Appeared in *The Linux Symposium 2010*, Ottawa, July 2010

Linux-CR: Transparent Application Checkpoint-Restart in Linux

Oren Laadan
Columbia University
orenl@cs.columbia.edu

Serge E. Hallyn
IBM
serge@hallyn.com

Abstract

Application checkpoint-restart is the ability to save the state of a running application so that it can later resume its execution from the time of the checkpoint. Application checkpoint-restart provides many useful benefits including fault recovery, advanced resources sharing, dynamic load balancing and improved service availability. For several years the Linux kernel has been gaining the necessary groundwork for such functionality, and now support for kernel based transparent checkpoint-restart is also maturing. In this paper we present the implementation of Linux checkpoint-restart, which aims for inclusion in Linux mainline. We explain the usage model and describe the user interfaces and some key kernel interfaces. Finally, we present preliminary performance results of the implementation.

1 Introduction

Application checkpoint-restart can provide many benefits, including fault recovery by rolling back applications to a previous checkpoint, better response time by restarting applications from checkpoints instead of from scratch, and better system utilization by suspending jobs on demand. Application migration is useful for dynamic load balancing by moving applications to less loaded hosts, fault resilience by migrating applications off of faulty hosts, and improved availability by evacuating applications before host maintenance so that they continue to run with minimal downtime.

While application checkpoint-restart can be performed at different levels, choosing the correct level to transparently support unmodified applications is crucial in practice to enable deployment and widespread use. The two main approaches for providing application checkpoint-restart are in userspace or in the operating system kernel. Userspace approaches are simple to implement and use, but lack transparency and severely limit the types of applications that can be supported. In contrast, kernel approaches utilize operating system support to provide full application transparency, without requiring any changes to applications.

Given the benefits of the kernel level approach, it has been the focus of several projects that aim to provide application checkpoint-restart for Linux [5, 9, 12, 14, 16, 20]. However, none of them is integrated into the mainstream Linux—they are implemented as kernel modules or sets of kernel patches instead. As such, they incur a burden both on users because they are cumbersome to install, and on developers because maintaining them on top of quickly changing upstream kernels is a sisyphean task and they quickly fall behind.

We present Linux-CR, an implementation of transparent application checkpoint-restart in Linux, which aims for inclusion in the Linux mainline kernel. Linux-CR builds on the experience garnered through the previous out-of-mainstream projects. It benefits from many of the supporting features needed for checkpoint and restart that are available upstream today, including the ability to isolate applications inside containers using namespaces, and to selectively freeze applications using the freezer control group. Linux-CR's checkpoint-restart is transparent, secure, reliable, and efficient, and does not adversely impact the performance or code quality of the rest of the Linux kernel.

The remainder of the paper is organized as follows. Section 2 describes Linux checkpoint-restart usage model from a user's point of view. Section 3 presents in detail the checkpoint-restart architecture. Section 4 provides an overview of the in-kernel API for checkpoint-restart. Section 5 presents experimental results. Section 6 discusses related work. Finally, we present some concluding remarks.

2 Usage

The granularity of checkpoint-restart is a process hierarchy. A checkpoint begins at a task which is the root of the hierarchy, and proceeds recursively to include all the descendant processes. It generates a checkpoint image which represents the state of the process hierarchy. A restart takes a checkpoint image as input to create an equivalent process hierarchy and restore the state of the processes accordingly.

Before a checkpoint begins, and for the duration of the entire checkpoint, all processes in the hierarchy must be frozen. This is necessary to prevent them from modifying system state while a checkpoint is underway, and thus avoid inconsistencies from occurring in the checkpoint image. Freezing the processes also puts them in a known state–just before returning to userspace–which is useful because it is a state with only a trivial kernel stack to save and restore.

Linux-CR supports two main forms of checkpoint: *container checkpoint* and *subtree checkpoint*. The distinction between them depends on whether the checkpointed hierarchy is "self contained" and "isolated". The term "self contained" refers to a hierarchy that includes all the processes that are referenced in it. In particular, it must include all parent, child, and sibling processes, and also orphan processes that were re-parented. The term "isolated" refers to a hierarchy whose resources are only referenced by processes that belong to the hierarchy. For example, open file handles held by processes in the hierarchy may not be shared by processes not in the hierarchy. A key property of hierarchies that satisfy both conditions is that their checkpoints are consistent and reliable, and therefore guarantee a successful restart.

Container checkpoint operates on process hierarchies that are both isolated and self contained. In Linux, isolated and self-contained hierarchies are created using *namespaces* [4], which facilitate the provision of private sets of resources for groups of processes. In particular, the PID-namespace can be used to generate a sub-hierarchy that is self contained. A useful management tool for this is Linux Containers [13], which leverages namespaces to encapsulate applications inside virtual execution environments to give them the illusion of running privately.

Checkpointing an entire container ensures that processes inside the container do not depend on processes from the outside. To ensure also that the checkpoint is consistent, the state of shared resources in use by processes in the container must remain unmodified for the duration the checkpoint. Because the processes in the container are frozen they may not alter the state. Thus it suffices to require that, at checkpoint time, none of the resources are referenced by processes from the outside. Combined, these properties guarantee that a future restart from a container checkpoint will always succeed. Note that to leverage container checkpoint, users must launch those application that they wish to checkpoint inside containers.

Subtree checkpoint operates on arbitrary hierarchies. Subtree checkpoints are especially handy since they do not require users to launch their applications in a specific way. For example, casual users that execute long-running computations can simply checkpoint their jobs periodically. However, subtree checkpoints cannot provide the same guarantees as container checkpoints. For instance, because checkpoint iterates the hierarchy from the top down, it will not reach orphan processes unless it begins with the `init(1)` process, so orphan processes will not be recreated and restored at restart. Instead, it is the user's responsibility to ensure that dependencies on processes outside the hierarchy and resource sharing from outside the hierarchy either do not exist or can be safely ignored. For example, a parent process may remain outside the hierarchy if we know that the child process will never attempt to access it. In addition, if outside processes share state with the container, they must not modify that state while checkpoint takes place.

