CHAPTER 5

Migration of Process Credentials

5.1 Introduction
5.2 The Process Identifier
5.3 The Mechanism
5.4 Concluding Remarks
CHAPTER – 5

Migration of Process Credentials

5.1 Introduction

Every process in the system is allocated some unique process identification number by the operating system. As the process identification number (i.e. the process-id) is dynamic in nature, it belongs to the dynamic state of a process. Therefore, the dynamic process migration mechanism must perform migration of the process-id which is allocated to the process to be migrated from the source workstation to the destination workstation. This chapter describes a technique to resume the process on the destination workstation with the same identity value which it possessed on the source workstation before process migration takes place.

As discussed earlier in chapter 1, the dynamic migration of a process is superior to the static migration of the process. One of the important required characteristics of dynamic migration of a process indicates that the process must resume its execution on the destination workstation with the same identity that it possessed when it was checkpointed on the source workstation. The unique process id is also referred to as process credential in the available literature and manuals on operating systems.

At the time of a new process creation, the process is assigned a system vide unique process id by the host operating system. The unique process id is the only facility available in user-space through which the associated process can be made enabled to be controllable or traceable by the owner of the process and the system administrator. Moreover, the application programmers may manage
Chapter-5: Migration of Process Credentials

the process by the associated unique process id through the application program also.

Therefore, the process migration mechanism must consider the checkpointing of the unique process id possessed by the process which is to be migrated. The mechanism must support not only checkpointing of the unique process id but also make arrangements such that on resumption on the destination workstation, the process continues execution with the same unique process id. [S. Potter, 05] has suggested related work in which the migrated process can be resumed with a desired pid value; but that is accomplished through some intermediate virtualization layer such that certain processes above this additional layer will see the process with the desired pid value, but it will differ from the pid (the original one) with which the kernel identifies the process which is running within the pod- it lacks transparency and efficiency.

Although the LINUX Process Control Block i.e. the task_struct process descriptor maintains the process identifier value which is allocated to the specific process by the LINUX kernel, the LINUX kernel does avail neither a user-level nor a kernel-level facility through which the process identifier value of an existing process can be modified later (for consistency and security reasons). In addition, LINUX does not allow the application programmer to start (i.e. fork) a process with the certain process identifier value [N. A. Joshi, 12a].

In this chapter, we have proposed a mechanism to checkpoint the process credentials possessed by a process under execution and to resume the process with the same credential (process id) on a destination workstation along with an implementation prototype for the same. Moreover, we have discussed a new mechanism to inject the process id which is specified by us into the kernel space (during process resumption on the destination workstation at the time of new
process ‘fork’) by introducing a new system call in the operating system’s kernel, so that the resuming process on the destination workstation will start with the process id which it possessed initially before process migration. The suggested mechanism takes care that the injecting of new system call should not affect execution of other processes on that system [N. A. Joshi, 12a].

5.2 The Process Identifier

Before we discuss the essential steps which are required for assigning a certain process identifier value to the process control block of the resumed process, we summarize here the fundamentals of the process identifier.

In LINUX, the processes are assigned a unique number to exclusively address them in the global namespace. This number is defined as an integer within the kernel-space; which has been typedef-ed into a LINUX kernel data type __kernel_pid_t which is again typedef-ed to the pid_t type [W. Mauerer, 08]. (The idea behind this is that various UNIX/LINUX kernel implementations may use dissimilar data types for management of the process identifier values i.e. the pid values. E.g. certain LINUX implementations would prefer to alter from the ‘int’-type process-id to ‘short’-type process-id for the sake of conserving space on embedded systems which possess a very limited storage capacity). All processes in a group of threads have an identical thread group id i.e. tgid. The process having no threads has the same pid and tgid.

