

Comparison of the Existing Checkpoint Systems

Byoung-Jip Kim/Watson/IBM
kimby@us.ibm.com
Oct 12, 2005

Contents

1. Comparison of each system
2. Summary of existing systems
 - 2.1 CRAK
 - 2.2 BLCR
 - 2.3 Epckpt
 - 2.4 ckpt
 - 2.5 CryoPID
 - 2.6 tcpcp
 - 2.7 Zap
 - 2.8 Duration

1. Comparison of Each System

Checkpointing means that we save the running state of a program into an image file. Restart then restarts the program based on the image file. So the basic questions are 1) which states need to be saved; 2) how to save these states; and 3) how to restore the states.

The state of a program in execution has many aspects. Basically, all the process private data need to be saved and recovered during checkpointing. This includes: address space, register set, open files/pipes/sockets, System V IPC structures, current working directory, signal handlers, timers, terminal settings, user identities (uid, gid, etc), process identities (pid, pgrp, sid, etc), rlimit etc. Each checkpoint system saves and restores only some of them. Address space and register set are the two mandatory components we have to save and restore, while others can be optional depending on the type of applications. Table 1 shows the supported features of each C/R system. For example, according to Table 1, the initial version of BLCR supports multithreaded programs, but no open files. Also, TCP, UDP, and other network resources are not supported. Ckpt saves and restores the entire address space and the signal state of a process, but not others such as open files, IPC state, and etc.

There are two basic categories of C/R systems: kernel level and user level. CRAK, BLCR, and Epckpt are kernel level, while ckpt is user level. They have different levels of transparency, complexity, portability and performance characteristics. Transparency means whether user applications need to be modified, recompiled or relinked. Generally speaking, adding support to kernel leads to better transparency, but more implementation complexity and less portability.

In the user-level approach, the operating system is unmodified, and remains unaware of checkpoint and restart. In order to save and restore the state of a program, user level checkpoint/restart implementations intercept a wide range of system calls and save program state in user-space. However, it seems undesirable to replicate kernel data structures in user-space: this type of shadowing is easy to get wrong. In contrast, a kernel-level checkpoint implementation can simply access the data structures it needs in the kernel, reducing the potential for such inconsistency.

Table 1. Comparison of Supported Features

Feature	CRAK	BLCR	Epckpt	ckpt	CryoPID	Duaction	Zap	MCR
Version	Linux 2.4	Linux 2.6	Linux 2.4.2	Linux 2.6	Linux 2.6	Linux 2.4	Linux 2.4	Linux 2.6
Implementation	kernel module + utilities	kernel module + library + utilities	kernel patch + utilities	library + utilities	n/a	n/a	kernel module + library	kernel patch + kernel module + libraries + utilities
Year	2001	2005	2001	2005	2005	2005	2002	2005
Author	Columbia	Berkeley Lab	Rio de Janeiro	Wisconsin	Bernard Blackham	Eternal Systems	Columbia	Miosys
Single-thread process (Address space, Register set)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multi-thread process	No	Yes	No	No	Planned	Yes		Yes
Process groups	Yes	Planned	Yes		Planned	Yes	Yes	Yes
Sessions		Planned						
pid & ppid	Yes	Yes	Yes	No			Yes	Yes
uid & gid		Planned		No				
Signal state/handlers	Yes	Yes		Yes	Yes		Yes	Yes
Resource limits	No	Planned		No				
Timers			Planned	No				
Current working directory	Yes	No	Planned	No				
SYS V IPC	No	No	SEM SHM	No			Yes	Yes
Regular files	Yes	Planned	Yes	No	Yes	Yes	Yes	Yes
stdin/stdout/stderr	Yes	Yes	Yes	No		Yes	Yes	Yes
Mmapped files	Yes	Yes	Yes		Yes		Yes	Yes
/proc files	No	/proc/<pid>		No				
Unnamed & named pipes	Yes	Planned	Yes	No			Yes	Yes
TCP/UDP sockets	Yes	No	No	No	Yes		Yes	Yes
Device files	No	/dev/null, /dev/zero	No	No			Yes	limited
MPI programs	No	Yes	No	No				Yes

Yes – A system can checkpoint the resource.

No – A system cannot checkpoint the resource.

Planned – Not support now, but it is planned to implement.