We support a third form of checkpoint: *self-checkpoint*. Self-checkpoint allows a running process to checkpoint itself and save its own state, so that it can restart from that state at a later time. It does not capture the relationships of the process with other processes, or any sharing of resources. It is most useful for standalone processes wishing to be able to save and restore their state. Self-checkpoint occurs by an explicit system call and always takes place in the context of the calling process. The process need not be frozen for the duration of the checkpoint. This form of checkpoint is analogous to the `fork` system call in that the system call may return in two different contexts: one when the checkpoint completes and another following a successful restart. The process can use the return value from the system call to distinguish between the two cases: a successful checkpoint operation will return a "checkpoint ID" which is a non-zero

positive integer, while a successful restart operation always returns zero.

2.1 Userspace Tools

The userspace tools consist of three programs: *checkpoint* to take a checkpoint of a process hierarchy, *restart* to restart a process hierarchy from a checkpoint image, and *ckptinfo* to provide information on the contents of a checkpoint image. A fourth utility, *nsexec*, allows users to execute applications inside a container by creating an isolated namespace environment for them. These tools provide the most basic userspace layer on top of the bare kernel functionality. Higher level abstractions for container management and related functionality are provided by packages like lxc [3] (of Linux Containers) and libvirt [2].

Figure 1 provides a practical example that illustrates the steps involved to launch an application inside a container, then checkpoint it, and finally restart it in another container. In the example, the user launches a session with an `sshd` daemon and a `screen` server inside a new container.

Lines 1–3 create a freezer control group to encapsulate the process hierarchy to be checkpointed. Lines 5–13 create a script to start the processes, and the script is executed in line 15. In line 7 the script joins the freezer control group. Descendant processes will also belong there by inheritance, so that later it will be possible to freeze the entire process hierarchy. Lines 8–10 close the standard input, output and error to decouple the new container from the current environment and ensure that it does not depend on `tty` devices.

In line 15, *nsexec* launches the script inside a new private set of namespaces. Lines 18–21 show how the application is checkpointed. First, in line 18 we freeze the processes in the container using the freezer control group. We use the *checkpoint* utility in line 19 to checkpoint the container, using the script's `PID` to indicate the root of the target process hierarchy. In the example, we kill the application once the checkpoint is completed, and thaw the (now empty) control group. We restart the application in line 23. Finally, in line 25 we show how to examine the contents of the checkpoint image using the *ckptinfo* utility.

The log files provided to both the *checkpoint* and *restart* commands are used for status and error reports. Should

```
 1  $ mkdir -p /cgroup
    $ mount -t cgroup -o freezer cgroup /cgroup
    $ mkdir /cgroup/1

 5  $ cat > myscript.sh << EOF
    #!/bin/sh
    echo $$ > /cgroup/1/tasks
    exec 0>&-
    exec 1>&-
10  exec 2>&-
    /usr/sbin/sshd -p 999
    screen -A -d -m -S mysession somejob.sh
    EOF

15  $ nohup nsexec -tgcmpiUP pid myscript.sh &

    $ PID=`cat pid`
    $ echo FROZEN > /cgroup/1/freezer.state
    $ checkpoint $PID -l clog -o image.out
20  $ kill -9 $PID
    $ echo THAWED > /cgroup/1/freezer.state

    $ restart -l rlog -i image.out

25  $ ckptinfo -ve < image.out
```

Figure 1: A simple checkpoint-restart example

a failure occur during either operation, the log file will contain error messages from the kernel that carry more detailed information about the nature of the error. Further debugging information can be gained from the checkpoint image itself with the *ckptinfo* command, and from the restart program by using the "-vd" switches.

2.2 System Calls

The userspace API consists of two new system calls for checkpoint and restart, as follows:

- **`long checkpoint(pid, fd, flags, logfd)`**

This system call serves to request a checkpoint of a process hierarchy whose root task is identified by `@pid`, and pass the output to the open file indicated by the file descriptor `@fd`. If `@logfd` is not `-1`, it indicates an open file to which error and debug messages are written. Finally, `@flags` determines how the checkpoint is taken, and may hold one or more of the values listed in Table 1.

On success the system call returns a positive checkpoint identifier. In the future, a checkpoint image may optionally be briefly preserved in kernel memory. In such cases, the identifier would serve to reference that image. On failure the return value is `-1`, and `errno` indicates a suitable error value.

In self-checkpoint, where a process checkpoints itself, it is necessary to distinguish between the first return from

Operation	Flags	Flag description
Checkpoint	CHECKPOINT_SUBTREE	perform a subtree checkpoint
	CHECKPOINT_NETNS	include network namespace state
Restart	RESTART_TASKSELF	perform a self-restart
	RESTART_FROZEN	freeze all the restored tasks after restart
	RESTART_GHOST	indicate a process that is a place-holder
	RESTART_KEEP_LSM	restore checkpointed MAC labels (if permitted)
	RESTART_CONN_RESET	force open sockets to a closed state

Table 1: System call flags (checkpoint and restart)

a successful checkpoint, and a subsequent return from the same system call but following a successful restart. This distinction is achieved using the return value, similarly to `sys_fork`. Specifically, the system call returns plain `0` if it came from a successful restart. In either case, when the operation fails the return value is `-1`, and `errno` indicates a suitable error value.

- **`long sys_restart(pid, fd, flags, logfd)`**

This system call serves to restore a process hierarchy from a checkpoint image stored in the open file indicated by the file descriptor `@fd`. It is intended to be called by all the restarting tasks in the hierarchy, and by a special process that coordinates the restart operation. When called by the coordinator, `@pid` indicates the root task of the hierarchy (as seen in the coordinator's PID-namespace). The root task must be a child of the coordinator. When not called by the coordinator, `@pid` must remain `0`. If `@logfd` is not `-1`, it indicates an open file to which error and debug messages are written. Finally, `@flags` determines how the checkpoint is taken, and may hold one or more of the values listed in Table 1.

