Project no. IST-033576

XtreemOS

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

Design and implementation of basic checkpoint/restart mechanisms in LinuxSSI
D2.2.3

Due date of deliverable: November 30^{th}, 2007
Actual submission date: November 30^{th}, 2007

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

Type: Deliverable
WP number: WP2.2
Task number: T2.2.3

Responsible institution: UDUS
Editor & and editor's address: John Mehnert-Spahn
University of Duesseldorf
Universitaetsstrasse 1
40225 Duesseldorf
Germany

Version 1.0 / Last edited by John Mehnert-Spahn / December 4^{th}, 2007

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	13.9.07	John Mehnert-Spahn	UDUS	initial version of the document
0.2	25.10.07	Michael Schoettner	UDUS	extensions and some polishing
0.31	28.11.07	John Mehnert-Spahn	UDUS	applying reviewer comments
0.32	3.12.07	Michael Schoettner	UDUS	formatting, typos
1.0	4.12.07	John Mehnert-Spahn	UDUS	final checks

Reviewers:
Erich Focht (NEC), Toni Cortes (BSC)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.2.3	Design and implementation of basic application checkpoint/restart mechanisms	INRIA, UDUS*

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

Abstract

This deliverable aims at presenting the prototype of the LinuxSSI-XOS kernel checkpointer as it is in month 18 of the project. Functionality here is what is currently working, although more features are half way ready. After a short introduction we briefly describe the overall XtreemOS (XOS) checkpointing architecture and how the LinuxSSI-XOS kernel checkpointer is integrated.

In the following section we first describe relevant concepts of LinuxSSI-XOS (Kerrighed): Kerrighed Distribted Data Management (KDDM) and ghosts. Subsequently, we present the prototype implementation where several import and export functions related to the ghost mechanisms had to be extended to include additional process/kernel state information (e.g. system KDDM set objects) in file ghosts. Furthermore, some system KDDM set objects needed to be created and initialized manually during the restart phase of a process.
The current implementation fits smoothly in the existing KDDM and ghost mechanisms and the LinuxSSI-XOS kernel checkpointer is able to checkpoint and restart sequential processes and also supports SYSV IPC shared memory segments.

Finally, this deliverable also includes a description for users and a reference to installation sources for LinuxSSI-XOS (Kerrighed).

Contents

Chapter 1

Introduction

Reliability of applications can be improved by periodically saving checkpoints in stable storage. In case of an error a backward error recovery can restart the application from the last checkpoint, avoiding a fallback to the initial state. Saving a checkpoint requires not only to save the application data but also relevant kernel contexts, e.g. task structs, file descriptors, socket states etc. The latter is not a trivial task but kernel state handling is mandatory to implement checkpointing and restart in a transparent fashion.

When checkpointing a distributed and parallel application, all processes running the application need to be checkpointed in a way that the set of checkpoints form a consistent snaphot, e.g. messages in transit need to be recorded, too. There are mainly two classes of checkpointing approaches: coordinated and independent strategies. The first one stops all involved nodes. After all nodes have stopped the application and kernel state of each node are recorded and written to disk. The latter one avoids the coordination overhead and each involved node saves checkpoints independently to disk. Furthermore in this case a consistent snapshot has to be detected during the recovery phase which may require analyzing a lot of checkpoint combinations and may even fail thus requiring to restart the application from the initial state (known as the domino effect). This worst case can be avoided by logging received messages on each node to be able to recover single nodes in case of a failure. Obviously, this is the well-known trade-off between costs during fault-free operation and recovery phase. Which checkpointing strategy to choose depends on the error frequency and as well as the application needs. The workpackage WP2.2 has decided to implement a coordinated checkpointing and recovery strategy for the first prototypes, because implementation and debugging is less difficult.

In the XtreemOS project checkpointing and restart must support distributed and parallel applications that run on several grid node types (a single PC or a cluster). Checkpointing of single PCs is done with an extended version of BCLR (WP2.1; more information can be found in the deliverable D2.1.3[**?**]) part of Linux-XOS.

Clusters are managed by the LinuxSSI-XOS system that is an extended Kerrighed version (WP2.2; more information can be found in the deliverable D2.2.1[**?**]). LinuxSSI uses the Single System Image (SSI) approach at kernel level making the cluster appear as one single powerful grid node. These two different kernel checkpoint/restart services will be controlled, e.g. in a coordinated fashion, by the grid checkpointer that will be developed within WP3.3 (ongoing development).

In this deliverable we describe the basic checkpointing/restart mechanisms that have been implemented in LinuxSSI-XOS. Because of numerous kernel modifications and the SSI concept, a tailor-made kernel checkpointer became necessary to save and restore all relevant LinuxSSI-XOS kernel structures.

Beside handling kernel and user process states, using the built-in ghost mechanism, the current implementation is also able to checkpoint a process hierarchy where the processes are spread over the cluster. This is a cluster specific optimization of the LinuxSSI-XOS kernel checkpointer that benefits from the SSI properties.