2. Summary of the Existing Systems

2.1 CRAK: Linux Checkpoint / Restart As a Kernel Module

1. Author: Columbia University
2. Year: 2001
3. Availability of source code: Linux 2.2 and Linux 2.4
4. Structure: one kernel module + utilities
5. Features
 - a. Refer to Table 1 for the details.
 - b. Support for process groups.
 - c. Support for open files.
 - d. Support for network sockets.
6. Design
 - a. Built with existing operating systems
 - b. Transparent to user applications
 - c. No kernel modification
 - d. Support parallel processes
 - e. Support network applications
 - f. Good performance
7. Implementation
 - a. Overview
 - i. Many packages use Unix signal handling mechanism to do checkpoint. When a process needs to be checkpointed, a special signal is sent to it. The real work is done by the signal handler. However, with a module, a special signal and a default signal handler cannot be added. As a result, a process cannot be checkpointed in its own context; instead it can be checkpointed from another process. The basic problem is then how to access an arbitrary process' private data from a kernel module, especially the address space and register set.
 - ii. A device file interface, /dev/ckpt, is implemented. It provides ioctl interface for checkpoint and restart. This device file is encapsulated in a helper library (ckptlib.c).
 - b. Checkpointing
 - i. Basic step
 1. Checkpoint command is sent to the kernel module on the local machine.
 2. Kernel module stops processes to be checkpointed.
 3. Kernel module saves the process state to file. The original process is then killed.
 4. Kernel module takes care of saving appropriate socket and file states.
 - ii. Address space: CRAK simply walks through all the segments, and for each one it saves its position, access attribute and content to a checkpoint image file. To reduce the overhead, an optimization is required when saving an address space of a process. The basic idea is that read-only and shared segments of an address space need not to be saved into the checkpoint images.
 - iii. Register set: For a register set, the only problem is finding where it is located. In Linux, a process' task structure is within the same 8KB frame

with its kernel stack, and the register set is saved at the top of the stack. Therefore, the location of the register set can be calculated.

- iv. Opened files: What needs to be saved for checkpointing a process is a pathname of a file. When restarting a migrated process, it has to be ensured that these files are opened with exactly the same file descriptors as before.
- v. Network sockets: During the socket migration, it needs to be changed the sock information, especially address and port. An important thing is that socks are put in various hash tables to make lookup more efficient. When changing its address and/or port, it has to be sure that the sock is still in the correct hash tables.

- 1. Half-close a connection

- a. When a process is checkpointed, it sends a FIN packet to the peer process.
- b. CRAK manually sets the sock state to `TCP_TIME_WAIT`.
- c. Kernel module saves TCP stack information such as the sequence number.

- 2. Half-recover a connection

- a. When a process is restarted, kernel module rebuilds all the sockets it previously had.
- b. In user space, `socket()` and `bind()` is called to create the socket.
- c. Kernel module fills in peer address, peer port and other TCP stack information it saved.
- d. Kernel module needs to rehash the TCP stack information, and fresh the rtable cache.
- e. At last, CRAK manually set the state to `TCP_ESTABLISHED`.

- 3. Fully Recover a connection

- a. In peer machine, CRAK needs to change the daddr and dport of the previous machine to the new machine.

- c. Restarting

- i. It's very similar to the system call "execve".

- 1. Create a new process
- 2. Maintain parent/child relationships for parallel processes
- 3. Recover process address space by mapping memory sections from the checkpoint image
- 4. Recover register set
- 5. Reopen files with the same descriptor numbers
- 6. Recover network socket

8. Pros

9. Cons

- a. CRAK does not support virtualization. Therefore, not all the data can be guaranteed to recover. For example, if the PID number is already used, it cannot be recovered. If the file to open is already opened and locked by another process, it cannot be opened again.
- b. CRAK does not support a multi-threaded process.

10. Reference

- a. Hua Zhong and Jason Nieh, CRAK: Linux Checkpoint/Restart As a Kernel Module, Technical Report, Columbia University.