On success the system call returns in the context of the process as it was saved at the time of the checkpoint. The exact behavior depends on how and when the checkpoint was taken. If the process was executing in userspace prior to the checkpoint, then the restart will arrange for it to resume execution in userspace exactly where it was interrupted. If the process was executing a system call, then the return value will be set to the return value of that system call whether it completed or was interrupted. For a self-checkpoint, the restart will arrange for the process to resume execution at the first instruction after the original call to `sys_checkpoint`, with the system call's return value set to `0` to indicate that this is the result of a restart. On failure the return value is `-1` and `errno` indicates a suitable error value.

3 Architecture

The crux of checkpoint-restart is a mechanism to serialize the execution state of a process hierarchy, and to restore the process hierarchy and its state from the saved state. For checkpoint-restart of multi-process applications, not only must the state associated with each process be saved and restored, but the state saved and restored must be globally consistent and preserve process dependencies. Furthermore, for checkpoint-restart to be useful in practice, it is crucial that it transparently support existing applications.

To guarantee that the state saved is globally consistent among all processes in a hierarchy, we must satisfy two requirements. First, the processes must be frozen for the duration of the checkpoint. Second, the resources that they use must not be modified by processes not in the hierarchy. These requirements ensure that the state will not be modified by processes in or outside the hierarchy while the execution state is being saved.

To guarantee that the operation is transparent to applications, we must satisfy two more requirements. First, the state must include all the resources in use by processes. Second, resource identifiers in use at the time of the checkpoint must be available when the state is restored. These requirements ensure not only that all the necessary state exists when restart completes, but also that it is visible to the application as it was before the checkpoint, so that the application remains unaware.

Linux-CR builds on the *freezer* subsystem to achieve quiescence of processes. This subsystem was created to allow the kernel to freeze all userspace processes in preparation for a full system suspend to disk. To accommodate checkpoint-restart, the *freezer* subsystem was recently re-purposed to enable freezing of groups of processes, with the introduction of the *freezer* control group.

Linux-CR leverages *namespaces* [4] to encapsulate processes in a self-contained unit that isolates them from other processes in the system and decouples them from the underlying host. Namespaces provide virtual private resource names: resource identifiers within a namespace are localized to the namespace. Not only are they invisible to processes outside that namespace, but they do not collide with resource identifiers in other namespaces. Thus, resource identifiers, such as `PIDs`, can remain constant throughout the life of a process even if the process is checkpointed and later restarted, possibly on a different machine. Without it, identifiers may in fact be in use by other processes in the system. To accommodate checkpoint-restart, namespaces were extended to allow restarting processes to select predetermined identifiers upon the allocation of their resources, so that those processes can reclaim the same set of identifiers they had used prior to the checkpoint.

For simplicity, we describe the checkpoint-restart mechanism assuming *container checkpoint*. We also assume a shared storage (across participating machines), and that the filesystem remains unmodified between checkpoint and restart. In this case, the filesystem state is not generally saved and restored as part of the checkpoint image, to reduce checkpoint image size. Available filesystem snapshot functionality [6, 10, 15] can be used to also provide a checkpointed filesystem image. We focus only on checkpointing process state; details on how to checkpoint filesystem, network, and device state are beyond the scope of this paper.

3.1 Kernel vs. Userspace

Previous approaches to checkpoint-restart have run the gamut from fully in-kernel to hybrid to fully userspace implementations. While many properties of processes and resources can be recorded and restored in userspace, some state exists that cannot be recorded or restarted from userspace. To provide application transparency and allow applications to use the full range of operating system services, we chose to implement checkpoint-restart in the kernel. In addition, an in-kernel implementation is not limited to user visible APIs such as system calls–it can use the full range of kernel APIs. This not only simplifies the implementation, but also allows use of native locking mechanisms to ensure atomicity at the desired granularity.

Checkpoint is performed entirely in the kernel. Restart is also done in the kernel, however, for simplicity and flexibility, the creation of the process hierarchy is done in userspace. Moving some portion of the restart to userspace is an exception, which is permitted under two conditions: first, it must be straightforward and leverage existing userspace APIs (i.e. not introduce specialized APIs). Second, doing so in userspace should bring significant added value, such as improved flexibility. Also, all userspace work must occur before entering the kernel, to avoid transitions in and out of the kernel.

For instance, the incentive to do process creation in userspace is because it is simple to use the `clone` system call to do so, and because it allows for great flexibility for restarting processes to do useful work after the process hierarchy is created and before the rest of the restart takes place. Indeed, the entire hierarchy is created before in-kernel restart is performed. Likewise, it is desirable to restore network namespaces in userspace[1]. Doing so will allow reuse of existing userspace network setup tools that are well understood instead of replicating their high-level functionality inside the kernel. Moreover, it will allow users to easily adjust network settings at restart time, e.g. change the network device or its setup, or add a firewall to the configuration.

3.2 Checkpoint

A checkpoint is performed in the following steps (steps 2–4 are done by the `checkpoint` system call):

1. Freeze the process hierarchy to ensure that the checkpoint is globally consistent.
2. Record global data, including configuration and state that are global to the container.
3. Record the process hierarchy as a list of all checkpointed processes, their `PIDs`, and relationships.
4. Record the state of individual processes, including credentials, blocked and pending signals, CPU registers, open files, virtual memory, etc.
5. Thaw the processes to allow them to continue executing, or terminate the processes in case of migration. (If a filesystem snapshot is desired, it is taken prior to this step.)

Checkpoint is done by an auxiliary process, and does not require the collaboration of processes being checkpointed. This is important since processes that are not runnable, e.g. stopped or traced, would not be able to

[1] However, as of the writing of this paper, this is yet undecided.

perform their own checkpoint. Moreover, this can be extended in the future to multiple auxiliary processes for faster checkpoint times of large process hierarchies.