The deliverable is organized as follows. Subsequently, we briefly describe the overall XtreemOS (XOS) checkpointing architecture and how the LinuxSSI-XOS kernel checkpointer is integrated. In section three we first present relevant concepts of LinuxSSI-XOS (Kerrighed): Kerrighed Distribted Data Management (KDDM) and ghosts. Then we present the prototype implementation and its status. Section four is devoted to user commands and installation notes, followed by the last section containing the conclusions and an outlook on future work.

Chapter 2

Overall XOS checkpointing architecture

2.1 General XOS Overview

XtreemOS is composed of two parts: XtreemOS foundation (XtreemOS-F) and XtreemOS grid services (XtreemOS-G), see figure 2.1.
LinuxOS-F comes in three flavours: Linux-XOS, LinusSSI-XOS, and Linux-XOS

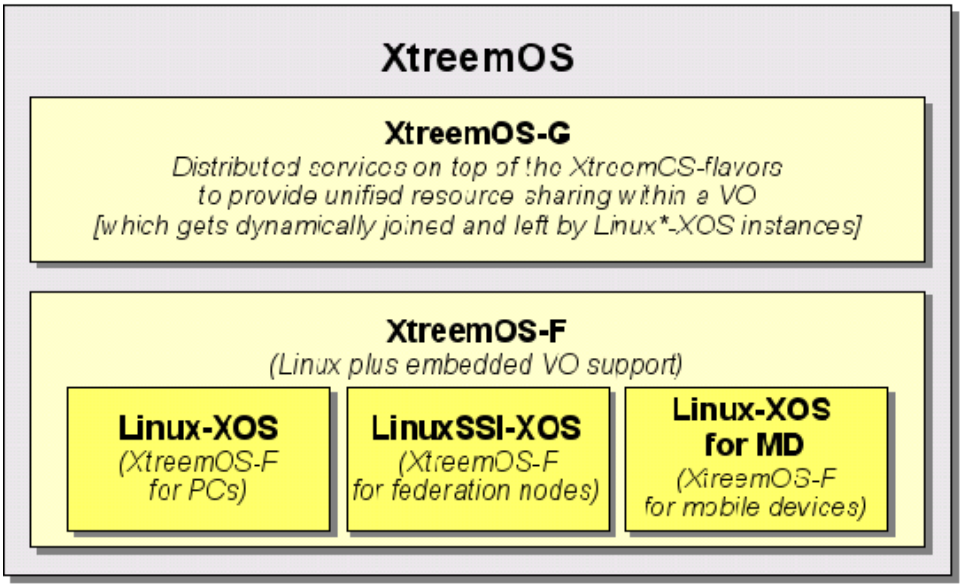

Figure 2.1: XtreemOS architecture

for mobile devices (MD). The Linux-XOS modifies the existing Linux operating system (OS) to support virtual organizations. LinuxSSI-XOS is a cluster operating system based on Kerrighed and Linux. These two XOS versions will be modified,

extended and configured towards providing support for checkpointing of grid applications. Currently, checkpointing and restart is not planned for Linux-XOS for MD.

The XtreemOS checkpoint mechanism will be responsible for checkpointing/restarting a grid application. A grid application may consist of one or more jobs. A job can consist of one or many job units. A job unit can consist of one or more processes each executing one or more threads. On each grid node there may be one job unit, only.

Checkpointing grid applications will be addressed by a hierarchical grid checkpointing architecture consisting of:

- kernel checkpointers - responsible for node states (user and kernel level) on each computing resource (handles processes, not job units),
- a job unit checkpointer and
- a job checkpointer taking care of a job/jobs that constitute a grid application.

Two different kernel checkpointers are developed within XtreemOS, one within WP2.1 for single PCs (Linux-XOS) and another one within WP2.2 for clusters (LinuxSSI-XOS). Both are implemented within XtreemOS-F. Linux-XOS extends BLCR, the most advanced open source implementation of checkpoint/restart, for Linux. LinuxSSI-XOS uses a custom checkpoint mechanism relying on the Single System Image (SSI) properties provided by the Kerrighed system.

The job unit checkpointer implements periodic checkpointing, staged checkpoints and garbage collection. It registers checkpointing strategies (e.g. coordinated checkpointing or independent checkpointing as explained in the introduction). It will take care about needed resources and will call the kernel checkpointer(s).
The job checkpointer is a distributed service that supervises checkpoints for a grid application by applying the checkpoint strategy to all job units of a job. It therefore registers job units with the checkpoint service on the grid nodes that run the job units. Furthermore, it provides resources to store checkpoints and detects node failures and reacts on them if necessary.

2.2 Application Execution Management (AEM)

The AEM is a central component of XtreemOS-G where important parts of the hierarchical grid checkpointing architecture will be implemented. AEM is implemented within WP3.3 and manages jobs by realising per-job and per-resource services (distributed and non-distributed) that will be run on the underlying grid nodes (Linux-XOS and LinuxSSI-XOS).

