Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Prototype of the first advanced version of LinuxSSI

D2.2.10

Due date of deliverable: May 31^{th}, 2009
Actual submission date: June 12^{th}, 2009

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP2.2
Task number: T2.2.3,T2.2.4,T2.2.5,T2.2.6

Responsible institution: INRIA
Editor & and editor's address: Christine Morin
IRISA/INRIA
Campus de Beaulieu
35042 RENNES Cedex
France

Version 1.0 / Last edited by Christine Morin / June 10, 2009

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	23/04/09	Eugen Feller	INRIA/UDUS	Initial check-in, update incremental checkpoint/restart, summary, LinuxSSI services and conclusion. Add a description of callback implementation
0.2	20/05/09	Christine Morin	INRIA	Introduction/conclusion
0.3	28/05/09	John Mehnert-Spahn	UDUS	Add publication
0.4	05/06/09	Eugen Feller	INRIA/UDUS	Haiyan review modifications
0.5	10/06/09	Eugen Feller	INRIA/UDUS	Samuel review modifications

Reviewers:
Samuel Kortas (EDF) and Haiyan Yu (ICT)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.2.3	Design and implementation of advanced chekckpoint/restart mechanisms	INRIA*,UDUS
T2.2.4	Design and implementation of advanced reconfiguration mechanisms	INRIA*,NEC
T2.2.5	Design and implementation of LinuxSSI distributed file system advanced feature	INRIA*
T2.2.6	Design and implementation of a customizable scheduler	XLAB*,INRIA

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

This document is a companion report to the XtreemOS cluster flavour prototype. It reports on the work done as part of WP2.2 during the last six months (M30-M36) on the design and development of XtreemOS cluster flavour. LinuxSSI is the heart of the foundation layer for XtreemOS cluster flavour. LinuxSSI leverages Kerrighed single system image cluster operating system developed in open source (http://www.kerrighed.org). Most of our recent work has focused on the design and implementation of new features in checkpoint/restart mechanisms. We have improved the support for incremental checkpointing and the support for callbacks has been implemented.

Checkpointing overhead can be reduced by saving only contents that have changed since the last checkpoint. Incremental checkpointing in LinuxSSI is based on a page based granularity. During the last six months, we have significantly improved the preliminary implementation available in November 2008. All needed mechanisms are now available: memory page modification detection based on the write bit, book keeping of modified pages based on a Linux-native radix tree, memory region modification detector able to detect region extension, creation, shortening and splitting. Some performance results are reported.

Kernel checkpointers in general lack application semantic knowledge. Hence they usually have no information about what needs to be saved and restored. Callbacks give the user the possibility to determine what resources need to be checkpointed and what to be restored. They are registered by the user and executed before taking a checkpoint. Due to the possibility of enabling the user to decide which ressources need to be checkpointed or not a performance increase can be achieved.

We have produced LinuxSSI-1.0-rc1 release in May 2009. It is based on the most recent Kerrighed development and includes the new checkpoint/restart features as well as all features from the previous LinuxSSI releases. XtreemOS cluster flavour is available on the most recent XtreemOS installation CD. XtreemOS-G services (AEM daemon, XtreemFS client) are able to run on top of LinuxSSI, which also supports the system level VO support mechanisms.

Contents

1 Introduction

This document describes the results of the work done with workpackage WP2.2 during the last six months of the XtreemOS project (M30-M36) on the cluster flavour of the XtreemOS system.

LinuxSSI is the heart of the foundation layer for XtreemOS cluster flavour. LinuxSSI leverages Kerrighed single system image cluster operating system developed in open source (`http://www.kerrighed.org`). Most of our work as part of WP2.2 focuses on the design and implementation of new features in LinuxSSI.

In this document, we describe the main features that have been developed in XtreemOS cluster flavour since the November 2008 LinuxSSI release. The new features relate to application checkpoint/restart developed as part of Task T2.2.3 *Design and implementation of advanced checkpoint/restart mechanisms*. Checkpointing overhead can be reduced by saving only contents that have changed since the last checkpoint. Incremental checkpointing in LinuxSSI is based on a page based granularity. Section 2 focuses on the improvements to the incremental checkpointing mechanisms introduced in Deliverable [3]. Kernel checkpointers in general lack application semantic knowledge. Hence they usually have no information about what needs to be saved and restored. Callbacks give the user the possibility to determinate what resources need to be checkpointed and what to be restored. They are registered by the user and executed before taking a checkpoint. Due to the possibility of enabling the user to decide which ressources need to be checkpointed or not a performance increase can be achieved. Section 3 describes the callback mechanisms implemented for LinuxSSI kernel checkpointer. Concluding remarks and future directions of work for LinuxSSI are given in Section 4.