The process id and the other credentials of the process are directly stored in the associated process control block i.e. the process descriptor data structure task_struct, namely, in the data members pid and tgid as shown below [N. A. Joshi, 12a]:
<linux/sched.h>
struct task_struct {
...
pid_t pid;
pid_t tgid;
uid_t uid, euid, suid, fsuid;
gid_t gid, egid, sgid, fsgid;
... 
};

Both the data members are of the type `pid_t`. In turn both of them resolve to the kernel’s internal data type `__kernel_pid_t`. Usually, for the x86 architectures, an unsigned integer is used; it indicates that at the same time $2^{32}$ unique process-ids can be allocated to the processes. The ‘pid allocator’ subsystem in the LINUX kernel is responsible to allocate the unique pid-values to the newly fork( )ed processes.

5.3 The Mechanism

Our goal is to resume the migrated process on destination workstation with the same process-id value (which the process possessed initially on the source workstation before process migration took place); we describe below the mechanism for the same [N. A. Joshi, 12a].

We describe here a mechanism in the form of supplementing our new system call ‘setforkpid()’ in the LINUX kernel [N. A. Joshi, 12a]. Invoking the `setforkpid()` system call before restoring the migrated process on the destination workstation, allocates the ‘specific’ process-id value (i.e. the pid-value which the migrated process possessed initially on the source workstation before process migration took place) to task descriptor of migrated process which is to be restored.
1. Determine a unique identification number for our new system call `setforkpid()`, through which it will be recognized inside the kernel-space. And register the constant in the `unistd_32.h` header file (this file consists of all system calls’ numbers.).

```c
#define __NR_setforkpid 327
```
(The system call numbers 0-326 have already been assigned to the existing system calls in the existing kernel. So, we have chosen the number 327 here.)

2. Determine the prototype for our system call `setforkpid()`. It is supposed to receive the process-id value (i.e. the `pid_t`-value which is to be allocated to the process to be restored on the destination workstation) as a parameter from the user-space. And register the prototype in the `syscalls.h` header file (this file contains prototypes of all the system call handler functions).

```c
asmlinkage long sys_setforkpid(pid_t forpid);
```
Here, we define a kernel-space function `sys_setforkpid()` for the function associated with our system call `setforkpid()`. The implementation of our system-call in kernel-space and the user-space invocation to our system call `setforkpid()` must match the specified prototype.

3. Procure the unreserved array-index position (in the system call table within the kernel space), which will hold the memory address of our system call’s implementation function `sys_setforkpid()` at the system’s booting time.

Append the following line in the `syscall_table_32.S` file (it represents the system call table).

```assembly
.long sys_setforkpid
```

4. Implement our system call’s handler function `sys_setforkpid()`.
(a) Create a file setforkpid.c in a new directory syscall_setforkpid in the kernel source tree.

(b) Implement the handler in the file setforkpid.c.

```c
#include<linux/linkage.h>
#include<linux/types.h>
#include<linux/unistd.h>
...
extern void set_specific_pid(pid_t forkpid);

asmlinkage long sys_setforkpid(pid_tforkpid)
{
...
/*printk(KERN_INFO "sys_setforkpid():parameter
forkpid=%d\n", forkpid); */
set_specific_pid(forkpid);
...
return forkpid;
}

Here, the extern function set_specific_pid() has been exported by us in the kernel source, as discussed in later steps.

5. Update the ‘Makefile’ which is associated with the kernel source tree by appending our directory-name syscall_setforkpid/ to the kernel source’s compilation path, such as:
core-y += kernel/mm/fs/ipc/security/crypto/block/syscall_setforkpid/

6. Also add the following lines to the kernel’s source tree. The lines are responsible to maintain the specific process-id value that is passed to the kernel-space from the user-space:

pid_t specific_pid = 0;
Chapter 5: Migration of Process Credentials

```c
pid_t get_specific_pid();
EXPORT_SYMBOL(get_specific_pid);

pid_t get_specific_pid()
{
    return specific_pid;
}

void reset_specific_pid();
EXPORT_SYMBOL(reset_specific_pid);

void reset_specific_pid()
{
    specific_pid=0;
}

void set_specific_pid(pid_t the_pid);
EXPORT_SYMBOL(set_specific_pid);

void set_specific_pid(pid_t the_pid)
{
    specific_pid = the_pid;
    ...
    //set pid here
    ...
}
```

Here, the `specific_pid` is a kernel symbol introduced by us in the kernel-space; it is responsible to hold the process-id value arriving into the kernel-space from the user-space.

7. Obtain the value of last allocated pid from the kernel. And allocate appropriate process-id value to the migrated process which is to be resumed on the destination workstation.