Much effort was put to make checkpoint robust in the sense that if a checkpoint succeeds then, given a suitable environment, restart will succeed too. The implementation goes to great extents to be able to detect whether checkpointed processes are "non-restartable". This can happen, for example, when a process uses a resource that is unsupported for checkpoint-restart, therefore it will not be saved at all. Even if a resource is supported, it may be temporarily in an unsupported state. For example, a socket that is in the process of establishing a connection is currently unsupported.

3.3 Restart

A restart is performed in the following steps (step 3 is done by the `restart` system call):

1. Create a new container for the process hierarchy, and restore its configuration and state.
2. Create the process hierarchy as prescribed in the checkpoint image.
3. Restore the state of individual processes in the same order as they were checkpoint.
4. Allow the processes to continue execution[2].

Restart is managed by a special *coordinator* process, which supervises the operation but is not a part of the restarted process hierarchy. The coordinator process creates and configures a new container, and then generates the new process hierarchy in it. Once the hierarchy is ready, all the processes execute a system call to complete the restart of each process in-kernel.

To produce the process hierarchy, it is necessary to preserve process dependencies, such as parent-child relationships, threads, process groups, and sessions. The restored hierarchy must satisfy the same constraints imposed by process dependencies at checkpoint. Because the process hierarchy is constructed in userspace, these dependencies must be established at process creation time (to leverage the existing system calls semantics). The order in which processes are created is important, because some dependencies are not reflected directly from the hierarchical structure. For instance, an orphan process must be recreated by a process that belongs to the correct session group to correctly inherit that group.

Linux-CR builds on two algorithms introduced by Zap [12] to reconstruct the process hierarchy. The algorithms are designed to create a hierarchy that is equivalent to the original one at checkpoint. The first algorithm, *DumpForest*, analyzes the state of a process hierarchy. It runs in linear time with the number of `PID`s in use in the hierarchy. The output is a table with an entry for each `PID`; The table encodes the order and the manner in which processes should be restarted. The second algorithm, *MakeForest*, reconstructs the hierarchy. It works in a recursive manner by following the instructions set forth by the table. It begins with a single process that will be used as the root of the hierarchy to fork its children, then each process creates its own children, and so on. For a detailed discussion of these algorithms refer to [12].

In rebuilding the process hierarchy, there are two special cases of `PID`s referring to terminated processes that require additional attention. One case is when a `PID` of a dead process is used as a `PGID` of another process. In this case, the restart algorithm creates a "ghost" process–a placeholder that lives long enough so that its `PID` can be used as the `PGID` of another process, but terminates once the restart completes (and before the hierarchy may resume its execution, to avoid races). Another case is when a `PID` represents a *zombie* process that has exited but whose state has not been cleaned up yet. In this case, the restart algorithm creates a process, restores only minimal state such as its exit code, and finally the process exits to become a zombie.

Because the process hierarchy is created in userspace, the restarting processes have the flexibility to do useful work before eventually proceeding with in-kernel restart. For instance, they might wish to create a new custom networking route or filtering rule, create a virtual device which existed at the host at the time of checkpoint, or massage the mounts tree to mask changes since checkpoint.

Once the process hierarchy is created, all the processes invoke the `restart` system call and the remainder of the restart takes place in the kernel. Restart is done in the same order that processes were checkpointed. The restarting processes now wait for their turn to restore their own state, while the *coordinator* orchestrates the restart.

[2] It is also possible to freeze the restarted processes, which is useful for, e.g., debugging.

Restart is done within the context of the process that is restarted. Doing so allows us to leverage the available kernel functionality that can only be invoked from within that context. Unlike checkpoint, which requires observing process state, restart is more complicated as it must create the necessary resources and reinstate their desired state. Being able to run in process context and leverage available kernel functionality to perform these operations during restart significantly simplifies the restart mechanism.

In the kernel, the `restart` system call depends on the caller. The *coordinator* first creates a common restart context data structure to share with all the restarting process, and waits for them to become properly initialized. It then notifies the first process to start the restart, and waits for all the restarting tasks to finish. Finally, the *coordinator* notifies the restarting tasks to resume normal execution, and then returns from the system call.

Correspondingly, restarting processes first wait for a notification from the *coordinator* that indicates that the restart context is ready, and then initialize their state. Then, each process waits for its turn to run, restores the state from the checkpoint image, notifies the next restarting process to run, and waits for another signal from the *coordinator* indicating that it may resume normal execution. Thus, processes may only resume execution after all the processes have successfully restored their state (or fail if an error has occurred), to prevent processes from returning to userspace prematurely before the entire restart completes.

3.4 The Checkpoint Image

The checkpoint image is an opaque *blob* of data, which is generated by the `checkpoint` system call and consumed by the `restart` system call. The blob contains data that describes the state of select portions of kernel structures, as well as process execution state such as CPU registers and memory contents. The image format is expected to evolve over time as more features are supported, or as existing features change in the kernel and require to adjust their representation. Any changes in the blob's format between kernel revisions will be addressed by userspace conversion tools, rather than attempting to maintain backward compatibility inside the `restart` system call.

Internally, the blob consists of a sequence of records that correspond to relevant kernel data structures and represent their state. For example, there are records for process data, memory layout, open files, pending signals, to name a few. Each record in the image consists of a header that describes the type and the length of the record, followed by a payload that depends on the record type. This format allow userspace tools to easily parse and skim through the image without requiring intimate knowledge of the data. Keeping the data in self contained records will also be suitable for parallel checkpointing in the future, where multiple threads may interleave data from multiple processes into a single stream.

Records do not simply duplicate the native format of the respective kernel data structures. Instead, they provide a representation of the state by copying those individual elements that are important. One justification is that during restart, one already needs to inspect, validate and restore individual input elements before copying them into kernel data structures. However, the approach offers three additional benefits. First, it improves image format compatibility across kernel revisions, being agnostic to data structure changes such as reordering of elements, addition or deletion of elements that are unimportant for checkpoint-restart, or even moving elements to other data structures. Second, it reduces the total amount of state saved since many elements may be safely ignored. Per process variables that keep scheduler state are one such example. Third, it allows a unified format for architectures that support both 32-bit and 64-bit execution, which simplifies process migration between them.