2 Incremental Checkpointing

2.1 Memory page modification detection

Checkpointing overhead can be dramatically reduced by saving only content that has changed since the previous checkpoint, [7]. The major challenge here is to *detect* these content modifications. Generally, there are page-based and variable based approaches.
Detection of content changes at variable-granularity-level is described in [5] where a compiler is manually modified to detect variable changes. Thus, the compiler is enabled to insert incremental state saving calls before a variable is changed. In [8] detection of variable changes is enabled without manual intervention, but via an executable editing library. Furthermore, special memory hardware exists that incrementally saves state of contained variables [4].
Detection of modified pages can be realised by taking existing page table entry bits, namely the *dirty bit* or the *write bit*, into account.
The *dirty bit* is of high importance for the Linux internal memory management components, especially for the Page Cache. A set *dirty bit* indicates to synchronize cache contained pages with those versions on disk and the swap partition. Just reading the *dirty bit* does not always inidicate changed content. After modified cache contained data are written back to disk, the bit is reset, thus, a past modification is not visible anymore. Bookkeeping of modified pages includes resetting the dirty bit to detect new modifications after a taken snapshot, which is dangerous since it affects Page Cache consistency. In [1] page modification detection is done based on the *dirty bit* being mirrored into one of the reserved entry bits.

Our approach to detect modified pages is *write bit* focussed. Each time a write-protection page-fault occurs, the *write bit* is set. Such exceptions are detected by the memory management unit. An exception handler is called and resolves the exception by removing the write-protection (*write bit* set to 1). Based on the detection we record modified pages in a bookkeeping control structure. At the initial checkpoint all pages of a program address space are saved. After each checkpointing operation all writable pages are explicitly made write-protected (*write bit* is reset to 0). In case an application attempts to write on such a page within a checkpoint interval, the triggered page-fault handler removes the write-protection. Thus, detection of modified pages is enabled during the next checkpointing operation. Depending on the application behaviour, generally just a subset of all application pages needs to be saved. In order to detect future write attempts in subsequent checkpoint intervals, the write-protection will be reactivated by explicitly resetting the page *write bit*. Special handling of newly added read-only pages and partially written pages after a checkpointing operation is detailed under 2.3.
During restart the last version of a physical page needs to be located out of multiple checkpoint images. Therefore, a dedicated bookkeeping control structure is required that keeps track of the last page's file location and offset within the checkpoint image file, described in the following section.
We are conscious about TLB entries, which must be flushed explicitly after each incremental checkpoint, since they are not updated with our *write bit* modification. The latter could result in inconsistency. However, each process context switch anyway results in flushing the TLB. Taking multiple checkpoints within one scheduler time slice is hard to achieve due to its shortness.

2.2 Bookkeeping of modified pages

Physical page content of an application address space may be spread across multiple incremental checkpoint images each potentially storing all or just a subset of all pages. At restart the complete content and its consistent versions must be loaded.
We use a bookkeeping control structure that keeps track of page locations and is based on the Linux-native *radix tree*. The *radix tree* provides fast lookup and insertion operations (O(1)) which are needed to keep the structure in sync with the process memory structure. In figure 1 the localisation of a virtual address-related physical page is shown. Each tree node is identified by a virtual address. A

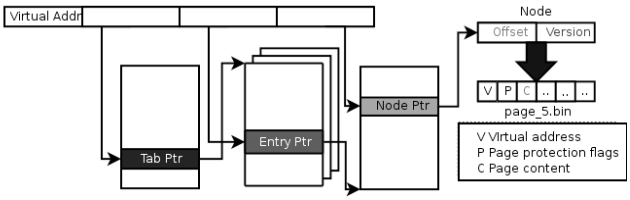

Figure 1: Bookkeeping control structure for modified pages.

