Project no. IST-033576

XtreemOS

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

T2.1.4 Detailed Specification and Workplan

XtreemOS Technical Report # 1

David Margery[a], Matthieu Fertré[c]

Report Registration Date: July 11, 2007

Version 0.1 / Last edited by David Margery / July 11, 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)	

[a] David.Margery@inria.fr
[c] Matthieu.Fertré@inria.fr

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	3/05/2007	David Margery	INRIA	Initial template
0.2	16/05/2007	David Margery	INRIA	Contribute to a callback registration interface and take Matthieu's remarks into account
0.3	31/05/2007	David Margery	INRIA	Refine Exemples
1.0	11/06/2007	David Margery	INRIA	Publish as techreport

Abstract

This document presents the system API envisionned for checkpoint/restart kernel mechanisms and details the initial implementation plan for those mechanisms in the context of the XtreemOS project which aims at Building and Promoting a Linux-based Operating System to Support Virtual Organizations for Next Generation Grids.

This work, initially planed as a derivative of the checkpoint/restart mechanims available in Kerrighed, is now based on the work around BLCR. This document therefore details the modifications to BLCR that are envisioned as part of the work on XtreemOS in order to identify possible collaborations with the original authors of BLCR. Of course, the aim and scope of the two projects being different, areas of possible disagreements are also identified so as to attempt to produce an implementation plan postponing incompatible developments for as long as possible.

Contents

Chapter 1

Introduction

1.1 Checkpointing in XtreemOS

The XtreemOS Grid Operating system will offer a native support for Virtual Organizations (VOs) in Linux. As described in the description of work (Annex 1) [1], the XtreemOS operating system is internally composed of two parts: XtreemOS foundation, called XtreemOS-F, and XtreemOS high level operating system services, called XtreemOS-G. XtreemOS-F is a modified Linux system, embedding VO support, checkpoint/restart mechanisms and providing an appropriate interface to implement XtreemOS-G services.

The Linux-XOS core mainly consists of the existing Linux operating system. Linux will be modified (with kernel patches), extended (with kernel modules) and configured (enabling optional components and/or exploiting the framework provided by the Linux system) according to the need to support virtual organizations as well as checkpoint/restart mechanisms, and in order to provide the features and API needed to implement the XtreemOS-G services.

On top of XtreemOS, a Grid application is executed on one or several Grid nodes. Such an application is composed of application units, any application unit being executed on a single Grid node. An application unit could be composed of a number of processes or threads of the host operating system. Checkpointing of applications is generally required due to the dynamic nature of Grid computing platforms, and in the case of XtreemOS is also due to the dynamic nature of virtual organizations. Application units running on a grid node may need to be moved to another node during their execution, or to be restarted, as a consequence of node failures, varying resource load, and changes in the policies of the involved VOs.

We should be able to restart an application unit running in the context of a VO on a different node with the same architecture within the given VO, eventually in a different administration domain. It is thus important to develop methods and interfaces to checkpoint/restart applications. XtreemOS approach is to extend Linux to integrate kernel level process checkpointing and to enable application unit checkpoint/restart.

Application checkpointing in XtreemOS is hierarchically decomposed into three levels of checkpointers: a kernel checkpointer, a system-level checkpointer and a grid-aware checkpointer. The two former checkpointers are implemented in XtreemOS-F, while the latter is a service in XtreemOS-G.

The kernel checkpointer adds two functionalities to a standard Linux kernel,

- it brings a process to a checkpointable state,
- it saves a snapshot of the state of a process to a file descriptor.

XtreemOS-F will need to augment the kernel and single system APIs to support Grid-enabled checkpointing. New APIs will have to be carefully designed according to the overall approach of minimal and local code changes, and existing systems and tools for checkpointing at the different levels have to be analyzed. This is especially important for the mechanisms providing restarted processes with information about changes in the environment (process id, IP address, hostname) and for those which allow restarted application units to coordinate and bring Grid-shared resources (e.g. network connections) to a correct state.

