Resources Virtualization in Fault-Tolerance and Migration Issues

G. Jankowski, R. Mikolajczak, R. Januszewski, N. Meyer, and M. Stroinski

Poznan Supercomputing and Networking Center
61-704 Poznan, ul. Noskowskiego 12/14, Poland
{gracjan,fujisan,radekj,meyer,stroins}@man.poznan.pl

Abstract. One of the low-level services Grids can benefit from is checkpointing. Unfortunately, the checkpointing technology imposes many common problems that are still open. The example of such often-encountered problems is recovering the identifiers of some checkpointed resources. This paper describes the idea of low-level virtualization of such identifiers. The mechanism allows overcoming the semantically imposed limitations.

1 Introduction

In this paper we focus on the coherency of restored process memory with reference to the surrounding environment and resources rather than recovering the processes themselves. Some variables in the process memory can hold the value of the identifiers that point to an external resource. After migration it can turn out that although the value has been restored properly, the external resource cannot be linked with this value any more. The solution to this problem can be the usage of transparent (from the program's viewpoint) virtualization of these resources identifiers.

2 Considered Resources' Description

Actually each resource type has its own semantics. Although the general idea of resources virtualization is invariable, the particular cases may require a little tune attention. In this paper we opt for tackling the System V IPC objects and their identifiers and keys. The System V IPC mechanisms include messages, shared memory and semaphores objects. Each object is unambiguously identified by its type and the key value. In order to gain access to a particular object, the process has to use one of the system calls *msgget()*, *shmget()* or *semget()* for messages, shared memory and semaphores, respectively. The key value, which determines the desired object, is passed to these functions and the return value is the object identifier through which further interaction with the object is performed. The operating system can forbid access to the object for any reasons (e.g. lack of rights to the object). If the process requires the object of IPC_PRIVATE key value then the operating system supplies a completely new IPC object, i.e. not shared with anyone. The problem is that during the recovery phase some other processes can already occupy the IPC objects that were originally used. Furthermore, the physical identifier of re-gained object will most probably have a different value than it originally did.

3 Virtualization

The main idea of the presented solution is intercepting the crucial system calls and replacing the virtual identifiers (passed by the process) with the physical ones. The function that is executed as a result of interception is called the *intercepting system call*. It is actually a wrapper. The wrapped function is named the *original system call*. The *original system call* is finally invoked by the *intercepting system call*. In some cases, the values that are returned by the *original system calls* are replaced, too.

The virtual identifiers that are used by the user programs are mapped to the physical ones by means of the *mapping tables*. Each mapping table defines a separated *mapping domain* of identifiers of the resources of the same type. There can be multiple *mapping domains* of a given kind of resources in the system at the same time, but one resource instance can be associated with only one *mapping domain*. The format of the *mapping table* depends on the kind of resources that are virtualized by this table. If any program has access to a *mapping table*, we say that this program is associated with the *mapping domain* that is defined by this table. Moreover, if a process is associated with the *mapping domain*, all its children inherit this association. To increase the legibility of the presented algorithms, we make an assumption that a single *mapping domain* can be associated with only one application (single- or multi-process). All domains associated with the application must be the domains of different type of resources. It means that for a given application and a given resource type, only one domain (i.e. only one *mapping table*) can exist. The next assumption that simplifies further study is that programs associated with different mapping domains should not communicate with each other. The *mapping tables* must be held in the shared memory. Access to this memory must be synchronized. The values of virtual identifiers must be unique within the scope of the *mapping domain*. However, the virtual identifiers from different domains can have the same values, even if these are the domains of resources of the same type. The first process that is created as a part of the multi-process program as well as the single process (the single-process application) is called the *root process*. All processes that are descended from the *root process* are named *branch processes*.

The virtual key and virtual identifier of all types of the System V IPCs have the same semantics. For this reason, in this section we refer to IPC objects generally instead of considering each of them separately. In order to simplify the following description, a special case when the object key equals IPC_PRIVATE was omitted.

The *mapping table* used for managing the *mapping domain* of the System V IPC object consists of four columns. The VKEY column holds the virtual key of an IPC object. The RKEY column holds the physical value of the IPC object. The VID column is the virtual value of the identifier of the IPC object and the RID column is the physical identifier of the IPC object.

During the initialization of the process, if it is the *root process* that is initialized, the memory for a *mapping table* it allocated, or if it is the *branch process*, it is attached to the *mapping domain* of its parent.