node entry stores the version of a dedicated incremental checkpoint image file (e.g. pages_5.bin for the fifth incremental checkpoint) containing the latest version of the page, and the offset in this file, since more than one page may be modified between two incremental checkpoints. The incremental

checkpoint image file stores data blocks each containing a virtual address, page protection flags and the page content itself. Of course, all node entries are also saved to disk at each incremental checkpoint.
During a checkpointing operation the bookkeeping control structure is updated. That means, bookkeeping entries targetting not yet referenced pages are added, file locations and offsets of pages that are *present and modified and already referenced* are being updated and saved to disk.
At restart the bookkeeping control structure is read from disk. Its data is used to localize memory pages out of multiple incremental checkpoint files to restore a process' address space.
The mere write-bit based page modification approach alone is not able to keep the bookkeeping control structure in sync with the process memory structure, especially if memory pages have been removed. A requirement is to delete such structure entries to avoid wasting memory. Reading from and writing structure content to disk decreases checkpointing performance. Another issue is that newly mapped read-only data can not be detected, which leads to inconsistency at restarting because the corresponding content is missing. The solution for both cases are detailed in the next section.

2.3 Challenge: memory region changes

Virtual pages belong to a bigger logical unit called memory region or virtual memory area (VMA). Memory regions can be seen as an overlay structure of continuous virtual pages. At one time one virtual address belongs to exactly one memory region. Over time one virtual address may be reassigned to a new memory region. They are implicitly created from user space (e.g. *mmap* system call) and are created/managed by internal memory management mechanisms.
Memory region changes in connection with read-only and writable pages must be taken into account when managing the bookkeeping control structure. Two criterias must be fulfilled, proper assignment of pages to memory regions, which can be influenced by dynamic region creation/removal, and clean management of bookkeeping control structure entries, outdated entries must be deleted. If the later contains inappropriate content, a restart may fail or result in inconsistency.According to Linux memory region management [2] four cases of memory region changes can occur:
Rule 1 (region extension): if a new range of addresses is to be added to a process, the kernel first tries to enlarge an existing memory region. This requires virtual address holes or free address blocks in the process' address space and access rights of the exisiting region and the additional addresses being equal.
Rule 2 (region creation): if a new memory region is created and attached to the process' address space, the kernel tries to merge neighbouring regions, as long as they share the same access rights.
Rule 3 (region shortening): a certain address block can be removed from a region. If this address block resides at the beginning or end of a region, the region is shortened.
Rule 4 (region splitting): if the address block to be removed resides within an existing region, the region is splitted into two smaller regions.

The following examples demonstrate the need for an additional criteria than only checking the *write bit* in order to detect memory region changes, and thus page content changes.
Case 1: An application maps file A in a separate memory region 2. The application gets checkpointed, afterwards file A gets unmapped, memory region 2 vanishes. The application maps file B, accidently having same size and using the virtual address block of former region 2. A new memory

region 2 will be created.

In case file B has been mapped as read-only a new incremental checkpoint does not include the new

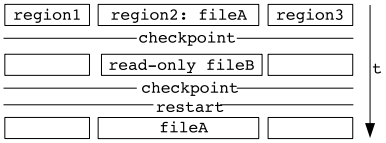

Figure 2: Region2 content with fileB has never been saved, restore old content of fileA

memory region 2 content, since no *write bit* has been set (see figure 2). At restart memory region 2 will be recreated containing file A (old memory region 2) content which is wrong.

In case file B has been mapped as writable and if it has been partially modified, a new incremental

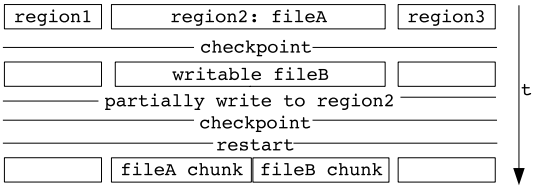

Figure 3: fileB was partially modified, restore mix of old and new content

checkpoint results in saving just the pages of region 2 with the *write bit* being set (see figure 3). After restart memory region 2 represents a mixture of file A and file B content which is wrong.

Case 2: this scenario is similar to the first one of case 1. However, a smaller file B is mapped, and thus a smaller memory region 2 will be created, resulting in a hole of the virtual addresses between new region 2 and region 3.