1.2 Overview of Application Checkpointing in XtreemOS

Based on the state-of-the-art, application checkpointing in XtreemOS involves three levels of checkpointer:

1. the kernel checkpointer, providing basic functionality to take a snapshot of a process.
2. the system checkpointer, providing checkpoint management at the application unit level, *ie* automatic checkpointing and snapshot management.
3. the grid checkpointer, providing checkpoint/restart facilities at the application level.

1.2.1 The Kernel Checkpointer

The kernel checkpointer offers a very basic checkpointing interface that enables

- checkpointing of a process,
- notification to the checkpointed process that it is about to be checkpointed,
- registration of callbacks from an application to tailor checkpointing to the application's needs,
- enabling and disabling of checkpoint from the application if it is written in a checkpoint aware way.

The callbacks are a means for the process to extend the boundaries of a checkpoint as made by the kernel level checkpointer.

1.2.2 The System Checkpointer

The system checkpointer is an OS service that manages checkpointing for an application unit. It registers checkpointing strategies and implements them.

- It will use resources given to it to call the kernel checkpointer or request those ressources on behalf of the calling process.
- It implements periodic checkpointing.
- It implements staged checkpoints.
- It implements checkpoint garbage collections.

1.2.3 The Grid Checkpointer

The grid checkpointer is the service responsible for supervision of checkpoints for an application: it applies the checkpointing strategy to all running application units.

- It registers the application units with the checkpointer service on the nodes running the application's application units.
- It provides ressources to store the checkpoints.
- It detects node failure and takes appropriate mesures to restart the application. It must therefore manage the credentials of the user running the application to enable restart.
- It is able to launch applications in a checkpoint/restart context.
- It coordinates taking a checkpoint of an application running on different nodes.

The system level and kernel level checkpointers are described in more details in the next section.

Chapter 2

Single Node Checkpointer

2.1 Process Checkpointing in Linux-XOS

Application requirements for checkpoint/restart mechanisms can be found in the WP2.1 and WP4.2 deliverables and the scope of these mechanisms is defined by the XtreemOS proposal [1]. The aim is to design and implement a checkpointer for a single node that can

1. be called from the command line or from the application,
2. checkpoint multi-threaded applications and communicating processes,
3. can capture the VO context of the application unit and
4. is implemented at kernel level

On the contrary, the checkpointer will not coordinate the checkpoint of an application running on multiple nodes, but it should be designed to ease the implementation of a global checkpointer.

It was initially thought that this work would be based on checkpoint/restart mechanisms available in Kerrighed, but it was quickly found out that this would involve to much work to extract only the checkpoint/restart mechanisms from Kerrighed's source code. Therefore, work for T2.1.4 is now built upon the work done around BLCR[2] as the two projects share most of their design principles.

BLCR is one of the most advanced open source implementation of a checkpoint/restart system for Linux. In particular, it is the only implementation with support for multi-threaded processes as well as for some implementations of MPI.

The design of BLCR, such as found in [2] states that,

1. checkpoint and restart are system calls, best implemented at kernel level,
2. checkpoint and restart should provide automatic support for the widest program features, without implementing distributed logic at kernel level,

3. checkpoint and restart should provide user-level callback interface to allow libraries and applications to support behaviors not handled in the kernel's checkpoint logic

To which XtreemOS adds

1. The checkpoint syscall should allow for different options
   (a) the system writes nothing. In this case, callbacks in the application will write all the information needed for restart. Here, the application uses the checkpoint file management framework provided by the kernel without using the kernel provided checkpoint mechanisms to provide the contents of the checkpoint file.
   (b) the system writes all it can (libraries, executable code, open files, IPC)
2. Restart should be able to take environment change into account
   (a) VO and user interface (command line or graphical (GUI)) context taken from restart, always
   (b) moved files or migrated services, optionally

In the following we describe our implementation plan for task T2.1.4: Design and implementation of basic application unit checkpoint/restart mechanisms.

