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 Linux
D2.1.3

Due date of deliverable: November 30th, 2007
Actual submission date: December 12th, 2007

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP2.1

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

Version 0.1 / Last edited by Pascal Le Métayer / October 18th, 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	18/10/07	Pascal Le Métayer	INRIA	Initial draft
0.2	6/11/07	Pascal Le Métayer	INRIA	Take comments from Christine into account
0.3	6/11/07	David Margery	INRIA	Work on executive summary and introduction
0.4	8/11/07	Pascal Le Métayer	INRIA	Work on executive conclusion
0.5	9/11/07	David Margery	INRIA	Work on the bibliography and minors revisions, as well as update to the latest version of the API
0.6	12/11/07	Pascal Le Métayer	INRIA	Minor modifications on API description
0.7	26/11/07	Pascal Le Métayer	INRIA	Take comments from Jérôme into account
0.75	05/12/07	Pascal Le Métayer	INRIA	Take comments from Julita into account
0.8	10/12/07	David Margery	INRIA	Take comments from Julita into account
0.9	11/12/07	David Margery and Pascal Le Métayer	INRIA	Merge last minute corrections

Reviewers:
Jérôme Robert (EADS), Julita Corbalan (BSC)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.1.4	Design and implementation of basic checkpointing/restart mechanisms	INRIA*, NEC

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

Contents

1 Executive summary

This deliverable describes the internals of basic application unit checkpoint and restart mechanisms for Linux-XOS. As this task leverages the work done at Berkeley Lab on BLCR, this deliverable describes the basic architecture of BLCR's implementation before focusing on our work.

In the context of XtreemOS, jobs are composed of application units running on different grid nodes. An application unit is then defined as a collection of processes under the control of one operating system instance (i.e. a grid node), either Linux-SSI or Linux-XOS. These processes could be multithreaded.

Due to the dynamic nature of virtual organizations, an application unit running on a grid node may need to be moved to another node during its execution. In the same way, a job may need to restart one of its application unit that has experienced failure of the node it was running on. Therefore Linux-XOS should implement methods and interfaces to checkpoint and restart applications.

In Linux-XOS, the checkpoint of a tree of processes running on a single node is based on BLCR (Berkeley Lab Checkpoint Restart). However, BLCR doesn't satisfy all the XtreemOS requirements. For instance, BLCR relies on the fact that all the librairies and the executable used by previously checkpointed processes are present at restart, which isn't a reasonable expectation in a migration context.

Therefore, our work has focused on 2 directions : produce an extension to BLCR to support additional functionality and define an API common to BLCR, Linux-XOS and Linux-SSI so that higher level software (application execution management software as well as user written software) can use different implementations of a kernel checkpointer. This deliverable describes design and implementation issues of our work.

2 Introduction

The aim of this document is to present the design and the implementation of basic checkpoint/restart mechanisms in Linux-XOS. Checkpointing in Linux-XOS is conceptually seperated in three layers: grid checkpointer, system checkpointer and kernel checkpointer. This document mainly describes the lower layer, the kernel checkpointer, as functionality for the system and grid checkpointer. System and grid checkpointer are being implemented in the framework defined by the Application Execution Management work of WP3.3.

The first section of this document presents how checkpoint functionalities can be decomposed into several layers.

As described in [2], Linux-XOS' implementation of checkpoint/restart mechanisms leverages BLCR checkpoint mechanisms for Linux. Hence, we provide background on BLCR in section 4. BLCR has been initially designed to checkpoint sequential and parallel applications (MPI application, for instance) running on clusters, but not in a grid context. One goal of Linux-XOS is to modify BLCR to adapt its uses to a job running on a grid.

Next, we present our work to enhance BLCR with the feature needed in a grid context, the ability to save executable and librairies linked to the checkpointed tree of processes.

Finally, this document presents the current iteration of our work on a stable and common API with the BLCR team. As we aim for the integration of our work on kernel checkpointing in the BLCR codebase, this API is still subject to changes after discussions and implementation attempts. Our use cases are also presented in this last section.

3 Overview of Checkpointing in XtreemOS

In XtreemOS, a job is defined as a set of application units executing on several grid nodes, and an application unit as a set of processes running on a given node.

Application checkpointing in XtreemOS is hierarchically divided into three levels: 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.

3.1 The Grid Checkpointer

The grid checkpointer is a service of the AEM (Application Execution Manager) responsible for supervision of checkpoints for an application: it applies the checkpointing strategy to all running application units.

- It manages resources to store the checkpoints.

- It detects failure of nodes executing the application units, and takes appropriate actions to restart the application. It must therefore manage the credentials of the user running the application to enable restart.
- It coordinates taking a checkpoint of a distributed application running on different nodes.

3.2 The System Checkpointer

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

- It offers an implementation of the kernel checkpointer for individual PC (not belonging to a LinuxSSI cluster), and an implementation of the kernel checkpointer for LinuxSSI clusters. In this document, we will only talk about checkpointing in Linux-XOS, based on BLCR. Kernel checkpointing for application running on a LinuxSSI cluster is discribed in [7].
- It uses resources given to it to call the kernel checkpointer or request those resources on behalf of the calling process.
- It implements periodic checkpointing.
- It implements staged checkpointing, meaning that a checkpoint could be seperated into several system calls (if the checkpoint must be coordinated, for instance).
- It implements checkpoint garbage collection.