In case file B is mapped read-only, no content of region 2 will be saved because no *write bit* is

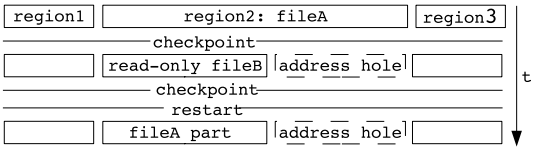

Figure 4: Region2 content with fileB has never been saved, restore part of fileA.

set (see figure 4). The reduction of virtual addresses of new memory region 2 is not reflected in the bookkeeping control structure. At restart parts of old memory region 2 will be recreated in the new address range of region 2. These bookkeeping control structure contains more entries than supposed to be which may result in wasted memory space.

In case file B has been mapped as writable, and if it has been partially modified, a mixture of old and new region 2 content will be reestablished at restart (see figure 5). Furthermore, the bookkeeping contains out-of-date data, since it does not reflect a memory region shrinkage.

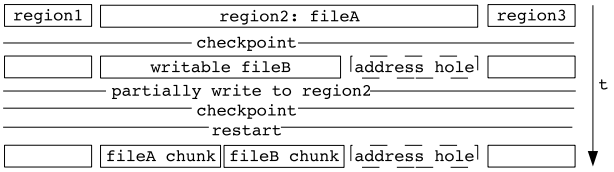

Figure 5: FileB was partially modified, restore mix of old and new content.

Case 3: three regions exist, a memory hole exists between region 2 and three. At checkpoint time the complete content of all regions is saved. Afterwards, region 2, which maps file A, gets unmapped, a new file B, which is bigger than the previous file A is mapped. New region 2 is placed between region 1 and 3, no memory hole between 1 and 2, as well as 2 and 3 exists.
In case file B is mapped read-only, no new region 2 related content is saved at an incremental check-

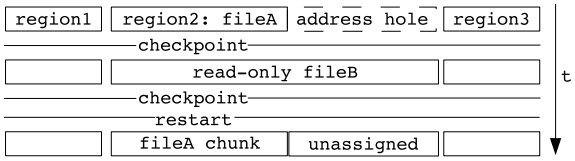

Figure 6: Region2 content with fileB has never been saved, restore fileA and unassigned space

point. Especially the additional virtual addresses of new region 2 opposite to old region 2, are not taken into account, since no *write bit* is set. At restart, region 2 contains file A (old region 2) content. Since the bookkeeping control structure is not aware of additional virtual addresses of new region 2 restart is likely to fail or causes inconsistency.
In case file B is mapped writable, and if it has been partially modified, a mixture of old and new

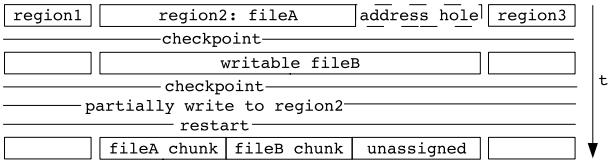

Figure 7: fileB was partially modified, restore mix of old and new content and unassigned space

content will be reestablished at restart for the address block covered by old region 2. Regarding the additional addresses of new region 2, the same effects are expected as explained shortly before.

Case 4: in the center of memory region 1 an address block is being write-protected having different access rights than the surrounding region 1 parts. Consequently, the Linux memory management enforces region 1 to be *splitted* into three parts. Region 1, 2 and 3, with region 2 containing the pages the *mprotect* call has been applied to.
In case the write-protected region 2 is not modified, or is partially modified, the same effects as described under Case 1 occur.

Case 5: region 1 contains an address block at the end or at the beginning which gets removed. The region gets *shortened*. In case region 2 is read-only, the appropriate bookkeeping control structure entries of the removed address block are not removed. Then, a new region gets created partially or fully covering the previously removed address block.
In case the new region is read-only, or has been partially modified, effects as detailed under Case 1 occur.