The checkpoint image is organized in five sections: a header, followed by global data, process hierarchy, the state of individual processes, and a finally a trailer. The header includes a magic number (to identify the blob as a checkpoint image), an architecture identifier in little-endian format, a version number, and some information about the kernel configuration. It also saves the time of the checkpoint and the flags given to the system call. It is followed by an architecture dependent header that describes hardware specific capabilities and configuration.

The global data section describes configuration and state that are global to the container being checkpointed. Examples include Linux Security Modules (LSM) and network devices and filters. In the future container-wide mounts may also go here. The process hierarchy section that follows provides the list of all checkpointed processes, their `PIDs` and their relationships, e.g., parent-child, siblings, threads, and zombies. These two section

are strategically placed early in the image for two reasons: first, it allows restart to create a suitable environment for the rest of the restart early on, and second, it allows to do so in userspace.

The remainder of the checkpoint image contains the state of all of the tasks and the shared resources, in the order that they were reached by the process hierarchy traversal. For each task, this includes state like the task structure, namespaces, open files, memory layout, memory contents, CPU state, signals and signal handlers, etc. Finally, the trailer that concludes the entire image serves as a sanity check.

The checkpoint-restart logic is designed for streaming to support operation using a sequential access device. Process state is saved during checkpoint in the order in which it needs to be used during restart. An important benefit of this design is that the checkpoint image can be directly streamed from one machine to another across the network and then restarted, to accomplish process migration. Using a streaming model provides the ability to pass checkpoint data through filters, resulting in a flexible and extensible architecture. Example filters include encryption, signature/validation, compression, and conversion between formats of different kernel versions.

3.5 Shared Resources

Shared resources may be referenced multiple times, e.g. by multiple processes or even by other resources. Examples of resources that may be shared include files descriptors, memory address spaces, signal handlers and namespaces. During checkpoint, shared resources will be considered several times as the process hierarchy is traversed, but their state need only be saved once.

To ensure that shared resources are saved exactly once, we need to be able to uniquely identify each resource, and keep track of resources that have been saved already. More specifically, when a resource is first discovered, it is assigned a unique identifier (*tag*) and registered in a hash-table using its kernel address (at checkpoint) or its tag (at restart) as a key. The hash-table is consulted to decide whether a given resource is a new instance or merely a reference to one already registered. Note that the hash-table itself is not saved as part of the checkpoint image; instead, it is rebuilt dynamically during both checkpoint and restart, and discarded when they complete.

During checkpoint, shared resources are examined by looking up their kernel addresses in the hash-table. If an entry is not found, then it is a new resource–we assign a new tag and add it to the hash-table, and then record its state. Otherwise, the resource has been saved before, so it suffices to save only the tag for later reference.

During restart the state is restored in the same order as has been saved originally, ensuring that the first appearance of each resource is accompanied with its actual recorded state. As with checkpoint, saved resource tags are examined by looking them up in the hash-table, and if not found, we create a new instance of the required resource, restore its state from the checkpoint image, and add it to the hash-table. If an entry is found, it points to the corresponding (already restored) resource instance, which is reused instead of creating a new one.

3.6 Leak Detection

In order to guarantee that a *container checkpoint* is consistent and reliable, we must ensure that the container is isolated and that its resources are not referenced by other processes from outside the container. When container shared resources are referenced from outside, we say that they *leak*; the ability to detect leaks is a prerequisite for container checkpoint to succeed.

Because the shared objects hash-table already tracks shared resources, it also plays a crucial role in detecting resource leaks that may obstruct a future restart. The key idea behind leak detection is to explicitly count the total number of references to each shared object inside the container, and compare them to the global reference counts maintained by the kernel. If for a certain resource the two counts differ, it must be because of an external (outside the container) reference.

Leak detection begins in a pre-pass that takes place prior to the actual checkpoint, to ensure that there are no external references to container shared objects. In this pass we traverse the process hierarchy like in the actual checkpoint but do not save the state. Instead, we collect the shared resources into the hash-table and maintain their reference counts. When this phase ends, the reference count of each object in the hash-table reflects the number of in-container references to it. We now iterate through all the objects and compare that count to the one maintained by the kernel. The two counts match[3] if and only if there are no external references.

[3] Actually, the hash-table count should be one less, because it does not count the reference that the hash-table itself takes.

This procedure is non-atomic in the sense that the reference counts from the hash-table and the kernel are compared only after all the resources have been collected. This is racy because processes outside the container may modify the state of shared resources, create new ones or destroy existing ones before the procedure concludes. For instance, consider two processes, one inside a container and the other not, that share a single file descriptor table. Suppose that after the pre-pass collects the file table and the files in it, the outside process opens a new file, closes another (existing) file, and then terminates. At this point, the new file is left out of the hash-table, while the other (closed) file remains there unnecessarily. Moreover, when the pre-pass concludes it will not detect a file table leak because the outside process exited, and the file table is only referenced inside the container. It will not detect a file leak even though the new file may be referenced outside, because the new file had not even been tracked.

To address these races we employ additional logic for leak detection during the actual checkpoint. This logic can detect in-container resources that are not tracked by the hash-table, or that are tracked but are no longer referenced in the container. Untracked resources are easy to detect, because their lookup in the hash-table will fail. To detect deleted resources, we mark every resource in the hash-table that we save during the checkpoint, and then at the end of the checkpoint, we verify that all the tracked resources are marked. A tracked but unmarked resource must have been added to the hash-table and then deleted before being reached by the actual checkpoint. In either case, the checkpoint is aborted.

3.7 Error Handling

Both checkpoint and restart operations may fail due to a variety of reasons. When a failure does occur, they must provide proper cleanup. For checkpoint this is simple, because the checkpoint operation is non-intrusive: the process hierarchy whose state is saved remains unaffected. For restart, cleanup is performed by the coordinator, which already keeps track of all processes in the restored hierarchy. More specifically, in case of failure the coordinator will send a fatal signal to terminate all the processes before the system call returns. Because the cleanup is part of the return path from the `restart` system call exit path, there is no risk that cleanup be skipped should the coordinator itself crash.