3.3 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 need,
- enabling and disabling of checkpoint from the application if it is written in a checkpoint aware way.

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

4 Basic checkpoint/restart in Linux: BLCR, a kernel checkpointer

According to the state-of-art made in the [2] BLCR has been chosen as a starting point of the kernel checkpointer in the PC flavour of XtreemOS.

4.1 Introduction to BLCR

BLCR [5] [6] [4] is one of the most advanced open source implementation of a checkpoint/restart system for Linux. BLCR is developed by the Berkeley Lab and provides checkpoint/restart mechanisms via kernel modules. BLCR provides support for a wide program features, and support some MPI implementation (OpenMPI).

Nevertheless, some features (like network resources) are not directly supported by BLCR. Indeed, BLCR philosophy is that it is preferable not to implement a feature rather than to implement it only partially or in a way that is not completely transparent to the application. Thus, BLCR offers users a way to tailor the checkpoint of their application via callback mechanisms. In the following sections, we will give an overview of the architecture of BLCR, of the callback mechanisms, and how the linkage between BLCR modules and an application has been implemented.

4.2 A kernel module architecture

BLCR is made of three kernel modules, described below :

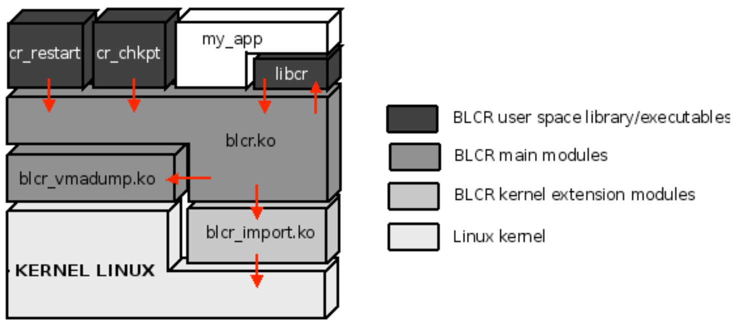

Figure 1: Link between Linux kernel, BLCR modules and BLCR library

4.2.1 blcr_vmadump module (In charge of the underlying dump of the process)

BLCR decided to begin their work with the VMADump kernel module [4] and extend and modify it to meet their needs. For historical reasons, the VMADUMP module has been kept separated from the “blcr.ko” kernel module.

VMADump is originally a part of Scyld’s Bproc (Beowulf Distributed Process Space) system. Bproc [1] provides operations to spawn processes on remote nodes, migrate processes, and send signals to remote processes in a Beowulf-style cluster. Remote spawn and process migration are implemented through VMADump. VMADump writes process state to a file descriptor. In Bproc, this file descriptor is a socket connected to a remote node. The other end of this file descriptor reads the process state as it is sent over the socket, and reconstructs the migrating process.

VMADump is designed mainly for this style of process migration, but can also be used for checkpoint/restart. Nevertheless, VMADump does not write the whole process state to the file descriptor. Part of the process state is originally managed by Bproc, but much of the process state is not transferred at all. Moreover, process migration in Bproc is voluntary, so that it’s not possible to force a process to migrate from one node to another without explicit cooperation.

Therefore, for BLCR, VMADump meets only some of their checkpoint/restart requirements and has been seen as a start point for a more complete checkpoint/restart mecanisms.

VMADump original module has been enhanced/adapted in order to support multi-threaded processes: first thread of the checkpointed process which accesses to the module writes all it can (header, linkage, credential, open files ..), whereas other threads only write the thread-specific portions of the given task.

4.2.2 blcr module (Main module)

The blcr module is the core module of the BLCR architecture, and behaves as an “orchestra conductor” during the time of the checkpoint. When the checkpoint module is first loaded, a special file is created (/proc/checkpoint/status). When it has been decided that the application should be checkpointed, this file is opened and an ioctl() call is done on it with a structure argument containing checkpoint parameters.

The initial effect of checkpoint’s ioctl call is to unblock the callback thread in the application, which upon returning to user-space runs the application’s thread-based callback functions, as described in figure 2. Callbacks are functions written by the user in order to tailor the checkpoint to his need as described in section 4.4. Once the last thread has finished to execute its callback, a checkpoint signal is sent to each of the threads in the application. Upon delivery of the signal, a BLCR library call is invoked. This routine runs any signal-based callbacks the application has provided, and then enters the kernel via an ioctl() call.