2.4 Solution: Memory region modification monitor

The special cases mentioned under 2.3 occur in combination of memory region modifications with read-only and writable content, e.g. such as shared segments, or anonymous memory region content or memory mapped files.
To tackle these issues we introduce an additional logical layer of modification detection - a memory region modification monitor. This monitor keeps track of memory region changes and thus complements the mere write-bit focussed approach of memory page modification detection. Based on monitor data the bookkeeping control structure can be kept in sync with the actual memory structure of a process at checkpoint time. It is sufficient to update the corresponding bookkeeping structure entries once, at checkpoint time, instead at each region modification event.
The monitor records region removals and additions. After each checkpoint, monitor data will be flushed. At checkpoint time the monitor entries are used to manage the bookkeeping control structure. Its entries are compared to control structures entries. In case a region has been removed, the start and end address of each monitor entry is used to delete appropriate bookkeeping control structure entries. This ensures control structure efficiency and consistency. In case a region has been added, relevant bookkeeping control structure entries are added and/or updated to reference appropriate checkpoint image contained pages. This allows whole new regions to be saved initially at the first incremental checkpoint after their creation.
Our monitor supports detection of *mmap* and *munmap* calls. Therefore, we insert a monitor notification function into the kernel functions *do_mmap* and *do_munmap*. Per *mmap* call the start and end address of an affected memory region are inserted into the memory region modification monitor. A detected *munmap* call results in deleting the appropriate entry of the monitor. For example, region creation detection, via the *mmap* call, results in initially saving the whole physical page content at the next checkpoint. Issues as listed under 2.3 can be avoided.
The memory region modification monitor has a similar structure as the bookkeeping structure. Its entries are organised in a radix tree providing fast access for entry removal, addition and retrieval. Each entry contains the memory regions start and end address of the covered virtual address range. The structure is shown in figure 8.

2.5 Incremental grid checkpointing

In order to realise one flavour of adaptive checkpointing, our incremental checkpointing enhanced kernel checkpointer has been integrated into the XtreemOS grid checkpointing architecture.
For the job checkpointer service to know when it is best to use a full or incremental checkpointing,

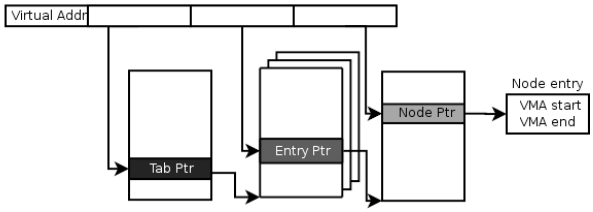

Figure 8: Memory region monitor structure

the number of modified pages of an application must be computed. Therefore, the job checkpointer service has been enabled to detect page modifications in a transparent manner for a given set of processes. Without modifying applications, the service can self-decide which checkpoint approach to be used by keeping efficiency.
Reporting page modifications from the kernel space to the user space domain has been achieved by setting up a new Linux Connector [6]. The service registers at a so-called *memory event connector (MEC)* at kernel-level to be informed about *do_page_fault* calls triggered by selected processes. The service receives MEC messages at user space and performs accounting on page faults on a per process base. In case the collected data exceed a certain threshold, the job checkpointing service enforces full checkpointing. Otherwise, incremental checkpointing is used.

2.6 Measurements

The testbed consists of two desktop nodes with Intel Core 2 Duo E6850 processors (3 GHz) and 2048 MB DDR2-RAM which are connected by a gigabit network. A master node runs a tftpboot and a NFS server providing a LinuxSSI image and a Linux environment including the directory for checkpoint image storage to a client node. Our test application allocates 1 MB of RAM and writes integer values to random locations in 1 millisecond intervals. A checkpoint is triggered by using the "-i" flag of the checkpoint command.
Figure 9 shows the resulting image size of full and incremental checkpoints. Figure 10 shows the

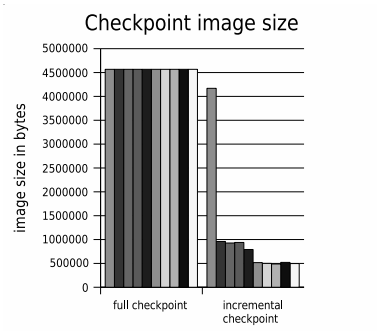

Figure 9: Image size of full and incremental checkpointing

checkpoint duration of full and incremental checkpointing if checkpoints are issued successively in one second intervals.

