Preserving System Call Details

3.1 Introduction
3.2 Design Characteristics
3.3 The Process and Process Descriptor
3.4 The Mechanism
3.5 The System Calls
3.6 Concluding Remarks
CHAPTER – 3

Preserving System Call Details

3.1 Introduction

From the user application’s perspective, the LINUX kernel is a transparent operating system layer — it is never really noticed, though it always exists. Processes never know about which memory regions are occupied by them, which regions are swapped, though the processes are involved in continuous communication with the kernel to request various system resources and thereby to get their work done. In order to meet their requirements, the processes use standard library routines which, in turn, call kernel functions. Here, the standard library is an intermediate layer to standardize and simplify the distribution of kernel routines across different kernels, whereas the LINUX kernel is an actual set of routines (services) that perform various system functions.

In order to make the kernel’s capabilities and routines available to user-space applications, the LINUX kernel facilitates ‘system calls’ through which the kernel routines can be accessed from the user-space. The checkpointing and process migration mechanism must emphasize on the mechanism of checkpointing the system calls in order to carry out reliable process migration. This chapter focuses on the mechanism for checkpointing the details of system call which is being executed by the victim process.

Dealing with the LINUX system calls requires understanding of the concept of a process. A process is represented in the operating system by a process control block or the process descriptor; it contains the information needed to manage the process. The process concept and the LINUX implementation of the process control block (PCB) i.e. the
task_struct data structure, are discussed in this chapter.

We have organized the overall presentation into two major mechanisms and their corresponding implementations: the process migration mechanisms and the load balancing mechanisms. The work regarding the process state checkpointing and the process migration mechanisms has been described in the Chapters 4 and 5 including the current one. The work on load balancing has been described in the sixth chapter. The, primary behavior of these mechanisms in the form of interactions among them has been outlined in this chapter. The mechanisms and their implementations described in this thesis have been developed with certain design characteristics and principles, which have been detailed in this chapter.

### 3.2 Design Characteristics

The well-behaved load-balancing and process migration mechanisms are difficult to achieve in practice. There are four important characteristics which must be focused upon, and we have made efforts to make the proposed mechanisms meet these characteristics.

#### 1. Transparency

The proposed mechanisms should provide the maximum transparency to not only the end user but also to the application programmer and the system administrator. The system is transparent to the end user in such a way that it itself manages the workload information of the networked workstations automatically. The system does not impose the burden about management of workload-information of the networked workstations on the end-user or the system-administrator. Unlike the LIBCKPT [J. S. Plank, 95], the proposed mechanisms provide maximum transparency to the application programmer by keeping the programmer free from any kind of forceful modification to the existing applications and the design of new applications (which are
yet to be developed). In order to use our process migration mechanism, there are no changes required to even a single source file of already running applications. This becomes advantageous to the application programmer, because it saves the programmer’s efforts by keeping him away from the lengthier tasks of application’s source tree management, version management, modification of the Makefiles (Makefiles are configuration files for automated compilation and linking of the application’s source tree) as the application source code remains unchanged. Such transparency does not lead the application programmer towards recompilation and relinking of the entire application’s source code.

Furthermore, our process migration and load balancing mechanisms have been implemented independently, i.e. they have not been designed in the form of checkpointing-library-routines such as in BLCR [J. Duell, 03] and Condor [M. Livny, 00]. Such library-routines are generally designed to be used by the application programmer; the application programmer has to include the checkpointing-library or header files in the application’s source files, thereby making the application and the checkpointing library dependent upon each other. The way of checkpointing mechanism implementation by means of library routines has two severe limitations: (i) any modifications (i.e. corrections or enhancements) in the checkpointing library files would further necessitate recompilation and relinking of the application itself; and (ii) this approach turns out the process migration and load-balancing mechanisms into passive mode; i.e. by introducing the checkpointing-library routines to the victim application’s source-code, the migration and load-balancing decisions get handed over to the application (i.e. the application developer) itself. Thus, the overall system behavior and performance will now have to rely upon the application’s nature and capacity (i.e. how timely, efficiently and correctly the application developer uses the checkpointing library). There should be no compulsion that the applications must be written
using some third-party libraries or toolkits, and without restricting their use of common operating system services.

