do not have much state, and also do not require human intervention on restart. As more features are added to the implementation, these applications could be checkpointed as well.

For network applications, we assume that the ports used by an application before the update are not used by some other application before the application is restored. It is possible to modify the kernel and prevent port numbers that are in the checkpoint from being assigned to another process before the checkpoint is fully restored.

Our implementation does not restore the state of TCP sockets in which connection initialization (three-way handshake) is in progress. This limitation does not affect the server side because it is not notified about the connection until the connection is established. However, the client side will receive a reset packet after the kernel update. In our evaluation, clients do not make frequent connections and so this limitation did not affect them. In the future, we plan to support checkpointing the state of connections in progress.

The ability to save only some running processes creates challenges when processes share resources. For example, if a group of processes is sharing a resource, but we only restore some of them, then the restored processes may not work correctly if they rely on the other processes in the group. Our implementation handles some of these cases, such as the sharing of dynamic libraries among processes, by carefully tracking the reference counts of virtual memory regions. However, we require that all processes sharing other resources, e.g., pipes and Unix sockets, are saved and restored together.

Chapter 5

Evaluation

We evaluate our system by analyzing our checkpointing format and code in terms of its suitability for supporting kernel updates. Then, we describe our experience with updating major releases of the kernel. Finally, we present performance numbers in terms of kernel update times and overhead.

5.1 Code Analysis

Table 5.1 provides a summary of our checkpoint format. There are a total of 13 data structures that are saved in the checkpoint. The table shows the number of fields in each data structure and the number of fields that we save from each data structure in the checkpoint. The saved fields include both primitive types or pointers to buffers that need to be saved. The rest of the fields are mostly implementation dependent and do not need to be saved, although a few more would need to be saved if we add support for the features discussed in Section 4.

The code consists of roughly 6,000 lines of code, as shown in Table 5.2. Roughly 90% of the code resides in a separate kernel module, while the rest of the code is spread across various sub-systems. We can characterize kernel functions into four categories, based on how high-level it is and unlikely to be changed over time: system calls, exported functions,

Data structure	**Nr of fields**	**Nr of saved fields**
vm_area_struct	16	7
mm_struct	51	5
task_struct	135	32
fs_struct	5	3
files_struct	7	1
file	18	10
sock	53	10
tcp_sock	76	48
unix_sock	13	10
pipe_inode_info	13	4
vc_data	82	12
fb_info	23	6
mousedev	18	7

Table 5.1: Kernel structures and checkpoint format

global functions and private functions. System calls are available to user applications and cannot be changed without breaking backwards compatibility and are thus the most stable. Exported functions are available to loadable modules and usually only have minor changes between kernel versions. Global and private functions are expected to change more frequently. Our checkpoint saving code uses 20 functions and all of them are exported. The restore code uses 131 functions, of which 5 are system calls, 93 are exported, 2 are global, and 31 are private.

We needed to use private functions for two purposes: 1) managing resource identifiers, and 2) performing low-level process creation and initialization. The kernel provides user threads with identifiers for kernel managed resources such as PID, file descriptors, port

Subsystem	**Lines of code**
Checkpoint module	5257
Architecture specific	81
Memory management	70
File system	23
Process management	10
Networking	428
Total	5869

Table 5.2: New or modified lines of code

numbers, etc. When a process is restored, we must ensure that the same resources correspond to the same unique identifiers. However, since these identifiers are never modified, the kernel does not provide any exported or high-level functions to manipulate them. We believe that our solution is better suited for kernel updates because it doesn't impose any overhead for virtualizing identifiers during normal operation [16]. We also used private functions during process creation. The restore code is similar in functionality to the implementation of the execve system call that executes a file from disk. In our case, we create a process from the in-memory checkpoint data. The modifications needed were similar to the effort required to add support for another executable format.

We had to modify some architecture-specific code to reserve memory during kernel boot. We also made some changes to memory management code to assign reserved memory to the restored processes. We needed to make some changes to the Ext3 filesystem to prevent orphan clean up on boot so that temporary files are not deleted (See Section 3.2). All changes to the networking code relate to adding progress tracking (See Section 3.3) for reads and writes on TCP sockets, or changes needed to support TCP timestamps. Some of the changes do not alter the functionality of the kernel but were needed to provide access to previously private functions.