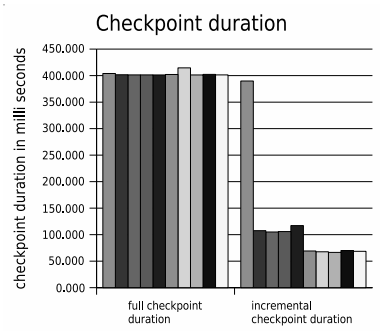

Figure 10: Full and incremental checkpointing duration

Both data sets indicate incremental checkpointing takes fourth time faster especially after the initial checkpoint. Figure 9 shows the resulting image size of full and incremental checkpoints. It appears that one incremental checkpoint image file is fourth time smaller than a full checkpoint image, especially after the initial checkpoint. Efficiency of incremental checkpointing relies on the write behaviour of an application. Since an additional control structure and a region monitor need to be maintained, incremental checkpointing may become inappropriate, especially when more pages have been changed per checkpoint interval. It is the task of the grid service to figure out the best-suited strategy.
Furthermore, restarting from an incremental checkpoint may result in accessing more than one image file rather than to just one file with full checkpointing. Increased I/O overhead, caused by reading from multiple files, is likely decrease restart performance.

3 Callbacks

3.1 Overview

Kernel checkpointers in general lack application semantic knowledge. Hence they usually have no information about what needs to be saved and restored. Callbacks give the user the possibility to determinate what resources need to be checkpointed and what to be restored. They are registered by the user and executed before taking a checkpoint. Due to the possibility of enabling the user to decide which resources need to be checkpointed a performance increase can be achieved. There are currently several callback enabled kernel checkpointers available. One of them is BLCR which is used by the XtreemOS Grid-Checkpointing service. LinuxSSI checkpointer which is also used by the Grid-Checkpointing service didn't have any callback support until now. We have designed and integrated a callback solution for LinuxSSI. Furthermore the implementation was reviewed by the Kerrighed core developers and merged into the official Kerrighed version. We will present the design and implementation in the following chapters.

3.2 Design

Our callback library is installed during the LinuxSSI setup and linked against each application which would like to support callbacks. We provide two possible approaches to register callbacks.
The first one presupposes the master process of each application to call the init routine of the callback library and register the callbacks. In that case the master process of each application is in charge of executing the callbacks only.
The second approach allows each process to register his own callbacks. Hence each process has to call the init and callback registration functions on its own. This approach requires the master process of the application to coordinate the callback execution of its child processes.
There are two types of callbacks which can be registered by the application. Signal handler and thread context callbacks. It is up to the programmer to choose which callback registration method to use. Anyhow most of the syscalls available are signal handler unsafe. Thus it is recommended to use thread context callbacks.
In order to trigger the callback execution a callback detection must be done first. We will describe how we detect either some application has callbacks registered or not in the chapter 3.3. A message has to be sent to the application after the detection. LinuxSSI checkpointer makes use of two utilities (checkpoint and restart) to trigger a checkpoint. Thus the callback library should be able to handle incoming callback execution requests from this tools and be able to send a reply upon a successfull or failed execution.
The library should be able to support registration of different callbacks on checkpoint and restart. Hence two messages can be send to the callback library. The first one initiates the callback execution on checkpoint and the second one on restart. After a checkpoint the applications needs to run as it used to run before taking the checkpoint. Which means that some of the resources have to be closed before taking the checkpoint and reopened appropriate after the checkpoint. We introduce a third message which is send after a successful checkpoint to execute the continue callbacks. These callbacks are in charge of reopening resources closed before taking a checkpoint.
The figures 11 and 12 illustrate the communication workflow between the callback library and the checkpoint/restart utility.