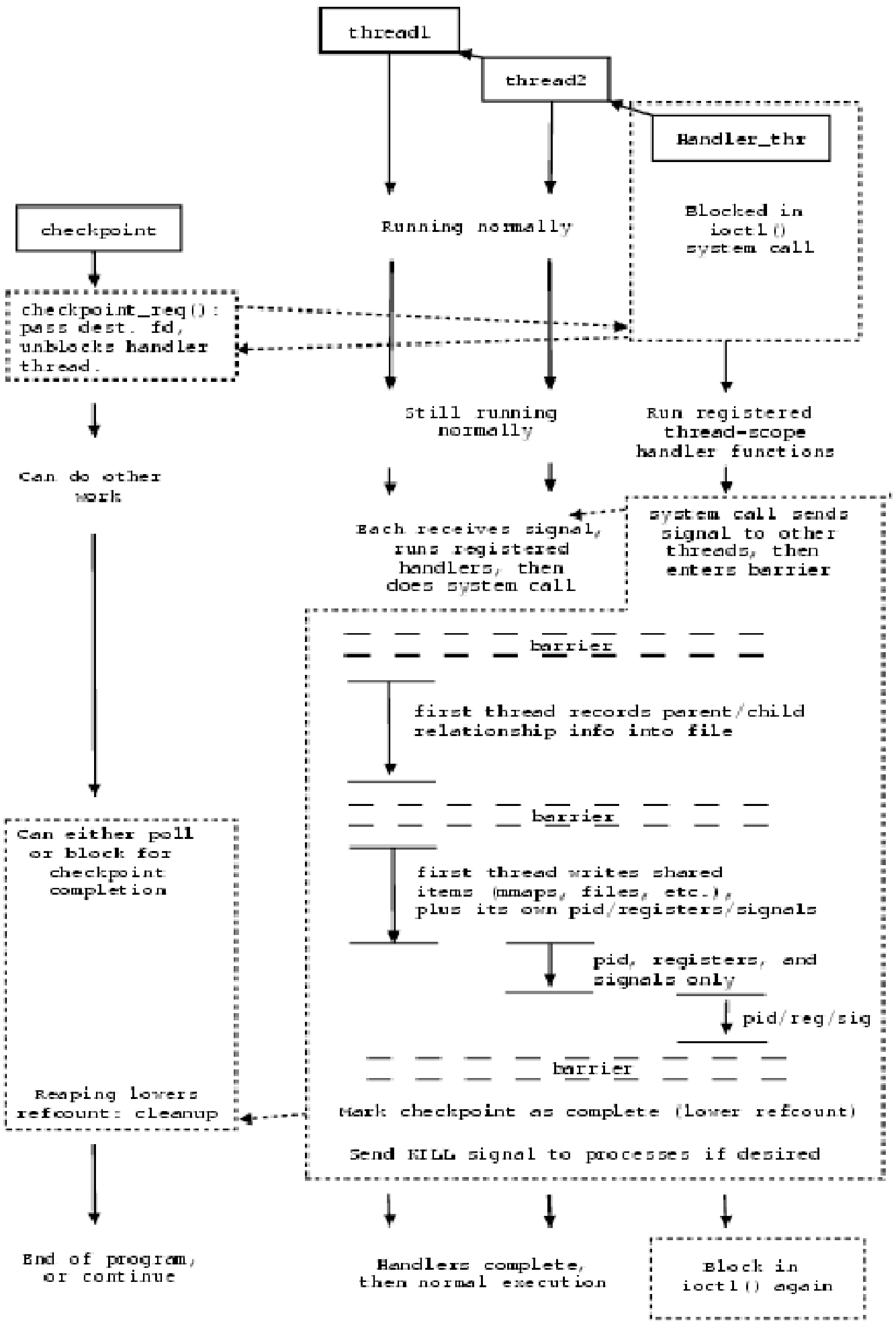
thread1
thread2
Handler_thr
checkpoint
Running normally
Blocked in ioctl() system call
checkpoint_req(): pass dest. fd, unblocks handler thread.
Still running normally
Run registered thread-scope handler functions
Can do other work
Each receives signal, runs registered handlers, then does system call
system call sends signal to other threads, then enters barrier
barrier
first thread records parent/child relationship info into file
barrier
first thread writes shared items (mmaps, files, etc.), plus its own pid/registers/signals
pid, registers, and signals only
pid/reg/sig
Can either poll or block for checkpoint completion
barrier
Reaping lowers refcount: cleanup
Mark checkpoint as complete (lower refcount)
Send KILL signal to processes if desired
End of program, or continue
Handlers complete, then normal execution
Block in ioctl() again

Figure 2: Checkpointing an Application, extract from [5]

Once done, all threads dump their state to the checkpoint file. As described above, BLCR mainly relies on a patched VMADump module to perform this dump, but in the original context of the VMADump use, some features were handled by Bproc. Thus, features that are not managed by VMADump are managed in the blcr.ko module. The table 3 sums up which functionalities are managed by each module.

blcr_vmadump.ko	blcr.ko
General purpose registers	task linkage
Floating point registers	credentials
Blocked signals	shared-anonymous mapping
sig_action structures	opened directory
Address space descriptor	opened regular file
Virtual memory area descriptor	opened pipes
Shared library name	character devices
Modified pages in library	
Virtual memory area descriptor	
Pages	

Figure 3: Managed functionalities

Restarting an application is largely the mirror image of checkpointing one, as described in figure 4. The restart utility is provided to resume jobs, and takes the name of a checkpoint file as its main argument. After filling out a structure containing the file descriptor that points to the opened checkpoint file, the restart utility performs an ioctl call, which results in a fork of the process which call the restart utility. The parent of this fork returns to user space, and waits for the restart to complete, while the child is cloned as many times by the kernel as there are threads in the application that is being restarted.