2.2 BLCR: Berkeley Linux Checkpoint/Restart

1. Author: Lawrence Berkeley National Laboratory
2. Year: 2004 ~ current
3. Availability of source code: Linux 2.4 (stable), Linux 2.6 (test)
4. Structure: two kernel modules + libraries + utilities
5. Features
 - a. Refer to Table 1 for the details.
 - b. Support for a multi-threaded process
6. Design
 - a. BLCR is implemented as a Linux kernel module and a user-level shared library.
 - b. Kernel module
 - c. User-level interface
 - i. Callbacks: BLCR provides a way to register user-level callback functions, which are triggered whenever a checkpoint is about to occur, and which continue when a restart is initiated. These callbacks allow the application to release uncheckpointable resources such as network sockets and open files before a checkpoint is taken.
 - ii. Critical sections: BLCR also provides user-level code with “critical sections,” in order to allow groups of instructions to be performed atomically with respect to checkpoints.
7. Implementation
 - a. Signal-based callbacks
 - i. To initiate a checkpoint within an application, BLCR sends it a signal. User application that needs to be checkpointed must be loaded with the user-level BLCR checkpoint library, which registers a signal handler for the checkpoint signal.
 - b. Thread-based callbacks
 - i. Because a set of standard library functions that are supported with signal handler context is quite limited, the signal-based callback has been expanded to thread-based callbacks.
 - c. Checkpointing
 - i. An application starts with BLCR checkpoint library which has registered a threaded callback. This thread blocks in the kernel until a checkpoint occurs.
 - ii. BLCR checkpoint utility initiates a checkpoint by opening up a special file (/proc/checkpoint/status: this file is created when the checkpoint module is first loaded), then calling ioctl() on it with a structure argument containing the type of checkpoint, a target identifier, and a file handler to which the checkpoint image will be written.
 - iii. In the first stage, the ioctl() call unblocks the callback thread in the application. The callback thread returns user-space and runs the application's thread-based callbacks. When all thread-based callbacks have entered cr_checkpoint, the callback thread reenters the kernel, and triggers the second stage of the checkpoint.
 - iv. In the second stage, the checkpoint signal is sent to each of the remaining threads in the application. If the signal is delivered, a BLCR library call is invoked. This routine runs any signal-based callbacks the application has provided, and then enters the kernel.

- v. In the third state, all the callback threads complete their callbacks and enter the kernel. The threads leave the barrier, and one is chosen to write out a header of the checkpoint file, which contains information on the threads and their parent/child relationships.
 - vi. In the fourth stage, the remaining threads write their registers, signal information, and pid into the checkpoint file.
 - vii. Once all the threads in a process have completed dumping their state, they enter the final barrier.
 - d. Restaring
 - i. The restart utility performs an ioctl call, which results in its being forked. The parent returns to user space, and waits for the restart to complete, while the child is cloned by the kernel as many as the application needs.
 - ii. The newly cloned threads perform restoring their information from the checkpoint file.
 - iii. After the final barrier the threads exit the kernel and enter user space.
 - iv. In user space, the callbacks return from cr_checkpoint and continue until they exit.
 - v. Finally, application code is resumed.
- 8. Pros
 - a. Because BLCR is implemented as a kernel module, it is easy to deploy. Kernel does not need to be patched, recompiled, and rebooted.
 - b. BLCR supports checkpointing a multi-threaded process.
 - c. BLCR provides a method guaranteeing that atomic operations are performed without interrupts when processes are checkpointed.
- 9. Cons
 - a. BLCR does not support virtualization such as process virtualization, IPC virtualization and network virtualization. Therefore, when a process is restarted, resource conflict problem may occur.
 - b. BLCR does not support for checkpointing network sockets.
 - c. BLCR does not support for checkpointing open files.
- 10. Reference
 - a. Jason Duell, The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart
 - b. Berkeley Linux Checkpoint/Restart User's Guide – version 0.4.0, http://mantis.lbl.gov/blcr/doc/html/BLCR_Users_Guide.html
 - c. BLCR Administrator's Guide, http://mantis.lbl.gov/blcr/doc/html/BLCR_Admin_Guide.html

2.3 Epckpt

1. Author: Federal University of Rio de Janeiro, Brazil
2. Year: 2001
3. Availability of source code: Linux 2.4.2
4. Structure: kernel patch + utilities
5. Features
 - a. Refer to Table 1 for the details.
 - b. Support for process groups
6. Design
 - a. Transparent checkpoint/restart
 - b. Minimize checkpoint image size by not checkpointing shared libraries and code sections
 - c. Directly send checkpoint image to remote host
 - d. Checkpoint a group of processes in the one-parent-many-children fashion
 - e. Save and continue of save and kill policies
7. Implementation
 - a. Overview
 - i. It installs a default signal handler in the kernel. When a checkpoint is triggered, the kernel sends a signal. The signal handler then will be activated and checkpoint the current process.
 - b. Three new system calls were created in Linux
 - i. `checkpoint (int pid, int fd, int flags)`: It can checkpoint all processes descendent process pid. It also supports checkpointing processes running on SMP.
 - ii. `restart (char *ckpt_filename)`
 - iii. `collect_data (int pid)`: It tells the kernel to log some information of the given process during run-time so that the process can be checkpointed later on.
 - c. Some utilities
 - i. `spawn`: It takes the filename of an application to run. It calls `collect_data()` to have some necessary information of the process logged by the kernel. Then, it forks off a child and this child calls `exec` system call in order to run the application.
 - ii. `checkpoint`
 - iii. `restart`
 - iv. `split, mrestart`: When the checkpoint is done over a group of parallel processes, they are all dumped in a single image. Before it can be restarted, this image must be split into individual process image. `Split` performs this job. `Mrestart` restarts the split process image repeatedly.
8. Pros
 - a. Epckpt minimizes the size of a checkpoint file by not storing read-only data such as code segment in address space.
9. Cons
 - a. Epckpt adds excessive information logging to the kernel in order to checkpoint a process.
 - b. Epckpt introduces a lot of overhead. It patches open and close system calls in order to know which files are opened. It also patches `mmap`, `fork`, `exit`, etc. This makes the implementation rather complex.
 - c. Even though Epckpt is designed to checkpoint/restart parallel processes, it is not flexible in supporting parallel processes. For example, splitting a checkpoint file