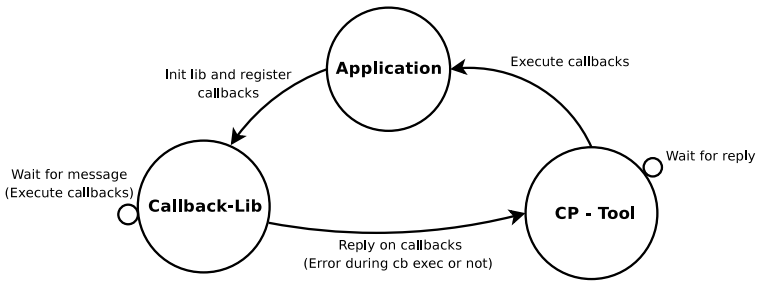

Figure 11: Callback workflow on checkpoint

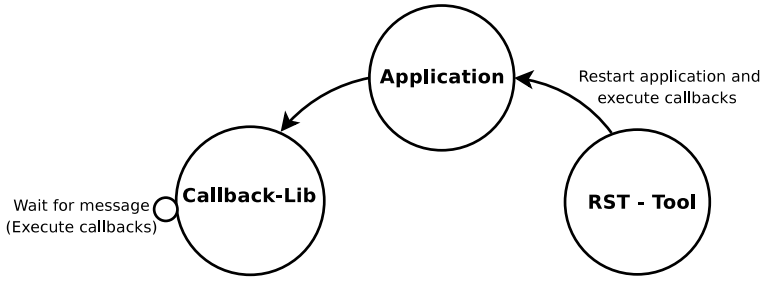

Figure 12: Callback workflow on restart

3.3 Implementation

Our implementation consists of mainly two files (libs/libkrgcb/libkrgcb.c and libs/include/libkrgcb.h). Furthermore we have made minor modifications to the checkpoint/restart commands and kernel.
As already mentioned in the chapter 3.2 there are several decisions which have been taken in order to realize the callback library.
The first decision regards the communication between the callback library and checkpoint/restart commands. Since there is currently no way to checkpoint/restart pipes, sockets and files we have decided to use signals in order to trigger the callback execution on checkpoint and restart. Three unassigned real time signals (37, 38 and 39) have been chosen in order to signalise the execution of checkpoint, continue and restart callbacks.
All callbacks registered for signal handler context will be executed outside signal handlers themselves. In case of a thread context callback registration a worker thread will be spawn on the initial thread context callback registration. This thread will be unlocked out of the signal handler and trigger the callbacks execution.
Moreover we employ a temporary message queue inside the checkpoint utility which is used to receive the status of the callback execution from the callback library for each application. Each message queue is identified by the checkpoint image directory and application id which is closed shortly before taking a checkpoint.
The second decision concerns the callback detection. It is necessary to detect either some application has callback support enabled or not before sending a signal to execute callbacks. Without such a propper detection checkpoint/restart utilities would be able to crash any application. Currently there are only few signals which are ignored by default in Linux (PWR, WINCH, CHLD and URG). Sending any other signal will lead to exit. We have studied several methods of callback detection including ones based on touching files on disk. Wherever they turned out to be insecure. A possible attacker could create a fake file and make the application crash. In order to realize a reliable callback detection a new kernel level bitmask has been integrated for each application in cooperation with Kerrighed core developers. This bitmask is set during the initialisation of the callback library and inspected by the checkpoint and restart commands before sending a signal.
The table 3.3 shows the current internal functions used by the callback library.

Currently the callback library provides the following external functions to be used by the application and checkpoint/restart commands:

In order to test the callback library we have developed a callback test application (tests/app-

Function	Description
int cb_count_read(hook, count)	Read callback count
int cb_count_write(hook, value, count)	Update callback count
int cr_detect_callbacks(pid, from_appid)	Detect callbacks
void free_info_memory()	Free info structure memory
void handle_cb_sig(signum)	Signal handler
int register_callback(hook, func, *arg)	Main function to register a callback
int run_callbacks(hook)	Run callbacks
int send_message(msg)	Send a message to checkpoint/restart utilities

Table 1: Internal functions - callback library

Function	Description
int cr_callback_init(void)	Init callback library
void cr_callback_exit(void)	Exit callback library
int cr_register_chkpt_callback(func, *arg);	Register checkpoint callbacks (signal)
int cr_register_restart_callback(func, *arg);	Register restart callbacks (signal)
int cr_register_continue_callback(func, *arg);	Register continue callbacks (signal)
int cr_register_chkpt_thread_callback(func, *arg);	Register checkpoint callbacks (thread)
int cr_register_restart_thread_callback(func, *arg);	Register restart callbacks (thread)
int cr_register_continue_thread_callback(func, *arg);	Register continue callbacks (thread)
int cr_execute_chkpt_callbacks(pid, from_appid)	Execute checkpoint callbacks
int cr_execute_restart_callbacks(pid)	Execute restart callbacks
int cr_execute_continue_callbacks(pid, from_appid)	Execute continue callbacks

Table 2: External functions - callback library

s/cb_test.c) using sockets. It has been included into the set of KTP regressions tests and can be used to test the callback registration and execution.