Furthermore, our mechanisms do not require any type of modifications or application of customized patches to the already installed system software including the system libraries, compilers, linkers and loaders. Moreover, the process migration system and the load-balancing system should maintain independence of the originating workstation of the process in the distributed system. The complete process migration mechanism (which also wants to provide the fault tolerance advantage) should neither leave nor generate any type of residual dependency on the originating workstation of the victim process once the process migration is over, otherwise it reduces the usage of process migration by insisting the source-workstation to serve the process even after it has migrated to the destination workstation. This concept does not leave resources occupied on the originating workstation and free up not only the resources but also the workstation itself.

Unlike some of the existing process checkpointing and migration systems discussed earlier, such as MOSIX [A. Barak, 93] and SPRITE [F. Douglis, 90], our mechanism does not leave any type of residual dependency on the originating workstation once the process migration is completed.

2. Consistency
The suggested process migration mechanism provides consistency to the execution of the resumed process after migration to the destination workstation.

At the time of process checkpointing on the source workstation of the process, the victim process could be executing some system call. After migrating to some destination workstation, the process must resume
execution from the system call which it was executing before migration took place on the originating workstation. Our process migration mechanism provides such consistency.

Moreover, in LINUX, every process is assigned a unique system-wide process identifier, using which the process can be controlled and communicated with in the user-space; i.e. the process-identifier or process-id or PID is of critical significance in user-space. The same process-id value should be given to that migrated process on the destination workstation (provided that the desired process-id value is available for allocation and it is not already allocated to some another process). Such an important feature is absent in the existing work such as CRAK [H. Zhong, 01] and CryoPID [CryoPID]. The suggested solution in this dissertation provides the above described consistency by managing allocation of the original process-id value to the migrated process on the destination workstation if the desired process-id value is available for allocation.

3. Optimality
The process migration mechanism is surely an overhead causing mechanism; it should consume least-time to carry out its operations. As discussed earlier in chapter 1, the process migration mechanism consists of three major phases: the process-state checkpointing, migration of checkpointed process image and resumption of the migrated process image on the destination workstation. Among these three phases, the first phase of process-state checkpointing can be managed such that it consumes least time. On the other side, management of the other two phases does not belong to the scope of this thesis, because of either limitation of network traffic and bandwidth or the specific process scheduling priorities set on the destination workstation.
Therefore, in order to optimally carry out the mechanism of process migration, we focus on the task of checkpointing the memory regions (which are occupied by the victim process) in the minimum time. The suggested mechanism in this dissertation provides the characteristic of optimality in process migration mechanism through optimized checkpointing of the memory regions.

4. Flexibility

The load-balancing and process migration mechanism should be flexible enough for usage for the end user. The concept of load-balancing reveals some rigidity – it always tries to exactly equalize the workload of every workstation in the network. This results into heavy uncertain resource usage pattern and complex migration decisions. Instead the more general term load-sharing is more meaningful and beneficial which intends to share the workload of heavily loaded workstations with the idle or lightly loaded workstations in order to gain some benefits of process migration. Nowadays, these two terms – load-balancing and load-sharing are used quite interchangeably and they both are roughly used for the purpose of distributing the load of workstations of the network [M. Nuttall, 97].

The mechanism serves the characteristic of flexibility by facilitating the load-balancing service both to the client and the server (here the server represents a dedicated workstation that is responsible to carry out the tasks of workload information management of all workstations in the network and issuing the process migration orders to particular workstations).

The suggested solution provides implementation of both the techniques: server-managed load-balancing and client-initiated load-balancing. With the feature of client-initiated load-balancing, the system also achieves an additional advantage of the process migration mechanism and it is fault tolerant. Thus, as discussed earlier in
chapter 1, the user may utilize the feature of client-initiated process migration in case of partial system failure.

Moreover, the system provides the characteristic of flexibility by facilitating the user to migrate a process of his choice to the destination workstation; thereby the system enables the user to migrate a ‘specific’ process from some partially failed (i.e. weak) workstation to some safe (i.e. healthy) workstation to gain the benefit of fault-tolerance. Also, the process migration mechanism must support the existing installed open source operating systems (such as LINUX). It should neither necessitate transition to the use of some new operating system nor heavy changes to the existing operating systems.