5.2 Experience with Updating Kernels

In this section, we describe the effort needed to use our system for performing kernel updates. We implemented our system starting with the Linux kernel version 2.6.28, released in December 2008, and have tested updating it, one major revision at a time, until version 2.6.34, released in May 2010 (roughly one and a half years of kernel updates). Table 5.3 shows that on average, each revision consists of 1.4 million lines of added or modified code. There were six million lines of changed code over the six revisions in 23,000 files, including many data structure modifications. We updated our code from one version to the next using a typical porting procedure: 1) extract all our code from the kernel and keep it in a separate git branch, 2) merge our code into the next major kernel release using git merge functionality, 3) compile the kernel and fix any errors, 4) once the kernel compiles, run an automated test suite that checks that all the features we have implemented are working correctly when updating between the kernel versions, and 5) commit the changes to our code so they can be used when moving to the next major release.

Table 5.3 shows the number of lines that had to be changed manually for each kernel release. These changes are small, both compared to the number of lines changed in the major release, as well as the number of lines in the checkpoint code. The majority of the changes were simple and were caught either during merge or during compilation. For example, several merge conflicts occurred when our code made a private function globally accessible, and some other nearby code was changed in the kernel update. Similarly, compilation errors occurred due to renaming. For example, we needed to use the TCP maximum segment size variable, and it was renamed from xmit_size_goal to xmit_size_goal_segs. These fixes are easy to make because they are caught by the compiler and do not the affect the behavior of the kernel or our code.

More complicated changes involved renaming functions and changing the function interface by adding arguments. In this case, we have to find out the new function name,

Kernel version	Lines of change in major release	Lines of change for checkpoint
2.6.29	1729913	42
2.6.30	1476895	16
2.6.31	1393049	7
2.6.32	1628415	5
2.6.33	1317386	50
2.6.34	882158	2

Table 5.3: Summary of updates needed for checkpoint code

and how to pass the new arguments to the function. For example, version 2.6.33 introduced a significant change to the interface used by the kernel to create and modify files and sockets. These changes were designed to allow calling file system and socket system calls cleanly from within the kernel. As a result, some internal functions used by our system were changed or removed, and our code needed to use the new file interface.

Our code needed to handle one significant data structure update conflict. Previously, the thread credentials, such as user_id and group_id, were stored in the thread's task_struct. In version 2.6.29, these credentials were moved into a separate structure, with the task_struct maintaining a pointer to this structure. We needed to change our system, similar to the rest of the kernel code, to correctly save and restore credentials. Two other data structure that we save, as shown in Table 5.1, were updated, but they required us to simply pass an additional parameter to a function.

Finally, the most difficult changes were functional bugs, that were not caused by changes to data structures or interfaces. We found these bugs when running applications. We encountered two such issues with TCP code. Previously, a function called tcp_current_mss was used to calculate and update the TCP maximum segment size. In 2.6.31, this function was changed so that it only did a part of this calculation, and

another function called tcp_send_mss was introduced that implemented the original tcp_current_mss behavior. Similarly, in 2.6.32, the TCP code added some conditions for setting the urgent flags in the TCP header, which indicates that out-of-band data is being sent. Our code was setting the urgent pointer to the value of 0, which in the new code set the urgent flag, thus corrupting TCP streams on restore. We needed to set the urgent flags based on the new conditions in the kernel.

All these ports were done by us within a day to a few days. We expect that kernel programmers would have found it much simpler to fix our code when updating their code. An interesting observation is that during porting, we never had to change the format of the checkpoint for any of the kernel versions. As a consequence, it is possible to freely switch between any of these kernel versions in any order. For example, it is possible to upgrade to version 2.6.33 from version 2.6.28, and then go back to version 2.6.30 seamlessly. This feature is not our goal, and we expect that the checkpoint format will change over time. We only intend to have a common checkpoint format between two consecutive major kernel releases.