Our aim with T2.1.4 is to augment BLCR with the following features:

- add an option to save the shared libraries used by the process in the checkpoint, rather than suppose that they will be present on the system when the process will be restarted.
- restart with a new security context (VO specific information) compatible with the one present in the snapshot of a process.
- at restart, provide information to the restarted process about the changes in the environment (process id, IP address, hostname).
- add a more complete strategy for files, and in particular implement the truncate strategy, where upon restart the more recent version of opened files are overwritten with the version present a checkpoint time.

2.1.1 Checkpoint/restart API

The following API for the kernel checkpointer is envisioned: First a structure is defined to exchange parameters about files (in the broad sense) between user space and kernel space :

```
struct {
  char * oldname ;
  size_t olname_len ;
  char * newname ;
  size_t newname_len ;
  int type ;
  int flags ;
} cr_ren_t
```

Then, the following system calls are defined:

- `/* checkpoint process pid */`
  `pid_t checkpoint (pid_t pid, int fd, int flags,`
  `cr_ren_t * fd_options, size_t fd_size);`
- `/* notify that a callback is ready for checkpoint */`
  `checkpoint_ready() ;`
- `/* prevent checkpointing */`
  `int checkpoint_disable() ;`
- `/* enable checkpointing */`
  `int checkpoint_enable() ;`
- `/* attempt to restart checkpoint stored in fd */`
  `pid_t restart (int fd, int flags,`
  `int * extra_fd, size_t extra_fd_size,`
  `cr_ren_t * fd_options, size_t fd_size);`
- `int register_callback(void * f, void * data, int flags)`
  `;`
- `int wait_for_checkpoint(void *f, void ** pdata, int`
  `timeout) ;`

This API should be seen as a set of new system calls as well as the interface extending the standard library implemented in C (libc) that will be provided to application programmers.

Adding new system calls to Linux is a difficult task. Therefore, in order to ease acceptance of our work by the Linux community, our implementation is likely to add this API by other means than new system calls, such as the `ioctl` call often used by modules. Therefore, at first, implementers should rely on the C interface.

Details

- Any checkpointing requests that are made for an application while checkpoints are disabled are queued until checkpointing is enabled unless the

NON_BLOCKING flag is provided, in which case taking the checkpoint will fail.

- On a call to `checkpoint`, if the KILL flag is set, the application is killed once the snapshot is taken.

- `fd_options` is a table of `struct cr_ren_t`, allocated by the caller and used to specify checkpoint or restart options on a file by file basis. Upon checkpoint, it can be used to get a list of checkpointed files, to specify the name of the checkpoint of a file that should be made or to pass specific checkpoint options for a special file openned by the application. Upon restart, it can be used to change the name of a file that will be restored to avoid overwritting the original copy or to specify a new path for files that were not part of the checkpoint and that are needed for a successfull restart.

- fd is an opened file descriptor.

- using the CLONE flags with `checkpoint`, the returned pid_t is the pid of a cloned process of pid, which can be checkpointed to a file descriptor later on, but not run, unless `restart` is called with fd pointing to `/proc/pid`.

- The restart system call attempts to restart the checkpoint using the credentials of the process running the restart. The flag MUST_REUSE_PID indicates that the restarted process should use the same pid (the call will fail if not possible). Otherwise a new pid will be used if necessary.

- when called with the NO_RUN flag, the restart system call only does not create a new process, but fills the `fd_options` table with the files related to the checkpoint. If `extra_fd_size` is 0, the size needed to get all information is stored in `extra_fd` upon return.

- The `register_callback` system call notifies the system that `f (data)` should be called before any checkpoint is taken. Called with the WAIT flag, the call is blocking until all the handlers are in place. The handlers can be managed by user level threads so that they are executed in a user level context enabling thread synchronisation. This will be notified to the system using the USER_THREAD flag and the user space handler should wait for a checkpoint using the `wait_for_checkpoint (f)` call. Here f is only a cookie used to coordinate the kernel with user managed threads. This call will timeout is the corresponding `register_callback` is not called. The `IS_SIGHANDLER` flag indicates that the `f` callback is of signal handler type and should be called in signal context.