**5. Efficiency**

The process migration mechanism must be efficient. It should cause the minimal overhead. Therefore, in order to reduce the overhead and achieve higher efficiency, the process migration mechanism should avoid the implementation of additional virtualization layer (such as ZAP [S. Osman, 02] and Xen [P. Barham, 03]) between itself and the underlying operating system. Instead, the system’s design should be such that the victim-process (the process which is to be migrated) and the process migration system should be allowed to be in direct contact (i.e. interaction) with the underlying operating system to achieve an objective of the transparent, fastening and efficient process migration mechanism.

**6. Responsiveness**

The process state migration mechanism should provide low delay in the checkpoint-request - i.e. the time between when a checkpoint is requested and when the checkpoint begins to be produced should be significantly less than the time required to produce the checkpoint.
This excludes techniques such as waiting for the program to reach to a known, simple, consistent state.

### 3.3 The Process and Process Descriptor

The concept of a process represents a program under execution or a running instance of a program. A binary program remains as a passive entity until it is not dispatched for execution; as soon as it is submitted for execution, LINUX generates an active entity which is called a ‘process’. The process incorporates the runtime information and environment of a binary program under execution [W. Stallings, 09]. A binary program is considered as a passive entity until it is launched for execution, whereas the process can be thought of as a program in action. The processes are dynamic entities as they are continuously changing as their machine code instructions are executed by the central processing unit.

The runtime information associated with a process includes: the microprocessor context, the memory regions occupied by the process (i.e. address space of the process), the process credentials like process identifier, priority of the process, details about the input/output files being currently used by the process, signal information and accounting details.

All processes do have their own such runtime information. In order to well-maintain this information, the LINUX kernel (core part of the operating system) maintains a per process data structure called ‘process descriptor’; the process descriptor concept is implemented as a LINUX C-structure called ‘task_struct’. And the linked list of all existing process descriptors i.e. the task list looks like the one shown in figure 3.1.
As nearly all of the work described in this dissertation has its roots reaching to the task_struct process descriptor, we describe in brief the process descriptor and its some of the component data members of our interest here [W. Mauerer, 08]:

The process descriptor task_struct is defined in the header file sched.h under the linux sub-directory in the LINUX kernel-header tree. As a new process is created, a new instance of the process descriptor task_struct is allocated from system memory to the newly created process. In the section that follows, we describe various members of the task_struct data structure.

```
// linux/sched.h
struct task_struct
{
    ...
    volatile long state;
```

The data member state describes a code-value for the current state condition of the process. Each process in the LINUX system is in exactly one of the five different states. This state is
represented by one of the five flag-values: TASK_STOPPED, TASK_RUNNING, TASK_UNINTERRUPTIBLE, TASK_INTERRUPTIBLE and TASK_ZOMBIE.

unsigned long flags;
The data member flag holds some indicator-values, indicating everything from whether the process is being created (PF_STARTING) or exiting (PF_EXITING), or even if the process is currently allocating memory (PF_MEMALLOC).

long utime, stime, cutime, cstime, start_time;
The data fields utime and stime hold the time the process has spent in user mode and system mode respectively, while cutime and cstime data fields hold the totals of the corresponding times for all child processes. The data field start_time holds the time at which the current process was generated.

unsigned int ptrace;
The data field ptrace represents whether the process is being traced by the ptrace system call or not.

long counter;
The field counter holds a value which gives the number of ticks for which the process can execute before a scheduler can execute.

unsigned long policy;
The field policy holds the scheduling policy applied to this process; it may have the values such as SCHED_RR, SCHED_FIFO and such other scheduling policy constants.

unsigned long rt_priority;
The field rt_priority holds a priority value assigned to the process.

sigset_t blocked;
The blocked signals by the process are represented by the field blocked.

struct signal_struct *sig;
The process may register signal handlers for certain signals. The signal handlers are represented by the field `sig`.

```
struct sigpending pending;
```

The list of signals which are still to be processed by the process is represented by the field `pending`.

```
struct task_struct *next_task, *prev_task;
```

All processes form a doubly linked list with the `next_task` and `prev_task` pointers of the `task_struct` type.

```
struct task_struct *p_pptr,*p_cptr,*p_ysptr,
    *p_osptr;
```