5.3 Performance

We have tested our system for updating the kernel while running several desktop and server applications. For desktop applications, we have tested the system with the simple Xfbdev X server, the Twm window manager and several X programs. The mouse, keyboard, console and the graphics work correctly after the update, without requiring any application modifications or user intervention. Interestingly, the mouse and keyboard were initially freezing after an update, but only if we kept moving the mouse or typing on the keyboard while the update occurred. This problem was solved after we fixed a bug in the quiescence code. We were unable to test a modern GUI environment such as Gnome/GTK because these applications use SYSV shared memory, which we do not

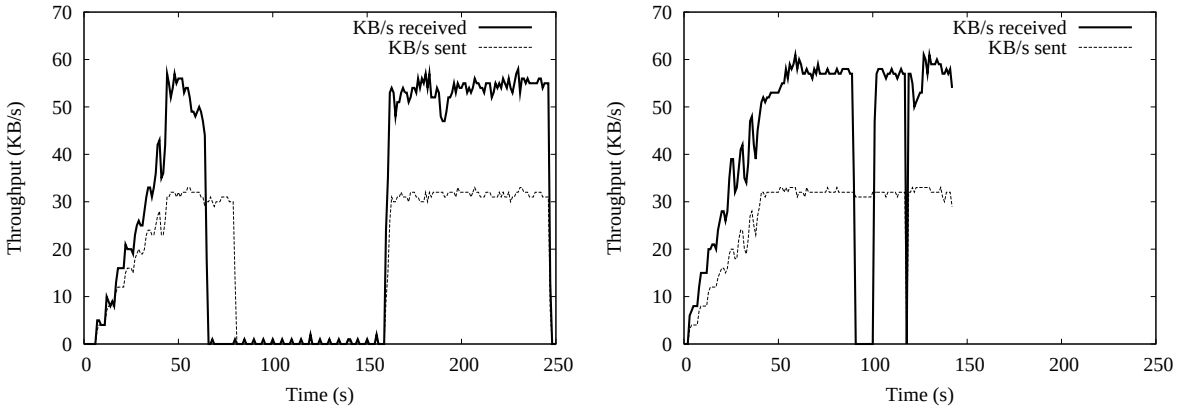

Figure 5.1: Quake reboot vs. update

support currently.

We conducted two types of experiments. First, we measured the throughput of server applications before and after the update. These experiments also show the downtime during an update. We used the Collectl system monitoring tool to measure throughput at the network interface level (sampled at one second interval). Second, we performed microbenchmarks to measure the per-process checkpoint size and time. All of our experiments run on the same machine with two Intel Xeon 3 GHz processors and 2GB of RAM, running Ubuntu 8.04 with our kernel that had support for updating the kernel.

5.3.1 Application Benchmarks

We tested several UDP (Quake game server, Murmer/Mumble voice-over-IP server) and TCP (Mysql, Memcached and Apache) server applications. Apache and Memcached used the epoll system call that our system does not support currently. Apache was compiled with epoll disabled and Memcached allows disabling epoll with environment variables. All these applications run after the update, without interrupting any requests in progress, and without requiring any other modifications. The Murmer and the Apache results were similar to the Quake results and are not shown.

5.3.1.1 Quake

We updated the kernel on the machine running the MVDSV 0.27 open source Quake server, while it was serving 8 ezQuake clients. The Quake server does not preserve its state across a reboot, and so the game needs to be restarted from the beginning. With our system, the clients resume after a short pause exactly where they were in the game before the update. For example, if a player was jumping through the air, the jump continues.

The ezQuake client was modified to make it easier to compare seamless updates and reboot. The unmodified client shuts down the current session and goes to the menu screen if the client stops receiving messages from the server for 60 seconds. To avoid the need to reconnect to the server by navigating the menu screen manually after a reboot, we modified the client so that it automatically attempts to reconnect to the server when no messages are received for 15 seconds. This change allows the client to reconnect to the rebooted server much faster and without any user interaction, making it easier to compare the systems. We chose the 15 second timeout because this time is much shorter than the time it takes for the server to reboot, and longer than the time it takes to update the kernel with our system. As a result, the client will reconnect as soon as the server is running again after the reboot, but it will not attempt to reconnect while the update is in progress. Note that the game state is still lost after the reboot, but not with seamless updates.

Figure 5.1 shows the network throughput with reboot versus update. With reboot, the clients timeout after 15 seconds and then begin attempting to reconnect every 5 seconds as shown by the little ticks at the bottom on the "sent" line. The server is down for roughly 90 seconds. In the update case, the server is operational in roughly 10 seconds, and all the clients resume normally.