AEM is devided into client and system components. The client provides user commands to initiate the execution of a job. Therefore the client calls the XtreemOS system side for job execution and managment as well as resource management. The system side implements the functionality offered to the client. The XOSD (XtreemOS Daemon, see deliverable D3.3.2 [**?**]) exists once per node executing several threads, including:

- job manager
- resource manager
- execution manager .

Global services as the JobDirectory and a resource selection mechanism support the job execution.

job manager
A job is described by a JSDL (Java Service Description Language) file that specifies a unique program file and initial requirements (checkpointing options included). For each job there is one responsible job manager. A job manager can manage one or more jobs. It receives and processes requests for various job services (jMontoring, jScheduling, ...). It knows at all times all the resources on which components of the job are running as well as the JSDL content. A job manager serves as interface to the job. Having a job ID the address of the belonging job manager can be retrieved from the Job Directory. The job can then be accessed. Requests to jobs not managed by a local job manager will be redirected to the node with the appropriate job manager.

resource manager
The resource manager handles all requests concerning resources.

execution manager
The execution manager is a distributed service that implements methods for managing the execution of jobs (or job units). Therefore it uses local services as the system calls *fork* and *exec* to create processes and manages Linux signals. In contrast to the Java Virtual Machine (JVM) the execution manager provides a comprehensive resource usage control that allows for example to export process IDs.

AEM will track the process hierarchy by monitoring fork system calls. The execution manager knows at all times the exact processes using a resource and belonging to one job. For more information on AEM, see the deliverables D3.3.1 [**?**], D3.3.2 [**?**] and D3.3.3/D3.3.4 [**?**].

2.3 AEM and Grid Checkpointing

The hierarchical grid checkpointing architecture will be integrated into XtreemOS as an extension to AEM. In figure 2.2 the planned AEM extensions (blue boxes)

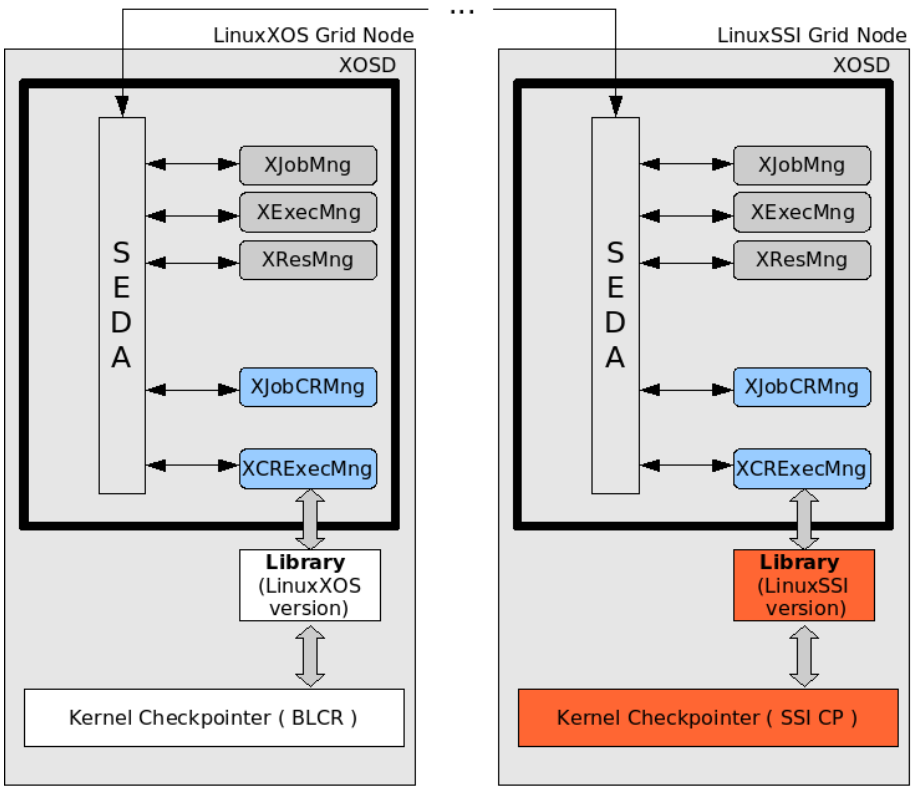

Figure 2.2: AEM and hierarchical checkpointing architecture

are presented as well as the relation of these new AEM services to the different kernel checkpointers.

The new AEM service *XJobCRMng* (JobCheckpointingRecoveryManager) corresponds to the grid checkpointer. The latter requires a strong interaction with the *XJobMng* which knows always the exact resources used by a job unit. This allows the XJobCRMng to coordinate checkpointing for a job. It will take into account checkpointing constraints/strategies that are provided in the JSDL file along with job requirements. Furthermore, it will be responsible for detecting node failures and it will indirectly acquire resources needed for saving checkpoints within XtreemFS.