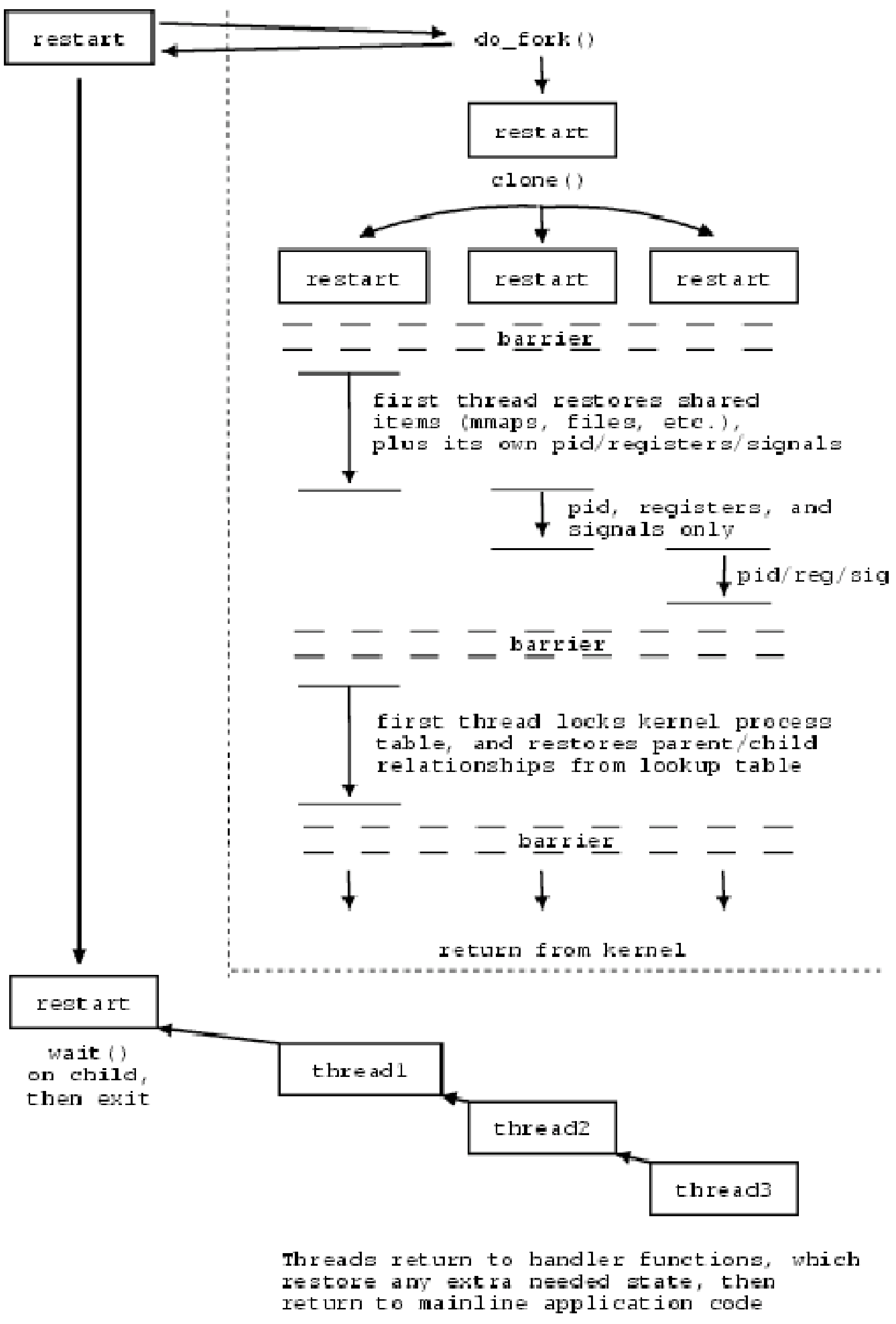
restart
do_fork()
restart
clone()
restart
restart
restart
barrier
first thread restores shared items (mmaps, files, etc.), plus its own pid/registers/signals
pid, registers, and signals only
pid/reg/sig
barrier
first thread locks kernel process table, and restores parent/child relationships from lookup table
barrier
return from kernel
restart
wait() on child, then exit
thread1
thread2
thread3
Threads return to handler functions, which restore any extra needed state, then return to mainline application code

Figure 4: Restarting an Application, extract from [5]

The newly cloned threads then perform a 'thaw' using VMADump (restore their state at checkpoint), taking turns as they read their information from the checkpoint file. One of the threads then uses the data from the checkpoint header to restore the pids of each of the threads, and their process relationships. At the end, the processes exit the kernel and enter user space, where their callback functions return and continue until they exit, after which regular application code is resumed.

4.2.3 blcr_imports module (In charge of exporting unexported symbol from the kernel)

The Linux kernel's interface to loadable modules provides less than the full symbol table used to link the general kernel code. However, to perform its task, BLCR needs access to both functions and data that are not exported to modules.

BLCR solution is to require that at build time, files *System.map* or *vmlinux* must be available in order to obtain absolute addresses of the needed symbols. These addresses are then hard-coded into the blcr_imports kernel module, which exists only to provide a symbol table with these addresses.