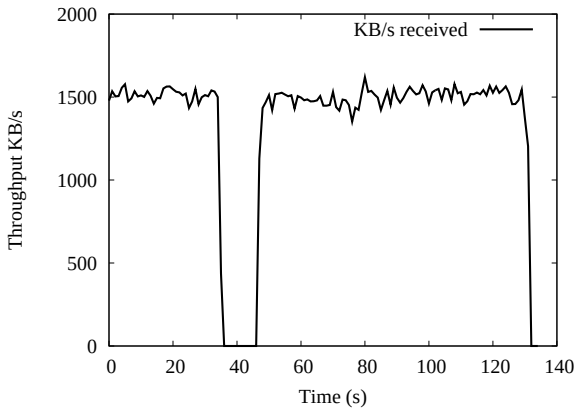

Figure 5.2: Mysql/sysbench update

5.3.1.2 MySQL

We used the sysbench OLTP benchmark to test the performance of MySQL server. This test involves starting transactions on the server and performing select and update queries in a transaction. The sysbench benchmark has no support for handling server failure. It returns an error when the MySQL server machine is rebooted and so we could not complete this test. Figure 5.2 shows the graph of TCP throughput as observed at the sysbench client while running the test. Similar, to the Quake results, the update time is again roughly 10 seconds in our system. The output of the server drops to zero while the update is being performed, but the connection to the client is not dropped. Once the update is finished and the new kernel is started, MySQL can continue to operate normally and the test runs to completion. There is no observable performance change before and after the update. No changes were required to MySQL or sysbench for this test.

5.3.1.3 Memcached

Memcached is a popular in-memory key-value caching system intended to speed up dynamic web applications. For example, it can be used to cache the results of database calls or page rendering. An application can typically survive a memcached server being

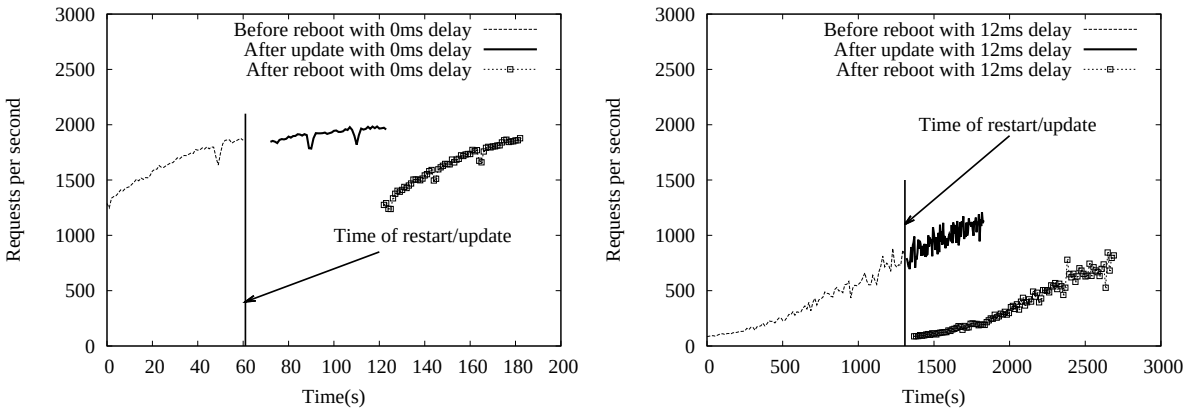

Figure 5.3: Memcached results after reboot vs. update

shutdown, because it can recalculate the results from scratch and start using the cache when it becomes available again. However, cache contents are lost on a reboot, while our system preserves the cache and so the application can use its contents right after the update.

In this test, we compare the performance impact of the Memcached cache being lost after a reboot versus being preserved in our system. We generated 200,000 key-value pairs, consisting of 100 byte keys and 400 byte values. Then, we used a Pareto distribution to send requests to prime the cache, so that 20% of the keys from the 200,000 key-pairs make up 80% of all the requests. Then we simulate a web application that uses Memcached for caching the results of database queries. This application makes key lookup requests to the Memcached server. For each key, the application first makes a get request from the cache. If the key is found, it makes the next get request. If it is not found, then it waits for a short time (delay) to simulate calculating a result, and then issues a set request to store the result in the server, before making the next get request.