There exist family relationships between the processes in the system, which are represented by the above listed pointers. One of the pointers is `p_pptr`, which points to the parent process’s process descriptor. The pointer the `p_cptr` is used to enable a process to access all its children processes, the `p_cptr` points to the process descriptor of the last created child process. The process descriptors of the children processes are linked with each other by means of a doubly linked list by the pointers `p_ysptr` (next younger sibling) and `p_osptr` (next older sibling).

```
struct linux_binfmt *binfmt;
```

The `binfmt` member represents execution file format of the associated binary file with the process.

```
struct mm_struct *mm;
```

The data member `mm` represents the address space of the process. It represents the memory descriptors of the process.

```
struct thread_struct tss;
```

The `tss` field represents the stored context of the process and points to the `thread_struct`. The `thread_struct` descriptor represents the underlying processor’s CPU-specific state.

```
pid_t pid;
```

Every process has its unique process ID called `pid`. In user space, we identify the process by this number.
(Here the datatype `pid_t` is the `typedef` of the basic datatype `int`).

```c
int leader;
```

It represents a leader in the thread group.

```c
tuid_t uid;
gid_t gid;
int ngroups;
```

LINUX allows a process to be assigned to a number of user groups at the same time. The individual user IDs and group IDs are handled by the above described fields. Each process may belong to a maximum of `ngroups` groups, which are held in the `groups` component of the task structure.

```c
char comm[16];
```

The data member `comm` holds the first 16 characters of the program’s name being executed by the process.

```c
...
```

};

### 3.4 The Mechanism

The process migration and the load balancing solutions described in this thesis have been divided into the following primary modules:

1. **The LoadBalancer** – The LoadBalancer mechanism and its consequent implementation package is responsible to perform load sharing among the workstations in a network.

2. **The LoadPortal** – The LoadPortal mechanism and its consequent implementation package is responsible to maintain the workload information of a workstation and communicate the same to the other workstations in the network.

3. **The ProcessMigrator** – The ProcessMigrator mechanism and corresponding implementation package is designed to carry out checkpointing of the current state of the victim process and to perform migration of the checkpointed process from the source
workstation to the destination workstation.

The ProcessMigrator module has been detailed in the chapters 4 and 5 apart from the current one; whereas, the LoadBalancer and the LoadPortal modules have been detailed in the chapter 6. In spite of their detailing, few of the significant interactions of the above listed mechanisms have been outlined below in brief. The LoadBalancer package determines the overloaded workstation wi with the help of LoadPortal package.

1. The LoadBalancer package is assigned the process-identifier of the victim process (the process which is to be migrated) Pi on the overloaded workstation Wi.
2. The LoadBalancer package determines an appropriate destination workstation Wj for migration of the process Pi using the LoadPortal package.
3. The LoadBalancer package orders the ProcessMigrator package to migrate the process Pi from the workstation Wi to the workstation Wj.
4. The ProcessMigrator suspends execution of the process Pi, extracts its state, and checkpoints the extracted state to persistent storage. Such a saved process state is called checkpointed image of a process.
5. The ProcessMigrator transfers the checkpointed image of the process Pi to the destination workstation Wj.
6. On the workstation Wj, the peer ProcessMigrator package extracts the process state from the received checkpointed image of the process Pi.
7. On the destination workstation Wj, a process instance Pj is created from the extracted process state of the process Pi.
8. The process Pj is now ready to be dispatched for execution on the destination workstation Wj.
The phases outlined above have been described in this chapter and the following chapters.

### 3.5 The System Calls

A system call is used by application (user) programs to request service which is provided by the operating system kernel. The following statements describe the significance of the system calls. The processor operates in two modes: user mode and kernel mode. In the user mode, the processor can execute only non-privileged instructions i.e. a subset of its instructions (such as arithmetic instructions), whereas in kernel mode, no such restriction applies, the full instruction set is available to the processor for execution. The LINUX kernel executes in kernel mode and therefore it can access a system’s hardware and software resources (e.g. primary memory, storage media, microprocessor registers and such all other resources) directly, whereas the user application executes in the user mode, and therefore it is not given direct access to these resources. Such a restriction on user applications is maintained so that the kernel can keep the overall system in safe state and secure from malicious user applications [D. P. Bovet, 05].