4.3 The LD_PRELOAD mechanism

The *VMADump* code, as well as many of the kernel functions on which it and BLCR rely, implicitly assume that they are operating on the "current" thread - with only a few exceptions the Linux kernel generally provides no mechanism to read or modify resources associated with another process. For this reason the preemption of a process to be checkpointed requires not only stopping all its threads (to prevent modification from racing with the checkpoint) but also forcing each thread to invoke a specific system call to allow checkpointing of its own resources.

In addition to the kernel module, which is responsible for the majority of the checkpoint and restart activity, BLCR is designed with a small shared library that contains the implementation of BLCR's extension interface to support callback mechanisms. Since this library is already a part of the BLCR architecture, the preemption activity has been combined with the management of callbacks, and a process which has to be checkpointed must load this library. For applications that use the callback interface, the library must already be linked explicitly to satisfy this linkage dependence. For applications that have not explicitly linked the BLCR shared library, a "cr_run" command is provided, which utilizes the LD_PRELOAD environment variable to load in the BLCR shared library.

Any process loading the BLCR shared library will automatically run this library's initialization function, which registers a signal handler with the lowest priority real-time signal number. When the kernel, in response to a checkpoint request, delivers the corresponding signal, this handler runs any callbacks registered through BLCR's extension interface and handles the implementation of critical sections in the interface (allowing atomicity of certain operations with respect to checkpoints). Once all registered callbacks are run, the signal handler makes the required system call to BLCR to allow the checkpoint to proceed.

4.4 The Callback and Critical Sections Mechanism

As it has already been said, BLCR excludes some features from its support list; however, it offers a simple way for a user point of view to interact with the checkpoint/restart mechanisms. Indeed, BLCR provides a way to register user-level callback functions, which are triggered whenever a checkpoint is about to occur, and which continue when a restart is initiated.

4.4.1 Signal-based callbacks

Signal-based callbacks are based on the fact that a signal is sent by BLCR to the concerned application before doing the call to VMADump (VMADump must operate on the current thread). Therefore, a way to let the user to execute some tailored functions before the dump is to register these functions as signal's handler. One significant drawback to this implementation is that it requires that user-registered checkpoint callbacks be able to run within signal handler context. Unfortunately, a large number of functions are not safe in a signal handler context (POSIX functions, for instance).

4.4.2 Thread-based callbacks

Thread-based callbacks have been implemented in BLCR in order to allow more flexible implementation of user callbacks. Thread-based callbacks are designed so that each callback runs in a different user-thread. The regular threads of the application are not stopped by BLCR during the execution of the threaded callbacks, to avoid deadlock. On a user-side, the drawback of such a choice is that the application which implements thread-based callbacks has to deal with synchronisation issues that concurrency between the application and the callbacks may introduce. One solution recommended by BLCR to cope with such a concurrency is to use a common “checkpoint lock”(reader/writer lock) if there is some concurrent access by multiple user threads. This lock is grabbed at every API entry, and also obtained (in ’write mode’) by the thread-based callback as its first action, and held throughout the checkpoint and restart, until the callback completes.

4.4.3 Critical sections

BLCR provides user-level code with “critical sections” in order to allow groups of instructions to be performed atomically with respect to checkpoints. These critical sections can be used by the user to guaranty that a section is not interrupted during its execution by a checkpoint.

This functionality is useful for instance in order to avoid an interruption of a network initialization, or to avoid that a checkpoint occurs while an asynchronous I/O operation is in progress.

5 Extending BLCR

In this part, we present the modifications done on BLCR in order to enhance BLCR with XtreemOS' specific needs. This API is both implemented by the LinuxSSI checkpointer and by the Linux-XOS checkpointer.

5.1 A new set of functionalities

Our aim is to augment BLCR with the following features, needed in the context of a checkpointer for processes belonging to a grid job:

- add an option to save the shared libraries used by the process in the checkpoint, rather than assuming that they are present on the system where the process is restarted. This is described as the SPMF (Save Private Mapped File) option in the rest of this document.
- 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, i.e. let the possibility for the user to save a list of the opened files during the checkpoint, and to offer him the possibility to change the path of these files at restart.
- restart with a new security context (VO specific information) compatible with the one present in the snapshot of a process.

5.2 Implemented functionalities

In T2.1.4, the SPMF option has been implemented.

Originally, BLCR saves only the filename of libraries and the executable that are used by the checkpointed process. Therefore, BLCR relies on the fact that these librairies and executable are at the same place when restart is made.

Goal of the SMPF option is to be able to restart a checkpointed application even if the libraries it uses or its executable have been moved/erased between the checkpoint time and the restart time, i.e. to generate a self-consistent snapshot.

This functionality is implemented as follows:

- At checkpoint time, after the dump made by the vma_dump module, a scan is made on all private/shared/executable maps of each process to detect the files to be saved.

- The files corresponding to the counted mapped files during the previous operation are saved into the snapshot file. As processes could have done some modifications during its execution, so an other scan is made to detect and save the pages that have changed.

- At restart time, previously saved files are regenerated in the /tmp directory, and these files are mapped. At least, saved changed pages at the end of the previous operation are copied in memory.