We use 12 ms for the delay value for a database access, which is the average time per transaction in our previous sysbench OLTP test. We also use 0 ms for the delay value to represent the best case, in which there is no cost for calculating a result. However, when a cache miss occurs, this instantaneous result still needs to be sent to the Memcached

Application	Quiescence time	Save state time	Restore state time	Checkpoint size
Quake	334.1 ± 7.5 ms	98.59 ± 2.1 ms	22.85 ± 0.01 ms	135.2 ± 0.03 KB
MySQL	337.7 ± 2.5 ms	332.0 ± 45 ms	74.63 ± 3.1 ms	463.1 ± 24 KB
Memcached	329.5 ± 0.02 ms	6.4 ± 0.1 ms	38.1 ± 15 ms	112.3 ± 0.2 KB

Table 5.4: Per-application checkpoint time and size

Stage	Time
Initialize kernel	4.5 ± 0.3 s
Initialize services	6.9 ± 0.3 s

Table 5.5: Kernel restart time

server, thus requiring a get and a set request.

In the first part of the experiment we prime Memcached by sending requests to it, and then in the second part we send a second set of request after doing a clean reboot or an update with our system and compare the performance in terms of requests per second. The results of the experiment with 0 ms delay and 12 ms delay are shown in figure 5.4. For 0 ms case we primed Memcached with 100,000 requests in the first part and issued another set of 100,000 requests in the second part. In the 12 ms case we primed with 500,000 requests and made another 500,000 requests in the second part. The time where the Memcached server was rebooted or updated is shown with an arrow in both graphs.

The graphs show that after a reboot the number of requests per second declines because the contents of the cache are lost, and each miss adds extra overhead by restoring the lost cached value. In contrast when using our system the contents of the cache are preserved which results in a smaller number of misses and the request rate stays at the same level as before the update. Performing a regular reboot temporarily removes the benefit of using the in-memory cache until the contents of the cache are restored, while our approach preserves the performance benefit.

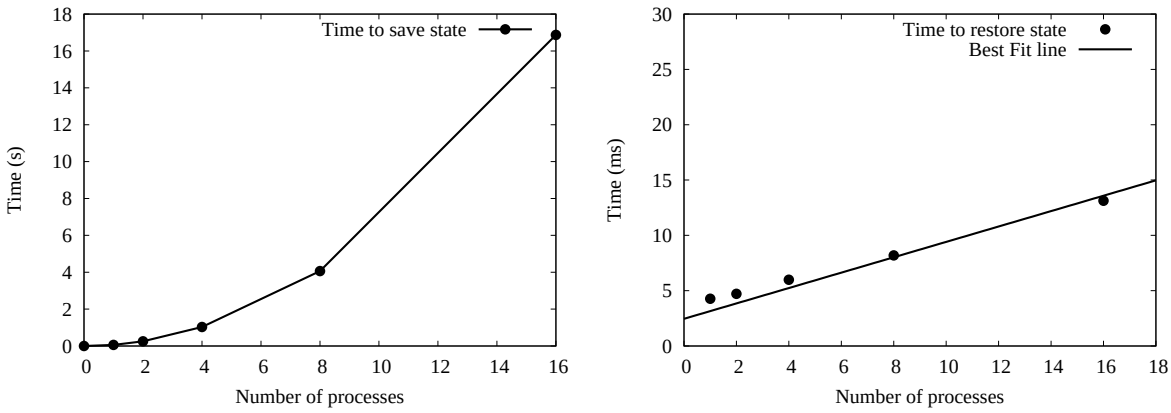

Figure 5.4: Mmap checkpoint-restore time

5.3.2 Microbenchmarks

Table 5.4 breaks down the time to reach quiescence, save the process state and restore the process state for the three applications described above. Quiescence time is measured from the time when we start the checkpoint process until we are ready to take the checkpoint (i.e., the last step in Section 3.2) and includes the time to shutdown all the devices. The save state time is the time it takes to copy the kernel state into the checkpoint. The time to initialize the kernel is measured from when the new kernel's code starts executing to when the kernel starts the init process. The time to initialize services is measured from when the init process starts to when the saved processes begin to be restored, and the restore time is the time it takes for the saved applications to be restored and start running. The kernel quiescence time is roughly 330 milliseconds for each of these experiments. The checkpoint save time ranges from 6-350 ms, but as discussed below, we expect that this time can be reduced significantly with some simple optimizations. The checkpoint restore time has a smaller range from 25-55 ms. All these times are much lower than the time it takes to initialize the new kernel and the system services, as shown in Table 5.5. The last column of Table 5.4 shows the checkpoint size for each application excluding the memory pages of the application.