The new AEM service *XCRExecMng* (CheckpointingRecoveryExecutionManager)

corresponds to the job unit checkpointer. It needs a strong interaction with the *XExecMng*. To enable the grid checkpointer to apply the checkpointing strategy to all grid nodes where job units of a grid application are running - a job unit checkpointer API is necessary. Such an API has been implemented for a job unit checkpointer prototype, see [**?**] for more information. In the context of coordinated checkpointing the job unit checkpointer will also need to send signals (STOP, CONT) to all processes belonging to the job unit on the grid node by calling the underlying kernel checkpointer.

The XCRExecMng uses an API to transparently call the underlying kernel checkpointer (LinuxSSI-XOS CP or Linux-XOS CP) via a library. The library comes in two versions (one for LinuxSSI-XOS and Linux-XOS). It implements the API according to the calling semantics of the appropriate kernel checkpointer. LinuxSSI-XOS and Linux-XOS will be configured with the appropriate library before system start.

Both have to be augmented in order to support XtreemOS specific features as for example Virtual Organization Management (VOM). (BLCR restart with new security context - VO specific information, see [**?**]).

In the following section the design and implementation of the LinuxSSI-XOS kernel checkpointer that has been implemented within the first 18 months will be presented.

Chapter 3

Basic Prototype: Checkpointing and Restart in LinuxSSI

The base for the LinuxSSI kernel checkpointer is the Kerrighed Distributed Data Management (KDDM) and the ghost mechanism. Both together provide a process migration facility that provides core functions for our checkpointing and restart implementation. Subsequently, we give a short overview of the relevant KDDM and ghost concepts that have been implemented by KerLabs for Kerrighed.

3.1 KDDM and Ghost

3.1.1 KDDM Introduction

KDDM is the central abstraction for distributed data management and sharing within a LinuxSSI cluster. A KDDM set is an instance of the KDDM mechanism. It is used to store objects and to share them between cluster nodes. KDDM set objects can correspond to physical process pages or blocks in the file cache but also are used to store various kernel data structures. KDDM set objects are also used by applications to transparently share a variety of resources (process address space, memory segments, data streams and files). KDDM based data sharing is realised by the concept of linkers. Linkers enable the transparent integration of KDDM sets into the Linux kernel, see figure 3.1.

Interface linkers connect system services such as the Virtual Memory or the Virtual File System with KDDM sets on one side.

I/O linkers combine KDDM sets with device managers on the other side. Each time a KDDM set is being created it is attributed a corresponding I/O linker that fits the type of data to be shared. An I/O linker supporting shared files differs from an I/O linker supporting a shared process address space. Linkers enable page faults to be resolved in a different manner as Linux does. For example a page - mapped

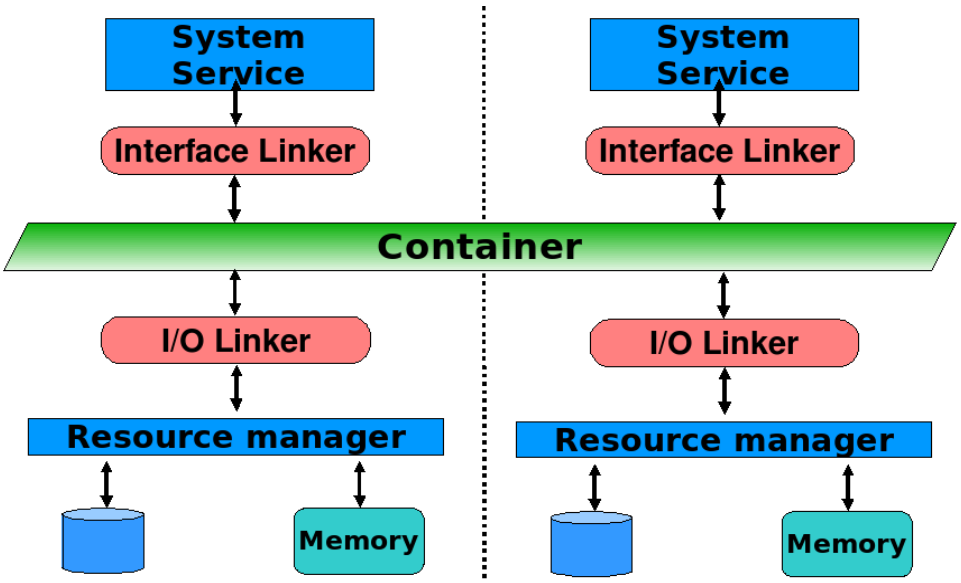

Figure 3.1: KDDM linkers

by a shared memory region and not present in local memory - can be retrieved from a remote node.

3.1.2 KDDM - Consistency Management

Object sharing via KDDM sets results in multiple copies of the same object in the cluster. Consistency is implemented by a MESI-like (MESI = Modified, Exclusive, Shared, Invalid) coherence protocol with write-invalidate semantic. The protocol allows multiple readers but only one writer. In case of a write request for a KDDM set object all existing copies of this object will be invalidated in the cluster before. Read requests do not trigger object invalidations but create new replicas.

The so called *first touch* inserts a resource (e.g. memory page) into the memory section of a KDDM set. Therefore a new KDDM set object is allocated and initialized with meta data and the resource data itself.