into separate ones and starting them repeatedly have to be done when restating parallel processes.

10. Reference

- a. Edurado Pinheiro, Truly-Transparent Checkpointing of Parallel Applications (Working Draft), Federal University of Rio de Janeiro
- b. <http://www.research.rutgers.edu/~edpin/epckpt/>

2.4 ckpt

1. Author: Wisconsin
2. Year: 1997 ~ current
3. Availability of source code: Linux 2.6
4. Structure: libraries + utilities
5. Features
 - a. Refer to Table 1 for details.
6. Design
 - a. ckpt inserts code into the program, that enables the program to checkpoint itself. ckpt supports asynchronous checkpoints triggered by signals sent by other programs.
 - b. ckpt writes its checkpoint of a program to a checkpoint file. A checkpoint file is an ordinary executable (in ELF format) that, when executed, continues the program from the point at which it was checkpointed.
 - c. Checkpoint files can be restarted on a different machine from the one on which the checkpoint was taken.
 - d. The ckpt API provides hooks for calling third-party code at stages of a program checkpoint and restart. Users can add their own code to these hooks to augment the checkpointing functionality with, for example, code to save open file descriptors.
7. Implementation
 - a. Checkpointing
 - i. A process linked with ckpt can be asynchronously checkpointed by sending the SIGSTP signal to it.
 - ii. Optionally, a process can be automatically checkpointed periodically.
 - iii. User code can call routines in ckpt.h to synchronously checkpoint itself.
 - b. Restarting
 - i. Checkpoint files are executables. To restart a checkpoint, execute the checkpoint file like an ordinary executable.
8. Pros
9. Cons
 - a. ckpt can checkpoint only the mandatory process resources such as address space and register set. If various resources used by a process need to be checkpointed, the process should be ckpt-aware so that it can run callbacks. It means that the process should be modified.
10. Reference
 - a. M. Litzkow, T. Tannenbaum, J. Basney, M. Livny. Checkpoint and Migration of UNIX Process in the Condor Distributed Processing System. University of Wisconsin Madison.
 - b. J. Pruyne, M. Livny. Managing Checkpoints for Parallel Programs. University of Wisconsin Madison.
 - c. <http://www.cs.wisc.edu/~zandy/ckpt>

2.5 CryoPID

1. Author: Bernard Blackham
2. Year: 2005
3. Availability of source code: Linux 2.4
4. Structure:
5. Features
 - d. No root privileges needed.
 - e. No kernel modifications to the kernel
 - f. No recompiling/relinking
 - g. No need to use LD_PRELOAD
 - h. Can migrate processes between machines
 - i. Can migrate processes between kernel versions (tested between 2.4 to 2.6 and 2.6 to 2.4)
6. Design
 - j. A program called freeze captures the state of a running process and writes it into a file.
 - k. The file is self-executing and self-extracting. To resume a process, you run that file.
7. Implementation
8. Pros
9. Cons
10. Reference
 - a. <http://cryopid.berlios.de>

2.6 tcpcp

tcpcp (TCP Connection Passing) provides an mechanism that enables cooperating applications to pass ownership of TCP connection endpoints from one Linux host to another one. tcpcp can be used as a building block for solutions of process migration and fail-over. CryoPID mentioned above uses tcpcp for network checkpoint.