```c
int last_pid = pid_ns->last_pid;

/*pid_ns is data structure which holds last allocated pid value; it is maintained by kernel*/
```
int pid;
/*we set this pid for our migrated process which is
to be restored in destination workstation*/
...
/*determine if the pid-allocation is requested
explicitly by our process-restore module.*/
if(strcmp(current->comm,"om_d")==0)
&
(!get_task_by_pid(get_specific_pid())
}
{
/*Yes, pid-allocation is requested explicitly
by by our process-restore module,
And right now duplicate pid does not exist in
the system.*/
pid = get_specific_pid();
/* set pid to the value of the kernel symbol
specific_pid. */
printk(KERN_INFO "om_d: set specific_pid to pid %d\n", pid);
reset_specific_pid();
/* reset the specific_pid to the default 0.*/
...
}
else
{
/*Otherwise, the pid-allocation is requested
implicitly by other processes except our
process-restore module, or right now some
duplicate pid already exist in the system. So,
let the kernel allocate the pid-value in its
default way.*/

pid = last_pid+1;
Here, om_d refers to the name of the daemon-module which performs restoration of the migrated process on the destination workstation. The, current->comm field refers to the name of the currently running process.

In order to keep the system in the consistent state, using the statement- ( !get_task_by_pid( get_specific_pid() ) ), we determines that there does not execute some other process with the same process-id value in the system (i.e. the pid-value which we want to allocate to our migrated process on the destination workstation).

8. Recompile this updated kernel source and reboot the system with the newly compiled kernel.

9. END

Thus, the mechanism for restoring the migrated process on the destination workstation with the original process-id value (which it possessed on the source workstation before migration took place) is shown above.

Here are the steps to be followed by our user-space om_d module on the destination workstation to restore the migrated process with a desired process-id value [N. A. Joshi, 12a]:

1. On the destination workstation, the om_d process migration daemon-module receives and unpacks the binary checkpoint-image file (this file represents binary checkpointed image of the
migrated process, which has been sent from the source workstation).

2. From the received checkpoint-image file in above step, fetch checkpointed value of the process-id which is to be assigned to the process to be restored on the destination workstation, say specific_pid.

3. Execute our system call before restoring the migrated process on the destination workstation.
   
   setforkpid(specific_pid);
   /* fork(…); */

Thus, the above discussed mechanism assists the process migration module to resume the migrated process with a desired process identifier value on the destination workstation. However, in order to restore the migrated process with the original process-id value, the same pid-value should be made available for allocation on the destination workstation; otherwise the default system generated pid-value is assigned to the process. We have maintained this aspect as there cannot exist more than one process with the same identity, otherwise the system will be dissolved into the inconsistent state which may probably lead to the overall system crash.

Moreover, checkpointing of the existing values of the process credentials can be performed with the help of the following [N. A. Joshi, 12a] data structure:

/*Obtain the Process Control Block i.e. the task descriptor task_struct*/
struct task_struct* ptr_to_pcb=find_task_by_pid(pid);
...
//lock the process descriptor
task_lock(ptr_to_pcb);
    /*Now checkpoint values of credentials of the
     desired process e.g.
     ptr_to_pcb->pid,
     ptr_to_pcb->tgid,
     ptr_to_pcb->uid and others..*/
    task_unlock(ptr_to_pcb);
    //Unlock the process descriptor

Thus, the process-credentials may be checkpointed with the help of the above shown statements.

5.4 Concluding Remarks

In the beginning of this chapter, we have described the significance of the process identifier i.e. the process-id, which is unique for every process running in the system. The chapter discusses the implementation of our novel system call setforkpid(), which we have supplemented to the LINUX kernel. This newly added setforkpid() system call helps the process migration mechanism to resume the migrated process with the same process-id which the migrated process possessed on the source workstation before migration to the destination workstation.