At any time there is exactly one node that is the owner of a KDDM set object. The owner manages all replicas of a KDDM set object and is mainly responsible for invalidating potential replicas before granting write access. The node who performs a first touch on a resource, automatically becomes the owner of the resulting KDDM set object. Ownership migrates to the node that performed the last write operation on this object. Every KDDM set object request results in communication with its owner. In order to find the owner of a KDDM set object, a node has to follow a so called *chain of probable owners*. If a node has already used an object,

then the node has a reference to another node, a probable owner. This node will be requested to deliver the object but this node may be no longer the owner but it knows to which node it has granted the ownership. Of course this node might also have lost ownership thus the request is forwarded again and so on.

If a node has never accessed an object it does not know a probable owner. In this case the first owner (the node that performed the first touch) is contacted which has a reference to the probable owner.

3.1.3 KDDM - Addressing a KDDM set object

In the following subsection we shortly describe how a KDDM set object is addressed. A KDDM set is identified by two levels of naming. Firstly, the namespace allows the developer to define which KDDM sets should be grouped by using name spaces. It can contain:

- all system KDDM sets related to a given Linux name space or,
- all KDDM sets linked to file address spaces, e.g. for a storage device or,
- a default name space is created during Kerrighed initialization. This name space hosts all the system KDDM set which are not linked to a Linux name space. Currently only the default name space is used.

Secondly, each KDDM set is identified by KDDM set index. Furthermore, a KDDM set object, within a given KDDM set, is addressed by a 64 bit identifier having a resource bound meaning. For example for retrieving a read copy of a PID system KDDM set object one has to use following function call:
kddm_get_object (struct kddm_ns ns, kddm_set_id_t set_id, objid_t objid);
with ns being the default namespace, set_id being 55 (PID system KDDM set) and objid being the process identification of the wanted pid KDDM set object.

To access KDDM set objects stored on remote nodes it is sufficient to know the corresponding KDDM set id, the KDDM set namespace and the object identifier. For efficiency reasons KDDM sets for some resources are created on demand only (lazy allocation of KDDM sets).

3.1.4 KDDM - User and System KDDM sets

There are basically two kinds of KDDM sets: one to host user application data *user KDDM sets* and another one to manage system resources *system KDDM sets* that are shared cluster wide.

There can be one or more user KDDM sets per user application e.g. exactly one for sharing a process address space (memory user KDDM set) and one shared memory user KDDM set per SYSV IPC shared segment. Resources are accessed via

system services such as the virtual memory or the Virtual File System (VFS). Memory user KDDM objects equal process memory pages, they are identified by their adress in the virtual memory address space. User KDDM sets are allocated lazily per node and are addressed by a dynamically attributed KDDM set identifier and a namespace.

Furthermore, there is one separate system KDDM set per system resource, e.g. process identifiers, process descriptors, shared memory keys and identifiers, signals, signal handlers and so on. System KDDM set data are the key to support the execution of a distributed application in LinuxSSI from the kernel point of view. Relevant system KDDM objects are:

- The pid KDDM object serves for PID allocation and recycling. There is one pid KDDM set object per process in the pid system KDDM.
- The task KDDM set object shares some fields of the Linux task_struct structure.
- The children_kddm_object KDDM set object supports reparenting of children between (distributed) threads of a thread group.
- The signal_struct KDDM set object supports sharing of signals in connection with distributed threads.
- The sighand_struct KDDM set object supports the sharing of signal handlers in connection with distributed threads.
- The app_struct KDDM set object keeps track of those processes belonging to one application.

System KDDM sets are being set up per node during system initialisation (not lazily). Each system KDDM set is identified by a fixed ID and a namespace. System KDDM set objects are not accessed by the user application via system services but directly by the system. Each system KDDM set object is identified by an identifier having a resource specific meaning referred to later on.

3.1.5 Ghost

Ghost is the base mechanism for process migration. An image of a process to be migrated is put into a *network ghost* and sent via the network. At the destination node another network ghost is set up and initialized with the transmitted data. This ghost is used to recreate the process from the source machine.

Furthermore, the ghost mechanism can be used to support checkpointing and restart. Instead of sending the process image via the network it is put into a *file ghost* and

saved on disk (the image is persistent). This image is read during restart by another file ghost that serves as source for process recreation. However, the information stored within a file ghost was not sufficient (at the beginning of the project) for process recovery, so more information needed to be included.

3.2 Prototype Implementation

3.2.1 Synergies between Migration and Checkpointing

Synergies between checkpoint and migration are obvious as both need to retrieve an image of a process. In LinuxSSI-XOS a sequence of so called *export functions* is issued by the kernel migration facility to extract data that represent a user level application at kernel level. These export functions can be used for checkpointing as well. However for checkpointing to work extensions had to be made to various export functions. As described before - KDDM sets host application and system data. These LinuxSSI-XOS kernel structures *need to be saved persistently* during the checkpointing operation and *must be restored* carefully during the restart operation of an application. On top of the existing checkpoint code (developed by KerLabs), that was non-functional at project start (due to Linux kernel 2.6 porting tasks), saving various KDDM set data has been the major contribution of WP2.2, that lead to a working checkpointing functionality.