Three implementations of this option have been produced and sent as patches to BLCR maintainers. Ths first two where rejected due to insufficient adherence to BLCR's code structure. In particular, the way the second implementation interacted with the vma_dump module, responsible for the initial dump and restoration of the memory of a process was rejected because it added an option to that module for efficiency reasons instead of scanning the list of memory areas multiple times in order to avoid modifications to that module. The third patch is currently under review.

Detecting changes in the environment can easily be implemented as shown in the use cases described in section 6.5

The API to handle changes to paths for opened files is described in section 6. As this API is still under discussion with BLCR, no implementation work has begun.

Finally, security context, in the form of tokens stored in a process's security keychain are taken from the process requesting the restart and not from the checkpoint file. No specific implementation work was needed here. Nevertheless, testing hasn't taken place yet as integration work with other tasks and workpackages is still undergoing.

6 A common API for Kernel checkpointers

6.1 Introduction

Based on BLCR's structure that has been described in section 4, a first version of an API for checkpointing was described in [3] and sent for review to the BLCR authors. We present here a second revision of the envisionned API. All code presented in this section should be considered as a derived work of BLCR and is thus released under the same license (GPLv2 or later).

This API is composed of two strongly related parts. The first part of the API describes the system calls that would need to be added to Linux to support checkpointing whereas the second part of this API is the one that should be used by programmers.

Therefore, the API described here 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 in the Linux community, our implementation is likely to add this API by others means than new system calls, such as the ioctl call often used by modules. Therefore, at first, implementers of higher level checkpointing layers should rely on the C interface.

6.2 File Management

Correctly managing files in checkpoint/restart context is challenging. First, paths to files can have changed between the checkpoint and the restart. This happens in particular for large input files that are stored in a grid-aware file system. Here, the best strategy would be to save the file name as seen on the initial system as well as the file pointer. Then, on the system where the restart request takes place, the restart caller would specify the new location of the file as seen by the local filesystem. The second case would be that in the same application consuming data from a large file, a library creates and uses temporary files. These files need to be saved with the chekpoint file and restored on the system where the restart is requested.

From these two cases, we conclude that part of the checkpoint/restart process (file renaming for example) is best handled with some help of userspace and that there must exist a way for userspace or for the application itself to describe file by file strategies.

Therefore, at the lower (kernel) level, calls to *sys_checkpoint* never save the contents of opened files, but they return a list of files that should be saved with the application unit checkpoint file. These files will be needed for successful restart. It is expected that at a higher (library) level, these files will be aggregated to the application unit checkpoint file during the checkpoint phase, and that this aggregation will be considered as the checkpoint file. During the restart phase, these files will be made accessible on the node running the restart, and a translation table between their previous and current name will be given as a parameter to the low level restart call.

In the context of XtreemOS, this help from userspace could be provided by the AEM layer. The rational for this is that when checkpoint is used to migrate a process from one node to another, some mechanism needs to be setup in order to give the migrated process the same view of the filesystem hierachy as on the node it originated from, if the contents of all files is not saved with the checkpoint (case of large data files for example). There are many ways of acheiving this, either by specifying fixed mountpoints for XtreemFS volumes, by specifiying additional constraints when looking for a node to migrate the process to, or, as made possible by the interface described here, by providing filename translation.

With this design, there is no need for a node running a checkpoint to access the complete content of a file that needs saving with the checkpoint. Indeed, the higher level aggregation mechanism could either save a file containing the location of the needed files, or request that the filesystem save a snapshot of the file and only save a reference to the file. Thus, a lot of network traffic can be avoided.

Moreover, there is no need for the kernel to receive options about specific files upon checkpoint, as no file contents will be saved by the kernel. All file information and contents are managed at the library level, except for kernel specific information such as the corresponding file descriptor, the file pointer, etc... The file's name is used as a means of identifying a file in both cases.

Therefore, the following API for files is defined:

To start, a structure is defined to exchange parameters about files between user space and kernel space :

```
struct {
  char * oldname ; // null terminated string
  char * newname ; // null terminated string
  int flags ;      // options related to that file
  int type ;       // reserved for extensions
} cr_ren_t
```

Then, the following call is defined to enable callbacks to notify that a file should be saved.

```
void cr_add_saved_file (char * filename) ;
```

This call should be used for pathological cases where a process is using files present on disk but not necessarily opened when a checkpoint occurs. For example, in order to manage a large number of files, the process could keep in its memory a list of thousands of files that it uses or needs, but with only a subset of this list opened at a given time.

6.3 Callback Management

The following API for callback management is then defined. Whether threads that execute callbacks are started by the kernel or at user level is not specified here, but initial implementation will suppose user-level threads.

First, some types are defined.

```
//opaque type for callback identification
typedef ... cr_callback_id_t ;
// callback function type
typedef int (*cr_callback_t)(void *) ;
```

The following system call notifies the system that *f(data)* should be called before any checkpoint is taken.

```
cr_callback_id_t sys_register_callback(cr_callback_t f
    , void * data, int flags);
```

The following options are available for *flag*:

- WAIT: This call is blocking until the handler is in place (has called *wait_for_checkpoint(f,. . .)*).
- USER_THREAD: The handlers are managed by user level threads and are executed in a user level context enabling thread synchronisation.
- IS_SIGHANDLER: Handler is called in signal context; *f* is seen as a signal handler.