![Diagram](image.png)

Figure 3.2 The user-space, kernel-space and system call
Sometimes, in certain circumstances, a user application may be in need of certain processing requirement directly from the LINUX kernel (e.g., `wait()` system call), which is not possible to get done through the intermediate layer of programming language or operating system commands; so, it requests the LINUX kernel to get the required work done. Such a request is made by using an appropriate system call (Figure 3.2). The system calls are used when a user space application wants to use certain services which are provided by the LINUX kernel; they appear like ordinary C-functions to the programmer, and they can be directly called in system programs and user-space applications. The number of available system calls varies from kernel to kernel.

The system call gets invoked by the associated unique wrapper function (such as `getpid()` provided by the glibc library; internally each system call is assigned and known by a unique number. As the system call executes code in the kernel mode, when a user space process invokes a system call, the execution mode of the process is changed from user mode to the kernel mode. When a user program issues a system call, it is actually calling a library routine. The library functions execute an assembly instruction "int $0x80" (also known as operating system trap or software generated interrupt) that changes the process execution mode from user mode to kernel mode [W. Mauerer, 08]. It also passes the system call number to the kernel using the `ORIG_AX` register. The arguments (if any) of the system call are also passed to the kernel space using other registers (`EBX`, `ECX`, etc.).

This results into execution (through the interrupt handler) of the `system_call` function. This function handles all system calls, as identified by the contents of `ORIG_AX`. As mentioned above, each system call has its own handler function predefined in the kernel space; the memory addresses of each of these handler functions are
stored in an array named "sys_call_table". So the service routine i.e. the handler function corresponding to the getpid() system gets executed. On completion of the service routine, the control is transferred back to the service routine and then to the user application along with the return values (Figure 3.3).

![Figure 3.3 System call execution flow](image)

**Mechanism for checkpointing the system call’s information**

The figure 3.3 shows that the instruction “INT $0X80” raises a software interrupt 128 (which represents a call gate to which a specific system call handler function is assigned to perform system call execution) and alters the execution mode from the user-mode to the kernel-mode. It indicates that during the time of process state checkpointing, if the process had been in some system call execution, then the instruction pointer IP must just have followed the “INT 0X80” instruction, i.e. the “0xCD 0x80” instruction, and therefore, the previous value of the instruction pointer must be “0xCD 0X80”.
Therefore, the memory address \texttt{IP-2} must hold the code \texttt{0xCD} and the address \texttt{IP-1} must hold the code \texttt{0x80}.

Here we suggest a mechanism along with some of the foremost steps of implementation to obtain the information about a system call which is being currently executed by the desired process in order to carry out process checkpointing and migration [N. A. Joshi, 09]. The mechanism suggested below, is applied after stopping execution of the desired process (through the SIGSTOP signal):

\textbf{BEGIN}

1. Obtain the process control block (PCB) associated with the desired process-id. As discussed earlier in this chapter, the PCB is represented by means of the ‘\texttt{task_struct}’ data structure in LINUX.

   \begin{verbatim}
   struct task_struct* ptr_to_pcb;
   int pid = <<process-id>>;
   /* pid is the process-id of the desired process which is to be migrated. */
   ptr_to_pcb = find_task_by_pid(pid);
   \end{verbatim}

2. Obtain the current values of the processor’s registers-set from the obtained PCB (i.e. \texttt{ptr_to_pcb}) in step 1.

   \begin{verbatim}
   struct pt_regs* ptr_to_registers;
   /* The structure pt_regs is the C-structure representation of microprocessor’s registers-set and it holds the current values of the processor’s registers-set. */
   ptr_to_registers = task_pt_regs(ptr_to_pcb);
   \end{verbatim}

3. Fetch the value of \texttt{orig_ax} register from the register-set (\texttt{ptr_to_registers}) obtained in step 2. The value obtained must be a valid system call number (i.e. between 0 and \texttt{__NR_syscalls}) if the desired process is currently executing
some system call. Otherwise, the process is not executing in system call.