3.2.2 Exporting data for migration

Inherent to migration is to immediatly recreate an application on the destination node. The network ghost abstraction is used to insert the system extracted data into a ghost and to send it via the network. The ghost image is not saved persistently on the source and destination node. Cluster-wide shared process data that resides in user and system KDDM sets will not be saved persistently. Ghost data and KDDM data that will not be saved at all since an interruption of the migration procedure (from extraction till deployment) is not expected to happen. As a consequence a node failure will result into loosing all KDDM set data needed to have the application running at an advanced and not initial state.

The question is how can the destination node access the appropriate KDDM set data (process address space, signal mask, pending signals, signal handlers and others) of the source node without sending them using a network ghost. In case of migration, only the user memory KDDM set is set up on the source node and initialised with the process address space. Only the *user memory KDDM set identifier is included in the network ghost instead of the memory content itself.* With the help of the identifier the appropriate local user memory KDDM set can be setup on the destination node and objects can be retrieved from the cluster by lazily allocating them in the local KDDM set. Migrating system KDDM set objects works the same.

3.2.3 Exporting data for checkpointing

Checkpointing requires to save all relevant system data that is needed for an application restart. Furthermore a checkpoint needs to be saved persistently to survive severe failures, e.g. reboot. Unlike checkpointing, migration does not depend on persistent data - such failures are not expected to happen. Instead of using a *network ghost* (as with migration) - the extracted data (non-KDDM data e.g. register content, virtual memory structure and stack as well as KDDM data) is inserted into a *file ghost* and therewith saved on disk persistently. The most challenging task was to recognise what application related user and system KDDM set content needed to be saved in addition into the checkpoint for a successful restart. Applications - especially distributed ones - heavily use KDDM set data in order to run in the cluster. Therefore the *export sequence had to be modified at multiple locations* within the kernel code. Extensions to some export functions were necessary to save the following system KDDM set objects:

- The signal_struct KDDM set object and
- the sighand_struct KDDM set object.

The signal_struct and sighand_struct KDDM set object had to be stored within a checkpoint because it is not sufficient to just save the KDDM set identifier like for process migration. In case of a severe failure all KDDM set content might be lost and cannot be addressed with the identifier. Nevertheless, it is not necessary to save the data of all system KDDM set objects since some of them can be recreated and *reinitialized at process restart with runtime information*.
Furthermore a new export function (export_process_pages) has been implemented within WP2.2 to save all memory user KDDM set objects that constitute the process adress space.

Checkpointing and restart of **SYSV IPC shared memory segments** has been implemented by WP2.2 during the first 18 months, too. Both cases - local processes and processes, distributed over several nodes, that share a segment - are supported by LINUXSSI. During checkpointing operation the shared memory key and the size of the shared memory file and of course the segment content itself needs to be saved. The shared memory content resides in the shm user KDDM and had to be exported into the file ghost explicitly (extension of function export_one_page). Since it is unknown at checkpoint time which of the processes, sharing the segment, will be restarted first, the whole segment content is attached to the checkpoint file of each process.
Furthermore, there is no need to save the contents of following SYSV-IPC-related system KDDM set objects: *shmkey*, *shmid*, and *ipcmap*. All these system KDDM set objects will be recreated during the restart phase by using SYSV IPC system calls, like done by the application before the failure. But the shared memory key needs to be saved as it is required during restart to create the shmkey system

KDDM set object *whose object id equals the shared memory key. The shmkey system KDDM object is initialized with the index of the shmid system KDDM set object in the shmid system KDDM set.* The shmid system KDDM object encapsulates relevant IPC kernel structures. According to this - relevant kernel structures for a segment can be accessed merely by having the shared memory key.
It is also necessary to save additional data for certain applications using SYSV IPC shared memory segments (e.g. for parent-child-processes) that ensures only one shared memory segment and one shared memory user KDDM set is being setup at restart.

3.2.4 Synergies between migration and checkpointing-based restart

There are also synergies between restart and migration in terms of rebuilding a process from an image. Analogous to the sequence of *export functions* there is a sequence of so called *import functions* that extract data from a ghost that internally represents an application and reset it in the system. Import functions are used by the migration and the restart functionality.

On top of these import functions (developed by KerLabs) the major contribution of WP2.2 was to recreate and reinitialize certain KDDM set objects that lead to a working restart functionality.

3.2.5 Importing data for restart

In contrast to migration the whole process address space is stored in the file ghost image. Using the recovered page table entries - the mapped physical addresses are populated with the process address space content. If there was a memory KDDM set before the checkpoint - no memory KDDM set needs to be set up right away at restart (lazy KDDM allocation). This does not apply to SYSV IPC shared memory and the shm user KDDM set (will be discussed later).