When a checkpoint or restart fails, it is desirable to communicate enough details about the failure details to the caller to determine the root cause. Using a simple, single return value from the system call is insufficient to report the reason of a failure. For instance, a process that is not frozen, a process that is traced, an outstanding asynchronous IO transfer, and leakage of shared resource leakage, are just a few failure modes that result all in the error `-EBUSY`.

To address the need to report detailed information about a failure, both `checkpoint` and `restart` system calls accept an additional argument: a file descriptor to which the kernel writes diagnostic and debugging information. Both the checkpoint and restart userspace utilities have options to specify a filename to store this log.

In addition, checkpoint stores in the checkpoint image informative status report upon failure in the form of (one or more) error objects. An error object consists of a mandatory pre-header followed by a null character (`'\0'`), and then a string that describes the error. By default, if an error occurs, this will be the last object written to the checkpoint image. When a failure occurs, the caller can examine the image and extract the detailed error message. The leading `'\0'` is useful if one wants to seek back from the end of the checkpoint image, instead of parsing the entire image separately.

3.8 Security Considerations

The security implications of in-kernel checkpoint-restart require careful attention. A key concern is whether the system calls should require privileged or unprivileged operation. Originally our implementation required tasks to have the `CAP_SYS_ADMIN` capability, while we optimistically asserted our intent to eventually remove the need for privilege and allow all users to safely use checkpoint and restart. However, it was pointed out that letting unprivileged users use these system calls is not only beneficial to users, but also has the useful side effect of forcing the checkpoint-restart developers to be more careful with respect to security throughout the design and development process. In fact, we believe this approach has succeeded in keeping us more on our toes and catching ways that users otherwise would have been able to escalate privileges through carefully manipulated checkpoint images, for instance bypassing `CAP_KILL` requirements by specifying arbitrary `userid` and signals for file owners.

At checkpoint, the main security concern is whether the process that takes a checkpoint of other processes in

some hierarchy has sufficient privileges to access that state. We address this by drawing an analogy between checkpointing and debugging processes: in both it is necessary for an auxiliary process to gain access to internal state of some target process(es). Therefore, for checkpoint we require that the caller of the system call will be privileged enough to trace and debug (using `ptrace`) all of the processes in the hierarchy.

For restart, the main concern is that we may allow an unprivileged user to feed the kernel with random data. To this end, the restart works in a way that does not skip the usual security checks. Process credentials, i.e. `UID`, `EUID`, and the security context[4] currently come from the caller, not the checkpoint image. To restore credentials to values indicated in the checkpoint image, restarting processes use the standard kernel interface. Thus, the ability to modify one's credentials is limited to one's privilege level when beginning the restart.

Keeping the restart procedure to operate within the limits of the caller's credentials means that scenarios consisting of privileged application that reduce their privilege level cannot be supported. For instance, a "setuid" program that opened a protected log file and then dropped privileges will fail the restart, because the user will not have enough credentials to reopen the file. The only way to securely allow unprivileged users to restart such applications is to make the checkpoint image tamper-proof.

There are a few ways to ensure the a checkpoint image is authentic. One method is to make the userspace utilities privileged using "setuid" and use cryptographic signatures to validate checkpoint images. In particular, checkpoint will sign the image and restart will first verify the signature before restoring from it. For instance, TPM [11] can be used to sign the checkpoint image and produce a keyed hash using a sealed private key, and to refuse restart in the absence of the correct hash. Another method is to create an assured pipeline for the checkpoint image, from the invocation of the checkpoint and restart system calls. Assured pipelines are precisely a target feature of SELinux, and could be implemented by using specialized domains for checkpoint and restart. Note, however, that even with a tamper-proof checkpoint image, a concern remains that the checkpoint image amounts to a persistent privileged token, which a clever user could find ways to exploit in new and interesting ways.

[4] Security contexts are part of Linux Security Modules (LSM).

4 Kernel Internal API

The kernel API consists of a set of functions for use in kernel subsystems and modules to provide support for checkpoint and restart of the state that they manage. All the kernel API calls accept a pointer to a checkpoint context (`ctx`), that identifies the operation in progress.

The kernel API can be divided by purpose into several groups: functions to handle data records; functions to read and write checkpoint images; functions to output debugging or error information; and functions to handle shared kernel objects and the hash-table. Table 2 lists the API groups and their naming conventions. Additional API exists to abstract away the details about checkpointing and restoring instances of some objects types, including memory objects, open files, and LSM (security) annotations.

The `ckpt_hdr_...` group provides convenient helper functions to allocate and deallocate buffers used as intermediate store for the state data. During checkpoint they store the saved state before it is written out. During restart they store data read from the image before it is consumed to restore the corresponding kernel object.

The `ckpt_write_...` and `ckpt_read_...` groups provide helper functions to write data to and read data from the checkpoint image, respectively. These are wrappers that simplify the handling, for example by adding and removing record headers, and by providing shortcuts to handle common data such as strings and buffers.

The `ckpt_msg` function writes an error message to the log file (if provided by the user), and, when debugging is enabled, also to the system log. The `ckpt_err` function is used when checkpoint or restart cannot succeed. It accepts the error code to be returned to the user, and a formatted error message which is written to the user-provided log and the system log. If multiple errors oc-

Group	Description
`ckpt_hdr_...`	record handling (alloc/dealloc)
`ckpt_write_...`	write data/objects to image
`ckpt_read_...`	read data/objects from image
`ckpt_msg`	output to the log file
`ckpt_err`	report an error condition
`ckpt_obj_...`	manage objects and hash-table

Table 2: Kernel API by groups

cur, e.g. during restart, only the first error value will be reported, but all messages will be printed.