This function should be called by the user inside any function registered as a callback.

```
pid_t sys_wait_for_checkpoint(void *f,
                              void ** pdata,
                              int timeout) ;
```

It's a blocking system call bounded by the *timeout* value which blocks the threaded callback into the kernel until a checkpoint is requested.

Once requested, the user specific logic is executed. The following sample code gives an idea of how user level callback can be used:

```
int thread_context (void * some_function) {
  cr_callback_t * f = some_function ;
  while (true) {
    void ** pdata ;
    res = sys_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) ;
  sys_register_callback(f, data, USER_THREAD|WAIT) ;
  return 0 ;
}
```

The figure 5 sums up the callback registration mechanism:

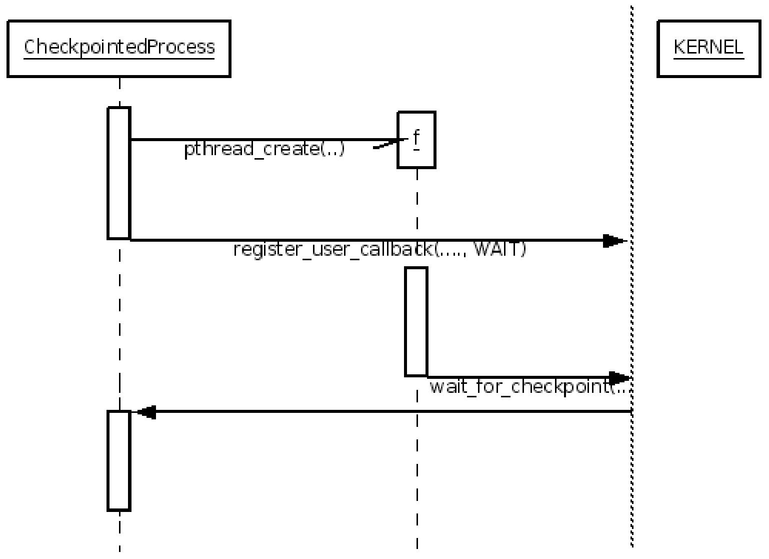

Figure 5: Callback registration mechanism

Finally, a callback should call one of those two functions once all preparations for checkpointing have finished :

```
// the callback has successfully finished preparing
   for checkpoint
// returns CR_DONE if the checkpoint has completed
//         CR_RESTARTED if the callback is called in
   restart context
int sys_continue_checkpoint() ;
// the callback has failed preparing for checkpoint
// and the checkpoint should be killed.
int sys_abort_checkpoint() ;
```

In the event a callback needs replaced or cancelled, the following function is defined

```
cr_callback_id_t
   sys_register_callback(cr_callback_id_t
       replaced_callback ,
                          cr_callback_t    f ,
                          void *           data ,
                          int              flags);
```

If f is NULL, the corresponding callback is cancelled.

Should a callback need information about the current checkpoint, are defined

```
struct cr_checkpoint_info {
  int signal ; // signal sent after checkpoint
  pid_t requester ; // pid of the process requesting
     the checkpoint
  pid_t primary_target ; // pid, pgid or sid of the
     requested checkpoint
  cr_scope_t scope ; //scope of the checkpoint :
     process,
  process_group , session
} ;

const struct cr_checkpoint_info *
   cr_get_checkpoint_info() ;
```

6.4 Core checkpoint calls

Finally, the following core checkpoint and restart functions are defined:

```
// temporarly prevent a checkpoint from starting
// blocks in case a checkpoint is in progress
int checkpoint_disable() ;
// re-enable checkpointing for this process
int checkpoint_enable() ;
```

The aim of these functions is to allow users to define critical sections in order to prevent a checkpoint to occur during the execution of the code located between the checkpoint*_disable()* and *checkpoint_enable()* functions.

Then a generic checkpoint request description is defined, as well as its initialisation function:

```
typedef struct cr_checkpoint_args_t {
  cr_version_t cr_version;
  cr_scope_t   cr_scope ;
  pid_t        cr_target ;

  int          cr_fd;
  int          cr_signal;
  unsigned int cr_timeout ;
  unsigned int flags ;
  cr_ren_t **  files ;
  size_t *     files_size ;
}

//checkpoint handle is an opaque type
// used to identify a checkpoint request
typedef ... cr_checkpoint_handle_t
```

```
void cr_checkpoint_args_init (cr_checkpoint_args_t *
    value) ;
void cr_checkpoint_handle_init (cr_checkpoint_handle_t
     *value) ;
```

At last, a checkpoint is possible:

```
pid_t sys_checkpoint (cr_checkpoint_args_t *args ,
cr_checkpoint_handle_t *handle) ;
```

A call to checkpoint triggers a checkpoint on the process specified by the pid. *args->cr_fd* is an opened file descriptor where checkpoint file will be written. Following flags are available :

- CLONE: 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*.
- NO_CALLBACKS: No callbacks are run before checkpointing. Useful if *cr_prepare_checkpoint* has been called.
- NON_BLOCKING: Default behaviour is that 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.
- SPMF: If the SPMF flag is set, binary and library(ies) used by the checkpointed process are saved in the file descriptor. Default behaviour is to save only the path of the binary and the library(ies). *fd_options* and *fd_size* are not used for the while, but exist for the reason noticed above.

args->signal specifies the signal to send to the target once the checkpoint has completed. Sending SIGKILL to the application at the end of the checkpoint process could be useful in a migration context. Sending SIGSTOP to the application at the end of the checkpoint process could be useful if the user wants to get a list of opened file descriptors via */proc/pid/fd*.