We also conducted a microbenchmark to measure the checkpoint and restore time

with increasing number of allocated frames in the system. The benchmark is run with 1, 2, 4, 8 and 16 processes. Each process in this benchmark allocates 16 MB of private memory using the mmap system call and writes to this memory to ensure that the kernel assigns page frames to the process. Figure 5.4 shows the checkpoint save and restore time with increasing number of processes. The save time is roughly one second per process in this benchmark with 16 processes and it grows with increasing number of processes because our implementation for saving state is not optimized. In particular, the code uses a linked list to detect shared pages. With 16 processes, and 16 MB of memory per process, there are 64K (2^{16}) pages in the linked list, making the search for those many pages in the list very slow ($\sim 2^{32}$ operations). These lookups can be sped up with a hash table or we could use the reverse mapping information available in the memory manager to detect shared pages. The restore time per process is roughly one ms per 16MB process because we take advantage of the kernel memory manager to ensure that shared pages are assigned to each process correctly. Note that this time mainly accounts for restoring the address space and is much smaller than the restore time for the benchmark applications (25-65 ms) which also need to restore other resources such as the network buffers.

Chapter 6

Conclusions

We have design a reliable and practical kernel update system that checkpoints application-visible state, updates the kernel, and restores the application state. We have argued that this approach requires minimal programmer effort, no changes to applications, and can handle all backward compatible patches. Our system can transparently checkpoint the state of network connections, some common hardware devices and user applications. It can achieve quiescence for any kernel update, and it restarts all system calls transparently to applications. We also performed a detailed analysis of the effort needed to support updates across major kernel releases, representing more than a year and a half of changes to the kernel. Our system required a small number changes to existing kernel code, and minimal effort to handle major kernel updates, consisting of a million lines of code. Finally, we evaluated our implementation and showed that it works seamlessly for several, large applications, with no perceivable performance overhead, and reduces reboot times significantly. As future work, we plan to add features currently missing in our implementation, such as support for SYSV IPC, epoll, and pseudo terminals, and checkpointing state for all hardware devices.

Bibliography

[1] Oracle database high availability features and products. `http://docs.oracle.com/cd/B28359_01/server.111/b28281/hafeatures.htm`.

[2] Performing rolling updates and upgrades in a db2 high availability disaster recovery (hadr) environment. `http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.ha.doc/doc/t0011766.html`.

[3] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 187–198, 2009.

[4] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: challenges and solutions to updating an operating system on the fly. In *Proceedings of the USENIX Technical Conference*, pages 1–14, 2007.

[5] Fernando Luis Vazquez Cao. Reinitialization of devices after a soft-reboot. Usenix Linux Storage & Filesystem Workshop, February 2007.

[6] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 35–44, 2006.

[7] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *Proceedings of the International Conference on Software Engineering*, pages 271–281, 2007.

[8] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Curios: improving reliability through operating system structure. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 59–72, 2008.

[9] Alex Depoutovitch and Michael Stumm. Otherworld: giving applications a chance to survive os kernel crashes. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 181–194, 2010.

[10] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson. Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. Linux Foundation, December 2010. `www.linuxfoundation.org/publications/whowriteslinux.pdf`.

[11] Oren Laadan and Serge E. Hallyn. Linux-CR: Transparent application checkpoint-restart in linux. In *Proceedings of the Linux Symposium*, 2010.

[12] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211–223, 2004.

[13] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 327–340, 2007.

[14] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 72–83, 2006.

[15] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 361–376, December 2002.

[16] Shaya Potter and Jason Nieh. Reducing downtime due to system maintenance and upgrades. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, pages 47–62, 2005.

[17] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the USENIX Technical Conference*, pages 141–154, 2003.

[18] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, and Michael M. Swift. Membrane: Operating system support for restartable file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[19] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2004.

[20] Michael M. Swift, Damien Martin-Guillerez, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Live update for device drivers. Computer Sciences Technical Report CS-TR-2008-1634, University of Wisconsin, March 2008.

[21] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. Engineering fault-tolerant tcp/ip servers using ft-tcp. In *Proceedings of the IEEE Dependable Systems and Networks (DSN)*, 2003.