The `ckpt_obj_...` group includes helper functions to handle shared kernel objects. They simplify the hash-table management by hiding details such as locking and memory management. They include functions to add objects to the hash-table, to find objects by their kernel address at checkpoint or by their tag at restart, and to mark objects that are saved (for leak detection).

Dealing with shared kernel objects aims to abstract the details of how to checkpoint and restart different object types, and push the code to do so near the native code for those objects. For example, code to checkpoint and restart open files and memory layouts appears in the `file/` and `mm/` subdirectories, respectively. The motivation for this is twofold. First, placing the checkpoint-restart code there improves maintainability because it make the code more visible to maintainers. Higher maintainers' awareness increases the chances that they will adjust the checkpoint-restart code when they introduce changes to the other native code. Second, it is more friendly to kernel objects that are implemented in kernel modules, because it means that the code to checkpoint-restart such objects is also part of the module.

To abstract the handling of shared kernel objects we associate a set of operations with each object type (similar to operations for files, sockets, etc). These include methods to checkpoint and restore the state of an object, methods to take or drop a reference to the objects so that it can be referenced when in the hash-table, and a method to read the reference count (in the hash-table) of an object. The function `register_checkpoint_obj` is used to register an operations set for an object type. It is typically called from kernel initialization code for the corresponding object, or from module initialization code as part of loading a new module. Each of the `ckpt_obj_...` takes the object type as one of its argument, which indicates the object-specific set of operation to use.

During checkpoint, for each shared kernel object the function `checkpoint_obj` is called. It first looks up the object in the hash-table, and, if not found, invokes the `->checkpoint` method from the corresponding operations set to create a record for the object in the image, and adds the object to the hash-table. During restart, records of shared kernel objects in the input stream are passed to the function `restore_obj`, which invokes the `->restore` method from the corresponding operations set to create an instance of the object according to the saved state, and adds the newly created object to the hash-table.

Several objects in the Linux kernel are already abstracted using operations set, which contain methods describing how to handle different versions of objects. For instance, open files have `file_operations`, and seeking in *ext3* filesystem is performed using a different method than in *nfs* filesystem. Likewise, virtual memory areas have `vm_operations_struct`, and the method to handle page faults is different in an area that corresponds to private anonymous memory than one that corresponds to shared mapped memory.

For such objects, we extend the operations to also provide methods for checkpoint, restore, and object collection (for leak detection), as follows:

To checkpoint a virtual memory area in a task's memory map, the `struct vm_operations_struct` needs to provide the method for the `->checkpoint` operation:

```
int checkpoint(ctx, vma)
```

and at restart, a matching callback to restore the state of a new virtual memory area object::

```
int restore(ctx, mm, vma_hdr)
```

Note that the function to restore cannot be part of the operations set, because it needs to be known before the object instance even exists.

To checkpoint an open file, the `struct file_operations` needs to provide the methods for the `->checkpoint` and `->collect` operations:

```
int checkpoint(ctx, file)
int collect(ctx, file)
```

and at restart, a matching callback to restore the state of an opened file:

```
int restore(ctx, file, file_hdr)
```

Here, too, the restore function cannot be part of the operations set. For most filesystems, generic functions are sufficient: `generic_file_checkpoint` and `generic_file_restore`.

To checkpoint a socket, the `struct proto_ops` needs to provide the methods for the `->checkpoint`, `->collect` and `->restore` operations:

```
int checkpoint(ctx, sock);
int collect(ctx, sock);
int restore(ctx, sock, sock_hdr)
```

Number of processes	Image size	Checkpoint time (to file)	Checkpoint time (no I/O)	Restart time
10	0.87 MB	8 ms	3 ms	10 ms
100	8.0 MB	72 ms	24 ms	72 ms
1000	79.6 MB	834 ms	237 ms	793 ms

Table 3: Checkpoint-restart performance

5 Experimental Results

We evaluated the current version of Linux Checkpoint-Restart to answer three questions: (a) how long checkpoint and restart take; (b) how much storage they require; and (c) how they scale with the number of processes and their memory size.

The measurements were conducted on a Fedora 10 system with two dual-core 2 GHz AMD Opteron CPUs with 1 GB L2 cache, 2 GB RAM, and a 73.4 GB 10025 RPM local disk. We used Linux 2.6.34 with the Linux-CR v21 patchset, with most debugging disabled and SELinux disabled. All optional system daemons on the test system were turned off.

For the measurements we used the *makeprocs* test from the checkpoint-restart test-suite [1]. This test program allows a caller to specify the number of child processes, a memory size for each task to map, or a memory size for each task to map and then dirty. We repeated the tests thirty times, and report average values.

Table 3 presents the results in terms of checkpoint image size, checkpoint time and restart time, for measurements with 10, 100 and 1000 processes. Checkpoint times were measured once with the output saved to a local file, and once with the output discarded (technically, redirected to `/dev/null`). Restart times were measured from when the coordinator begins and until it returns from the kernel after a successful operation. Restart times were measured reading the checkpoint images from a warm-cache.

The results show that the checkpoint image size and the time for checkpoint and restart scale linearly with the number of processes in the process hierarchy. The average total checkpoint time is about 0.8 ms per process when writing the data to the filesystem, and drops to well under 0.3 ms per process when filesystem access is skipped. Writing the data to a file triples the checkpoint time. This suggests that buffering the checkpoint output until the end of a checkpoint is a good candidate for a future optimization to reduce application downtime at checkpoint. Average total restart time from warm cache is about 0.9 ms per process. The total amount of state that is saved per process is also modest, though it highly depends on the applications being checkpointed.

To better understand how much of the checkpoint and restart times is spent for different resources, we instrumented the respective system calls to measure a breakdown of the total time. For both checkpoint and restart, saving and restoring the memory contents of processes amounted to over 80% of the total time. The total memory in use within a process hierarchy is also the most prominent component of the checkpoint image size.