2.1.2 Examples

Saving the libraries in the checkpoint

This would be done with a call with the SPMF (for Save Private Mapped Files) flag on the first call to `checkpoint` for that process. Subsequent calls can omit that flags, as the binary and the library files should not have changed. In this case, the first checkpoint file is kept and used to restore the libraries and the executable file and the most recent checkpoint file is used to restore all data that has changed in the address space of the process.

For restart, the extra_fd parameter is used to give the additionnal checkpoint files needed. This opens the possibility of a future implementation of incremental checkpoints with an arbitrary big number of intermediate files.

Clone the process as checkpoint

This is done using the CLONE flag during checkpoint. The cloned process can then be checkpointed to another media with a subsequent call to checkpoint, or restarted after /proc/pid of the clone process has been openned and given to the restart call

Changing the path of a data file

First call restart with the NO_RUN flag is the previous name was not known, then fill a `fd_options` table with the old name (including path) and the new name (including path) and call restart.

Set a handler in thread context

This is sample code to give an idea of how user level callback can be used.

```
int thread_context (void * some_function) {
  function_t * f =  some_function ;
  while (true) {
    void ** pdata ;
    res = wait_for_checkpoint(f,pdata,0) ;
    if (res)
       panic(''callback not registered'') ;
    f(*pdata) ;
  }
  return 0 ;
}

int register_user_callback(function_t f, void * data) {

  pthread_create(thread, attr, thread_context, f) ;
  register_callback(f, data, USER_THREAD) ;
  return 0 ;
```

```
}
```

Get notified that the context has changed

This can be done for any context parameter with a simple handler written in the folowing way:

```
int callback () {
  context_data_t context_data = get_context_info() ;
  int restarted = checkpoint_ready() ;
  if (restarted) {
    context_data_t new_context_data = get_context_info() ;
    if ( new_context_data != context_data)
       notify_context_change() ;
  }
  return 0 ;
}
```

2.1.3 Advanced modifications

We feel most of the modifications presented up to now could be implemented without changing BLCR to much.

Nevertheless, the current version of BLCR relies of the addition of a library using the LD_PRELOAD mechanism. We would like to extend BLCR is such a way that statically linked executables can be checkpointed, as these kind of executable file are bound to be used to ease the deployment of applications on a Grid. This could lead to the need for a kernel patch, which is proabably incompatible with the BLCR philosophy.

Handling of signals in thread context whithout using user level threads could also be implemented in a kernel context in a later version.

Chapter 3

Conclusion

We should be able to collaborate with BLCR, as authorized by XtremOS' IPUDC comitee, on a few of the issues mentionned here.

Appendix A

XtreemOS-F Design Strategy

As we have mentioned, our analysis involves evaluating changes and modifications to standard Linux systems which occur at quite different levels of the system implementation. Since the XtreemOS project in the overall aims at introducing new, Grid-enabling features into Linux, then, beyond the technical aspects of soundness, performance and scalability of design, all choices related to the specification of XtreemOS-F need to be evaluated in terms of impact on the Linux code base and on the underlying open-source software development process.

In order to enhance the impact factor of the XtreemOS project on Linux and to ease acceptance of the new features into the standard code base, the approach we follow aims at being minimal with respect to kernel code patches, and at keeping required changes localized in dynamically loadable kernel modules. We exploit as much as possible existing community standards and kernel APIs. This reduces the pressure to get VO related changes accepted by the kernel developer community.

The rationale is that widely accepted and useful features are maintained by the whole Linux community, less commonly used or esoteric features are maintained by smaller and smaller sub-communities, eventually by their originators. Whenever change maintenance needs to be carried on in synch with the evolution of system code, the effort required from those communities becomes a practical limit. By adopting existing features, pushing small changes into the mainstream and ensuring that the maintenance of more complex contributions is relatively independent form kernel evolution, we enhance the likelihood that XtreemOS-F, in its maturity, will successfully exploit the collaboration of the open-source software development community.

Bibliography

[1] XtreemOS consortium. Annex 1 - description of work. Integrated Project, April 2006.

[2] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Berkeley Lab Technical Report, 2003.