Here, __NR_syscalls is a constant in kernel which holds the total number of available system calls.

```c
if(ptr_to_registers->orig_ax <= 0 ||
    ptr_to_registers->orig_ax > NR_syscalls)
{
    printk(KERN_INFO "The process is not in system call.\n");
...
}
```

4. Obtain the previous instruction code of the instruction pointer IP (i.e. `ptr_to_registers.ip`). For this purpose, we have to obtain the two code values stored at the two memory addresses IP-2 (i.e. `ptr_to_registers.ip-2`) and IP-1 (i.e. `ptr_to_registers.ip-1`) respectively.

In order to obtain these two code values, in this step, we find the corresponding kernel-space addresses for the previous two user-space addresses i.e. `ptr_to_registers.ip-2` and `ptr_to_registers.ip-1` and name them as `ptr_to_code1` and `ptr_to_code2` respectively.

The kernel space address is calculated using the following sub-steps (a) to (e):

(a) Find the address of page global directory of particular address and check for its validity.

```
pgd_t* global_dir = pgd_offset(ptr_to_pcb->mm, address);
```

Here, the address refers to the user-space address.

It will assume the values `ptr_to_registers.ip-2` and `ptr_to_registers.ip-1` respectively.

(b) Find the address of page upper directory from the global address and check for its validity.
pud_t* upper_dir = pgd_offset(global_dir, address);
...
(c) Find the address of page middle directory from the upper address and check for its validity.
pmd_t* middle_dir = pmd_offset(upper_dir, address);
...
(d) Calculate the address of page table entry from the middle address and check for its validity.
pte_t* pg_table_entry = pte_offset_map(middle_dir, address);
...
(e) Calculate the actual kernel address using page table entry.
...
char* kernel_space_address = (char *)
    page_address(pte_page(*pg_table_entry))
    + (address & ~PAGE_MASK);

Here, the kernel_space_address is the desired resultant address in kernel space which will be referred to as code1 and code2 respectively now onwards.

5. Obtain the two code values stored at the addresses ptr_to_code1 and ptr_to_code2 obtained in step 4 and name them as code1 and code2 respectively.
if(ptr_to_code1)
    int code1 = * ptr_to_code1;
if(ptr_to_code2)
    int code2 = * ptr_to_code2;

6. Compare the two code values obtained in the step 5 with the constants 0xCD and 0x80 respectively. If they match then it indicates that the desired process is in a system call.
if((code1 == (char)0XCD) && (code2 == (char)0X80))
{  
    ...  
    printk(KERN_INFO "The process is executing a  
    system call\n");  
}  

7. And the value obtained in step-3 from the register orig_ax is  
the system call number, which the desired process is executing.  
printk(KERN_INFO "The process is executing the \%d  
    system call.\n", ptr_to_registers ->orig_ax);  

8. Reset the instruction pointer IP in the process control block to its previous instruction. i.e.  
ptr_to_registers.ip -= 2;  
/* And set the AX register in the process control block i.e.  
ptr_to_registers to the appropriate value. */  

9. Append this updated register set (ptr_to_registers) to the  
external checkpoint-image file which is to be migrated to the destination workstation.  
END  

Then, the updated register set along with the checkpoint-image file will be migrated (through process migration mechanism) to the destination workstation. After migrating to the destination workstation, the migrated process will resume execution right from the instruction where it was stopped, checkpointed and migrated from its originating workstation; i.e. if the process was in the system call during the checkpointing mechanism, then the migrated process will resume execution of the system call on the destination workstation after completion of the process migration operation.
3.6 Concluding Remarks

This chapter focuses on major challenging issues arising during migration of a process that is currently in the state of execution of some system call. The chapter presents significant guidelines to resume partially executed system call on a destination workstation after process migration. We have made attempts to present a solution in the form of a novel process migration tool, which we call OptiMigrator.

In the beginning of this chapter, we have emphasized upon the OptiMigrator’s design characteristics such as – transparency, consistency, optimality, flexibility, efficiency and responsiveness. The chapter also talks about the concept of process and the process descriptor by highlighting the important data members of the data structure task_struct. Moreover, the chapter introduces the sub-modules of OptiMigrator, which we call – ProcessMigrator, LoadBalancer and LoadPortal – and gives outline about their interactions.

Finally, the chapter describes a mechanism and its implementation which helps the migrated process to resume the execution of a system call on the destination workstation.