To look more closely at the impact of the process size, we measured the checkpoint time for ten processes each with memory sizes increasing from 1 MB to 1 GB that the process allocated using the `mmap` system call. We repeated the test twice. In one instance the processes only allocate memory but do not touch it. In the second instance the processes also dirty all the allocated memory. In both cases, the output was redirected to avoid expensive I/O. The results are given in Table 4. The results show strong correlation between the memory footprint of processes and checkpoint times. Even when memory is untouched, the cost associated with scanning the process's page tables is significant. Checkpoint times increase substantially with dirty memory as it requires the contents to actually be stored in the checkpoint image.

Task size	Checkpoint time (clean)	Checkpoint time (dirty)
1 MB	0.5 ms	1.3 ms
10 MB	0.7 ms	6.4 ms
100 MB	1.7 ms	58 ms
1 GB	10.6 ms	337 ms

Table 4: Checkpoint times and memory sizes

6 Related Work

Checkpoint-restart has been the subject of extensive research [17, 18, 19], spanning all four approaches: application level, library mechanisms, operating system mechanisms, and hardware virtualization; See [12] for a detailed discussion on these approaches.

Many application checkpoint-restart mechanisms have been implemented in Linux, some in userspace [7, 8] and others in the kernel [5, 9, 14, 12, 20]. Ckpt [7] modifies tasks to let them checkpoint themselves. The checkpoint images are in the form of executable files which, when executed, restart the original process. CryoPID [8] also uses an executable file for the checkpoint image, but relies on `/proc` information to checkpoint a task. Userspace approaches do not capture or are unable to restore some parts of a process's system state, and are limited in which applications they support. In contrast, Linux-CR is an operating system mechanism that can save and restore all relevant state, and can do so completely transparently to the application.

EPCKPT [9] and CRAK [20] provide partial support for checkpoint-restart for Linux 2.4 series, as a kernel patch and kernel module respectively. BLCR [5] is aimed primarily at HPC users. It consists of a library and a kernel module. Applications must be checkpoint-aware so as to discard unsupported resources. None of these provide virtualization. Zap [12, 16] and OpenVZ [14] implement both containers and checkpoint-restart. OpenVZ consists of an invasive out-of-tree kernel patch, and Zap is a kernel module.

The Linux-CR project emerged as a unifying framework to provide checkpoint-restart in the Linux kernel. It builds on Zap, but it is implemented in the kernel. Linux-CR improves on earlier work in that the design and implementation are done in a clean way and geared for inclusion in the mainstream Linux kernel, so that it will be maintained as part of the kernel.

Linux-CR can detect when resources leak outside of containers. This leak detection logic was inspired by OpenVZ, which was the first to propose this mechanism. However, leak detection in OpenVZ is incomplete because it does not handle race condition when scanning for resources, allowing untracked and deleted resources to remain undetected. Linux-CR addresses this by extending the mechanism to explicitly discover and handle resources that escape the initial scan.

7 Conclusions

Previous work on application checkpoint and restart for Linux does not address the crucial issue of integration with the mainstream Linux kernel. We present in detail Linux-CR, an implementation of checkpoint-restart, which aims for inclusion in the mainline Linux kernel. Linux-CR provides transparent, reliable, flexible, and efficient application checkpoint-restart. We discuss the usage model and describe the user interfaces and some key kernel interfaces of Linux-CR. We present preliminary performance results of the implementation.

A key ingredient to a successful upstream implementation is the understanding by the kernel community of the usefulness of checkpoint-restart. Without this we would fail to receive from the community the invaluable review and advice upon which we have relied. That we have in fact received much such help shows that the usefulness of checkpoint-restart is recognized by the community. We therefore have high hopes that, with the community's help, the project will succeed in providing checkpoint-restart functionality in the upstream kernel.

Legal Statement

Acknowledgments

The Linux-CR project is the result of the hard work of several developers. Major contributors include Matt Helsley, Dave Hansen, Sukadev Bhattiprolu, Dan Smith, and Nathan Lynch. Additional guidance has come from Arnd Bergmann, Ingo Molnar, Louis Rilling, Alexey Dobriyan and Andrey Mirkin. Various reviewers gave countless valuable comments for the patchset. Jason Nieh provided helpful comments on earlier drafts of this paper. This work was supported in part by the DARPA under its Agreement No. HR0011-07-9-0002, by NSF grants CNS-0914845 and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

References

[1] Checkpoint/restart testsuite. `http://www.linux-cr.org/git/?p=tests-cr.git;a=summary`.

[2] Libvirt LXC container driver. `http://www.libvir.org/drvlxc.html`.

[3] Linux Containers. `http://lxc.sf.net`.

[4] S. Bhattiprolu, E. W. Biederman, S. E. Hallyn, and D. Lezcano. Virtual Servers and Checkpoint/Restart in Mainstream Linux. *SIGOPS Operating Systems Review*, 42(5), 2008.

[5] Berkeley Linux Checkpoint/Restart User's Guide. `http://mantis.lbl.gov/blcr/doc/html/BLCR_Users_Guide.html`.

[6] BTRFS. `http://oss.oracle.com/projects/btrfs`.

[7] ckpt. `http://pages.cs.wisc.edu/~zandy/ckpt/`.

[8] CryoPID - A Process Freezer for Linux. `http://cryopid.berlios.de`.

[9] EPCKPT. `http://www.research.rutgers.edu/~edpin/epckpt/`.

[10] EXT4. `ext4.wiki.kernel.org`.

[11] T. Group. Tcg tpm specification version 1.2 - part 1 design principles, 2005.

[12] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[13] LXC: Linux container tools. `http://www.ibm.com/developerworks/linux/library/l-lxc-containers`.

[14] A. Mirkin, A. Kuznetsov, and K. Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the 2008 Ottawa Linux Symposium*, July 2008.

[15] Network Appliance, Inc. `http://www.netapp.com`.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

[17] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Dept. of Computer Science, University of Tennessee, July 1997.

[18] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, July 2002.

[19] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, Washington, DC, Apr. 2005.

[20] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart As a Kernel Module. Technical Report CUCS-014-01, Columbia University , 2001.