**(args->files)* points to an array of *cr_ren_t* describing the files to save with the checkpoint file written to *args->cr_fd* to form a complete checkpoint. The size of this array is **(args->files_size)*, and it it allocated by the library. Memory should be freed after usage.

If the checkpoint was non blocking, the following function should be used to poll for completion:

```
int checkpoint_poll(cr_checkpointhandle_t *handle ,
    struct timeval *timeout)
```

If timeout is null, the call is blocking. If *timeout is 0, the call is non-blocking. Otherwise, the call waits for checkpoint completion or the specified amount of time. A negative return value indicates failure, a positive value success and 0 time-out.

In the event the checkpoint is coordinated, the following function calls all callbacks, and the process should then be checkpointed with the NO_CALLBACKS flags.

```
pid_t sys_checkpoint_prepare (cr_checkpoint_args_t *
    args,
cr_checkpoint_handle_t *handle) ;
```

Once the checkpoint has been taken, this function continues the process that was previously stopped for checkpointing

```
pid_t sys_checkpoint_continue (cr_checkpoint_args_t *
    args,
cr_checkpoint_handle_t *handle) ;
```

Finally, the restart function, that uses a filename translation array of *cr_ren_t* and can consume more that one checkpoint file to enable restarting a process from more than one context file in the event incremental checkpointing is implemented:

```
pid_t sys_restart (int fd, int flags,int * extra_fd,
    size_t extra_fd_size,cr_ren_t * files, size_t
    files_size);
```

A call to restart respawn a previously checkpointed process stored in *fd. fd* is an opened file descriptor from where the checkpoint file is read. The followings flag are available:

- MUST_REUSE_PID: By default, the restarted process uses a different *pid* than the one that was used by the previously checkpointed process. The flag MUST_REUSE_PID indicates that the restarted process should use the same pid; the restart fails if it's not possible.

extra_fd, *extra_fd_size*, *fd_options* and *fd_size* are not used for the while.

```
pid_t poll_checkpoint(pid_t pid, int flag, void*
    result, int timeout) ;
```

As the checkpoint system call is not always a blocking call, a *poll_checkpoint* is provided in order to check/block the completion of a previous checkpoint call.*checkpoint* and *poll_checkpoint* must be called by the same process, so that only the process which has initiated the checkpoint can poll it.

- *pid*: pid of the monitored checkpointed process. If the value of this argument is -1, the poll is done on all the checkpoint initiated by the process which calls poll_checkpoint. Otherwise, poll is done on the specified *pid*.
- *flag*: By default, *poll_checkpoint* is a blocking call. If the NON_BLOCKING flag is set, *poll_checkpoint* becomes a non blocking call and is seen as a check operation.
- *result*: Return value of the checkpoint call.
- *timeout*: Used for bounding the poll operation.
- returned value: The returned value is the pid of the process which has been polled. It is useful to know which checkpoint operation has been completed if the poll is made on a set of processes.

6.5 API Usage Examples

6.5.1 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.

6.5.2 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 opened and given to the restart call.

6.5.3 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 ;
}
```

6.5.4 Get notified that the context has changed

This can be done for any context parameter with a simple handler written in the following 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 ;
}
```

7 Conclusion

Work on checkpoint/restart in Linux-XOS has started late (M14) due to a late recruitment. However, a working checkpointer for XtreemOS-F (BLCR project enhanced with SPMF functionalites 6.5.1) is already available [8] for all 2.6 kernels.

Integration of checkpointing mechanisms in XtreemOS-G is in progress: after a meeting in Barcelona with WP3.3 members in october 2007, a common implementation strategy has been agreed upon. This strategy implies that grid and system checkpoint level checkpointing should be implemented in the AEM (Application Executive Manager).

Next steps of the development will be to integrate patch and propositions about the API into the BLCR main stream; first steps in this way are promising. This API is not implemented for the time being as we hope to converge to a common API with the BLCR project members.

References

[1] http://bproc.sourceforge.net/.

[2] David Margery, Christine Morin, Luis Pablo Prieto, Haiyan Yu, An Qin, Erich Focht, Yvon Jégou, Adrien Lèbre, Oscar D. Sanchez, and Massimo Coppola. D2.1.1 linux XOS specification, November 2006.

[3] David Margery and Matthieu Fertré. T2.1.4 detailed specification and workplan. Technical report, XtreemOS technical repport, July 2007.

[4] E. Roman. A survey of checkpoint/restart implementations. Technical Report LBNL-54942, Berkeley Lab Technical Report, 2003.

[5] 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.

[6] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *In Proceedings of SciDAC 2006*, June 2006.

[7] John Mehnert Spahn. D2.2.3 design and implementation of basic checkpoint/restart mechanisms in linuxssi, November 2007.

[8] Yvon Jégou, Haiyan Yu, and Pascal Le Métayer. D2.1.4 prototype of the basic version of linux-XOS, November 2007.