When the *xxxget()*[1] *intercepting system call* is invoked, first by means of the *mapping table* the virtual key (the one which is passed to the *intercepting system call*) is translated to the physical one. If the *mapping table* does not contain mapping for the current virtual key, the physical one is assigned the same value as the virtual one. The physical key is passed to the *xxxget() original system call*. If this function returns the error code, it is forwarded to the user process and the execution of the *intercepting system call* is finished. If the *original system call* returns the correct IPC object identifier (physical identifier), the RID column of the *mapping table* is searched for it. If the searching succeeds, the row that contains that value is marked and the virtual identifier is given the value of the VID column of the marked row. Otherwise, if the current *mapping domain* does not contain the virtual identifier of the same value as the one returned by the *original system call*, the new virtual identifier is given the same value as the just obtained physical one. If the physical identifier is not in the RID but in the VID column of the *mapping table*, the new virtual identifier is given an arbitrary value that is different from all the other within the VID column. If the VID column of the *mapping table* does not contain the just established virtual identifier yet, the new row is added to this table. The VKEY, RKEY, VID and RID columns are given the values of virtual key, real key, virtual identifiers and real identifiers, respectively. Finally, the value of the virtual identifier is returned as a result of the *intercepting system call*.

The *intercepting system calls* that operate on the IPC objects take the virtual IPC object identifier as a parameter. Before the *original system call* is called, the virtual object identifier must be translated into the physical one. To achieve that, the VID column of the *mapping table* is searched for the virtual value that has been passed to the *intercepting system call*. The row that contains the searched value is marked. The physical identifier that is passed to the *original system call* is taken from the RID column of the marked row. The value returned by the *original system call* is forwarded to the user process.

The values of physical keys and objects identifiers are the same as the values of the related virtual ones until the execution is not interrupted by migration or failure. The presented algorithm can be applied by the user-level and kernel-level checkpointing solutions, but it better fits the former case. To simplify the description below, the algorithms made by the *root process* and the *branch processes* are described separately.

The *root process* is recovered as the first one. It reallocates and restores the *mapping table* and then, for each row in this table, tries to request from the system the IPC object of the same key value as it was before the recovery phase

[1] *xxxget()* stands for *msgget()* or *shmget()* or *semget()*.

(by means of the *xxxget() original system call*). If, unluckily, the originally used key is occupied by another process, the new physical key is given an arbitrary value that is different from all other keys that are currently occupied. The value returned by the *xxxget() original system call* is the new physical value of the IPC object identifier. The RKEY and RID columns of the current row of the *mapping table* are updated with the values of the new physical key and identifier, respectively. If the type of the recovered IPC object is the shared memory and if the *root process* had it attached to its own memory space, the memory is attached to the *root process'* address space. Finally, the state or content (depending on the case) of the just recovered object is restored.

When the *root process* finishes the recovery phase, each *branch process* is attached to the *mapping table*. Generally, from the point of view of System V IPC objects, access to a correctly filled *mapping table* is sufficient for the *branch processes* to be executed properly. However, if the recovered IPC object is the shared memory, the *branch process* for each row in the *mapping table* additionally makes one more following step. If the shared memory, which is associated with the current row in the *mapping table*, was attached to the current process, it is reattached.

When a system call which releases the IPC object ends with success, the row that is correlated with it must be removed from the *mapping table*. When the *root process* is finished, the shared memory containing the *mapping table* must be freed.

4 Conclusion

The idea presented above is a kind of hint how to look at the resources virtualization issue rather than any formal and stiff standard proposition. An attempt to define and classify the basic notions and terms associated with resources virtualization has been made. The authors of this paper have implemented the presented conception in *psncLibCkpt* package, a checkpointing library used on user level, which supports the System V IPC objects virtualization (PROGRESS project: `http://progress.psnc.pl`).

References

1. J.S. Plank, M.Beck, G. Kingsley, and K.LI. Libckpt: Transparent Checkpointing Under UNIX, Conference Proceedings, Usenix Winter 1995. Technical Conference, pages 213-223. January 1995.
2. Hua Zhong and Jason Nieh. CRACK: Linux Checkpointing / Restart As a Kernel Module. Technical Raport CUCS-014-01. Department of Computer Science. Columbia University, November 2002.
3. Eduardo Pinheiro. Truly-Transparent Checkpointing of Parallel Applications. Federal University of Rio de Janeiro UFRJ. `http://www.research.rutgers.edu/ edpin/epckpt/paper_html/`.