1. Author: Werner Almesberger
2. Year: 2005
3. Availability of source code: Linux 2.6
4. Structure: kernel patch + library
5. Features
 - a. Pass live TCP connection from one host to another
 - b. Transparent to peer (no wrapper library, proxy, etc.)
 - c. As transparent as possible to application (on host)
6. Design
7. Implementation
 - a. Passing the connection
 - i. The application at the origin initiates the procedure by requesting retrieval of Internal Connection Information (ICI) of a socket.
 - ii. tcpcp isolates the connection: all incoming packets are silently discarded, and no packets are sent.
 - iii. The kernel copies all relevant variables.
 - iv. After retrieving the ICI, the application empties the receive buffer.
 - v. The origin sends the ICI and any relevant application state to the destination.
 - vi. The destination opens a new socket.
 - vii. The application at the destination now sets the ICI on the socket.
 - viii. Network traffic belong to the connection is redirected from the origin to the destination host.
 - ix. The application at the destination makes a call to activate the connection.
 - x. If there is data to transmit, the kernel will do so.
8. Pros
9. Cons
 - a. Not transparent to application.
10. Reference
 - b. Werner Almesberger, "TCP Connection Passing", Ottawa Linux Symposium, July 2004

2.7. Zap

Zap is a system for transparent migration of legacy and networked applications. Zap uses partial OS virtualization to allow the migration of process domains (pods), essentially process groups, using a modified Linux kernel. Their approach is to isolate all process-to-kernel interfaces, such as file handles and sockets, into a contained namespace that can be migrated. Zap completes CRAK by adding an OS virtualization layer.

1. Author: Steven Osman et al. (Columbia University)
2. Year: 2002
3. Availability of source code: Linux 2.4
4. Structure: kernel module + libraries
5. Features
 - a. Transparent to applications
 - b. No kernel modification
 - c. Support network socket migration
 - d. Support legacy IPC mechanism
6. Design
 - a. Implemented as a kernel module
 - b. Checkpoint-restart mechanism of CRAK
 - c. Thin virtualization layer on top of OS
 - i. Solve naming conflict by virtualization
 - d. Pod abstraction
 - i. Supports legacy IPC mechanism
 - e. Supports network socket connection migration
7. Implementation
 - a. PID and PIC Key Virtualization & Migration
 - i. Create unique namespace for the POD
 - ii. Names are virtualized
 - iii. When entering a system call, replace POD virtual identifiers with real ones.
 - iv. When exiting a system call, replace real return values with POD virtual ones.
 - v. Mask out identifiers that do not belong to the POD.
 - b. File System Virtualization & Migration
 - i. Make private file system by using chroot and nfs
 - ii. Assume centralized nfs server to mount
 - iii. Mount shared files (e.g. executable program files, read-only files)
 - iv. Save private files to local file system (e.g. configuration files)
 - c. Networking Virtualization & Migration
 - i. Two network addresses
 1. Persistent internal address
 2. Host-dependent external address
 - ii. For connection migration
 1. Transport layer sees virtual address
 2. Network layer sees real address
 3. Transport layer independent
 4. Initial virtual address is real address
8. Pros
 - a. Zap achieves the goal of process migration relatively simple using virtualization.

- b. Their approach is considerably faster than previous work, largely due to the smaller units of migration.

9. Cons

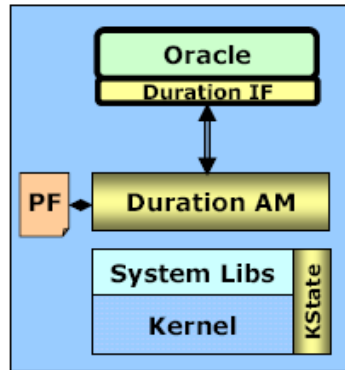
- a. In terms of security, intercepting system calls and replacing them in the kernel can incur security flaws.
- b. In terms of the migrating cost, migration in their system is still on the order of seconds at best. Pods are entirely suspended, copied, and then resumed.
- c. From the aspect of network virtualization, they do not address the problem of maintaining open connections for existing services. To avoid downtime due to migration, Zap needs to install a temporary proxy process just before a pod migrates. This approach is not appropriate for network servers, because of the relatively down time.
- d. Zap is relatively weak to support for various devices that could be used by process in a pod.

10. Reference

- a. Steven Osman, et al, "The design and Implementation of Zap: A System for Migration Computing Envrionments", OSDI 2002

2.8 Duration

1. Author: Eternal Systems
2. Year:
3. Availability of source code:
4. Structure:



Eternal **Duration**™ overview.

Duration is a high availability infrastructure that enables deployment of enterprise applications in a variety of configurations.

5. Features
 - a. Transparent and automatic checkpointing of memory and storage.
 - i. checkpointed state: file descriptors, file pointers, and file contents.
 - b. Flexible checkpointing of multi-process, multi-threaded applications.
 - c. Virtual IP addressing for transparent client redirection.
 - d. User-initiated process migration and process launch.
6. Design
7. Implementation
8. Pros
9. Cons
10. Reference:
 - a. Eternal Systems, Protecting HPC Cluster Applications – White Paper, <http://www.eternal-systems.com>
 - b. Eternal Systems, Eternal Duration – Data Sheets, <http://www.eternal-systems.com>