3.4 User Manual

The callback library is installed by default during the LinuxSSI setup. It can be found inside the "/usr/local/lib" directory. Applications can link (-libkrgcb) it to support callbacks. In order to use the library the programmer first has to include the "libkrgcb.h"-Header file:

```
#include <libkrgcb.h>
```

Afterwards the initialisation routine must be called out of the master process (see 3.2) using the function: "cr_callback_init();"

Callbacks can be registered by using the functions described above. The following example provides a simple signal handler context callback registration of a checkpoint, restart and continue call-

back:

```
cr_register_chkpt_callback(&callback1, NULL);

cr_register_restart_callback(&callback2, NULL);

cr_register_continue_callback(&callback2, NULL);
```

The callback library must be exit appropriately by using the “cr_callback_exit();“-function after a successfull process execution.

4 Conclusion

The LinuxSSI-1.0-rc1 release has been produced in May 2009 and included in the latest XtreemOS 2.0 CD. The XtreemOS cluster flavour is available on the latest XtreemOS installation CD. XtreemOS-G services (AEM daemon, XtreemFS client) are able to run on top of this release which also supports the system level VO support mechanisms. Since November 2008, LinuxSSI kernel checkpointer has been optimized through incremental checkpointing. Furthermore callback support has been introduced in LinuxSSI and integrated in the official Kerrighed version. The LinuxSSI kernel checkpointer is fully integrated in the XtreemGCP Grid checkpointing service and is able to checkpoint multi-process applications with multiple threads.

In our future work directions, we plan to extend the LinuxSSI kernel checkpointer to use the support provided by kDFS for file checkpointing. Checkpointing and restart of SYSV IPC shared segments will be reviewed for integrating it into the previously introduced framework for saving and restoring shared structures. Moreover, the needed support for checkpointing applications composed of processes communicating through Inet sockets will be implemented. Then, it will be possible to checkpoint/restart parallel applications such as MPI applications.

In order to support the migration of IP services executed on top of LinuxSSI, we plan to add dedicated mechanisms to the regular Linux network stack. This improvement will allow moving an IP service from one node to another one within a LinuxSSI cluster, transparently to the service clients running on remote nodes. External IP services, including those XtreemOS-G service dameons running on all Grid resource nodes, could take advantage of this feature.

Moreover, we plan to define and experiment advanced load balancing strategies using the customizable scheduler framework.

Finally, we will continue our efforts to push LinuxSSI patches in Linux mainstream development. According to the strategy defined in the early stages of the XtreemOS project, we need to first push the KDDM Kerrighed foundation layer into Linux kernel. Towards this goal, we plan to provide a preliminary support to node addition/removal in the KDDM layer.

During the last twelve months of the project, we will continuously provide support to the community and in particular to partners involved in WP4.2 who are in charge of validating XtreemOS and those involved in WP4.1 who are responsible for the packaging of the XtreemOS software. Extensive testing activities will take place to further increase the stability of XtreemOS cluster flavour.

References

[1] *Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers*, Washington, DC, USA, 2005. IEEE Computer Society.

[2] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly, 2006.

[3] XtreemOS consortium. Design and implementation of the first advanced version of LinuxSSI. Deliverable D2.2.8, November 2008.

[4] R. M. Fujimoto, J.-J. Tsai, and G. Gopalakrishnan. Design and performance of special purpose hardware for time warp. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 401–409, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[5] F. Gomes and A. F. Bosco. *Optimizing incremental state-saving and restoration*. PhD thesis, University of Calgary, Calgary, Alta., Canada, Canada, 1996. A.-U. Brian.

[6] M. Helsey. Process event connector. 2005.

[7] J. Mehnert-Spahn, E. Feller, and M. Schoettner. Incremental checkpointing for grids. Linux Symposium, March 2009.

[8] Darrin West and Kiran Panesar. Automatic incremental state saving. In *Proc. 10th Workshop on Parallel and Distributed Simulation (PADS 96*, pages 78–85, 1996.