Two more steps have been mandatory referring to KDDM set data restoration. In the *first recovery stage* it was necessary to recreate certain system KDDM set objects. For a process to be visible in the cluster different import functions had to be modified. It was not easy to identify the appropriate locations in the kernel code where modifications had to be applied to. In contrast to migration - the following system KDDM set objects had to be created explicitly (via first touch) at restart:

- task
- pid
- child

- sighand_struct
- signal_struct
- app_struct

System KDDM set objects do not vanish while a process is being migrated. However a node failure causes their inevitable loss. A successful process restart requires these system KDDM set objects to be recreated. For recreating system KDDM set objects one needs to know which object identifier is mapped to an object. Depending on the type of system KDDM set object, the object identifier has a resource specific meaning. The following table indicates this relation:

System KDDM Object	Object Identifier Meaning
task	process ID
pid	process ID
child	thread group ID
sighand_struct	custom unique ID
signal_struct	thread group ID
app_struct	application ID

Especially for the sighand_struct system KDDM object the identifier had to be inserted explicitly in the ghost image at checkpoint time, since it cannot not be derived at restart time.

However, recreation is not enough. In the *second recovery stage some of the KDDM set objects had to be reinitialised explicitly*. Therefore sighand and signal system KDDM set objects use data saved in the ghost image at checkpoint time. For the child system KDDM set object only the list of children is initialized if merely a single process (with no children, other threads) exists. The task and pid system KDDM set object are reinitialized by using the current values of the task to be restarted.

The approach to restart SYSV IPC shared memory is to *emulate* a subset of the functionality called by the system calls *shmget* and *shmat*. At restart the first restarted process of the shm application sets up a shm segment. The saved shm key is used to recreate a shm file in the special shm file system with the same name existed at checkpoint time. In the context of emulating *shmget* the shm user KDDM set is created. Furthermore all pages constituting the shared memory file are allocated in the Page Cache. This process initializes the allocated pages with the shm content of the checkpoint. The following processes, that link to the segment, do not initialize the segment, even if the content is available in their checkpoint file. Saving the shm segment content *per process* causes the processes to be independent - at restart no process needs to wait for one process owning the shm content. The existing functionality to read in process memory had to be extended for replaying shared memory content.

To avoid multiple set ups of the same shm segment, by e.g. a local child process or distributed processes, the shmid_object system KDDM set object type had to be extended.

For every SYSV IPC shared memory there is a *kernel internal identification* (shared memory identification) that is inserted into a process' namespace. This identification does not need to be the same at restart. However it is sufficient for the *shared memory key* to remain the same after restart since it is used at user level to address the shared segment. That's why no virtualisation of this kernel resource is necessary for a successful restart.

More information related to checkpoint and restart in the context of KDDM is provided under the *Kerrighed Wiki page - EPM and Checkpoint* [**?**].

3.3 Status of the current implementation

3.3.1 Checkpointing processes

The LinuxSSI Kernel Checkpointer is currently able to checkpoint and restart a single process executing one thread. Furthermore, processes having parent-child relation can be checkpointed and restarted as well. The following state information is extracted and recreated:

- *Thread state* - the registers (instruction pointer, stack, ...).
- *Process memory* - the process address space.
- *PID* - Each task is identified by a PID. Restarting a task with the same PID can be achieved using the pid system KDDM set because it supports PID allocation and PID reservation.
- *Pending signals and signal mask* - Signals not yet delivered (pending signals) can be saved and restored using the signal_struct system KDDM set.
- *Signal handler* - The callback functions to be executed when signals are delivered can be saved and restored using the sighand_struct system KDDM set.

3.3.2 Open Files

If a process has open files at checkpoint time, the file descriptors state can be saved and recreated. However no snapshots of these opened files are taken. The latter is planned to be supported by the new Kerrighed cluster file system *kdfs*.

3.3.3 Communication streams

It is planned to checkpoint/restart pipes and sockets, as soon as the implementation of the Kerrighed Dynamic Stream facility will be finished.

3.3.4 SYSV IPC Shared Memory Segments

The current implementation is able to checkpoint and restart processes using SYSV IPC shared memory segments. These processes may be in a parent-child relationship or not, they may run on one or several nodes.

Chapter 4

Installation and Configuration

For installing and configuring LinuxSSI please refer to *XtreemOS Deliverable D2.2.7*, section installation and configuration [**?**] .

Chapter 5

User Manual

5.1 Using Checkpoint/Restart

Before the checkpoint and restart functionality can be used some prerequisites have to be met.

First, the directory */var/chkpt* must be created by the administrator with read and write permission enabled. After successfully taking a checkpoint, a directory will be created under the mentioned directory. The PID of the checkpointed process serves as directory name.

Second, a so called capability has to be set. Certain system features can be en/disabled with capabilities. Such a capability has to be switched on for checkpointing to work. The command for enabling the checkpointing functionality is:
krgcapset -d +CHECKPOINTABLE.This command must be executed in the shell on which the process to be checkpointed will be started.

A checkpoint is created by issuing the following command: checkpoint PID.After successfully taking a checkpoint, a directory (checkpointed process' PID as directory name) should have been created under /var/chkpt. The following files will be created in this directory each including the serial number (SN) of the checkpoint in their name, e.g. '4711' for the file names mentioned below:

- *description_v4711.txt* (ascii file): short overview description of all files belonging to the checkpoint.
- *global_v4711.bin* (binary file): app_kddm_object KDDM object values as the applicationID, the serial number of the checkpoint and the kerrighed node mask.
- *node_5_v4711.bin* (binary file): description of local tasks involved in the checkpoint. The sample file belongs to node 5.

- *task_234_v4711.bin* (binary file): per task (e.g. PID='234') kernel structures such as registers, stack, signal mask, ...
- *task_mm_234_v4711.bin* (binary file): all pages of the process address space.

Repeated checkpointing of an application results in multiple files with the same base name but an SN increased by one at each time acheckpoint is taken. Checkpoints taken at different times can thus be distinguished.

An application is restarted by executing the following command: restart PID SN. Providing a SN allows to specify one out of many checkpoints taken during application execution.

For an application consisting of two or more processes, each of them having parent-child relationship, the PID of the common ancestor has to be provided to the restart command.

5.2 Limitations/Costs of Checkpointing Resources

The LinuxSSI-XOS kernel checkpointer development is an ongoing process. Currently, it is possible to checkpoint and restart applications: consisting of a sequential process (execution of one thread) and consisting of one or more sequential processes that are local (not distributed) and that have parent-child relation. Furthermore, the kernel checkpointer takes into account whether a process has open files and recreates this state but does not take snapshots of files modified by the process. It is planned that Kdfs (or KerFS) will provide a file snapshot functionality. Furthermore, neither sockets nor pipes are recovered at the moment. However both resources will be supported in future versions, depending on the development progress of the Kerrighed dynamic stream facility (Pascal Gallard from KerLabs is working on that).

Chapter 6

Conclusion and Future Work

At the beginning of the XtreemOS project an existing checkpointing implementation for Kerrighed was not operational. The main reason was the Kerrighed port to Linux kernel version 2.6 which required many changes in the Kerrighed kernel code. The existing checkpoint implementation did not take KDDM sets and their behaviour into account. The new approach is based on the integration of KDDM set object data into the checkpoint file and rebuilding KDDM set objects at restart. Thus KDDM-based checkpointing benefits from synergies between migration and checkpoint/restart.

It took considerable time to analyze all Kerrighed-related structures and concepts, especially the KDDM and ghost mechanisms. Several import and export functions related had to be extended to include additional process/kernel state information (e.g. system KDDM set objects) in the file ghost. Furthermore, some system KDDM set objects needed to be recreated and reinitialized manually during the restart phase of a process. The current implementation fits smoothly in the existing KDDM and ghost mechanisms and the LinuxSSI-XOS kernel checkpointer is able to checkpoint and restart sequential processes and also supports SYSV IPC shared memory segments.

Future work includes checkpointing/restart of following resources: files, pipes, and sockets. The latter two are both depending on Kerrighed's dynamic streams that are still under development. Another important goal is to also support parallel applications. Finally, during the next months we will design and implement the integration of the LinuxSSI-XOS kernel checkpointer within the overal XOS checkpointing/restart architecture. For example the LinuxSSI-XOS kernel checkpointer is able to coordinate sequential processes having parent-child relation. On the other hand the grid checkpointer is responsible for implementing a checkpointing strategy for grid nodes. Another example is kdfs (Kerrighed Distributed Filesystem) which will support taking snaphots of files.

It has to be analyzed to what extent there will be optimizations for the XOS SSI cluster appearing as a single grid node and what additional functionality requirements will show up within the LinuxSSI-XOS kernel checkpointer.

Bibliography

[1] J. Corbalan, G. Pipan, and T. Cortes. Requirements and specification of xtreemos services for job execution management d3.3.1. 2006.

[2] J. Corbalan, G. Pipan, and T. Cortes. Design of the architecure for application execution management in xtreemos d3.3.2. 2007.

[3] J. Corbalan, G. Pipan, T. Cortes, Matej Artac, Ales Cernivec, Eva Milosev, and Urus Jovanovic. Basic services for application submission, control and checkpointing d3.3.3 - basic service for resource selection, allocation and monitoring d3.3.4. 2007.

[4] M. Fertre. Prototype of the basic version of linuxssi d2.2.7. 2007.

[5] D. Margery and M. Fertre. T2.1.4 detailed specification and workplan. 2007.

[6] D. Margery, C. Morin, E. Focht, T. Ropars, A. Lebre, and O.D. Sanchez. Specification of federation resource management mechanisms. 2006.

[7] J. Mehnert-Spahn and Matthieu Fertre. Kerneldevelepmcheckpoint. 2007.

[8] Pascal Le Métayer. Design and implementation of basic checkpoint/restart mechanisms in linux d2.1.3. 2007.

[9] Pascal Le Métayer. System checkpointer : Conception and user pointer of view. 2007.