IMPLEMENTATION OF PROCESS FORENSIC FOR SYSTEM CALLS

Dr. Narayan A. Joshi
Associate Professor, CSE Department, NIRMA University, Ahmedabad, Gujarat, India.

Dr. D. B. Choksi
Professor, Department of Computer Science, Sardar Patel University, Vallabhbh Vidyanagar, Gujarat, India.

ABSTRACT

Mechanisms such as dynamic process migration, process forensic, process checkpoint-restart are concerned with process-state checkpointing. Dynamic process migration mechanism migrates a partially executed process from its originating workstation to other workstation, thereby it supports load balancing in distributed & networked systems letting applications to access computing resources spread across the network; such workload relocation among workstations may result into substantial improvements in overall system performance. The process checkpointing mechanism is responsible for identifying current dynamic and static state information of a desired process. However, determination of information of a particular system call under execution by a process to be checkpointed is one of the challenging tasks for the process checkpointing and migration mechanisms. A kernel-level solution for checkpointing the system call details is described in this paper.

Keywords: System Call, Process Forensic, Process Checkpointing, Process Migration.

1. INTRODUCTION

From an application developer’s perspective, the LINUX kernel appears to be a transparent layer in the operating system environment. In general, the presence of LINUX kernel is not noticed, however it always exists and continuously serves its best. In order to carry out assigned computing tasks, user-level processes are not required to know about which regions of memory and region address-ranges are occupied by them, which region-pages are swapped to the disk, though the processes remain in nearly continuous interaction with the underlying kernel to avail required services and thereby to get their work done.
For the sake of getting the assigned work done, the user-level processes take services provided by library routines. The library routines, in turn call the kernel-level functions. The role of standard library routines becomes significant as they act as an intermediate layer between the user-space process and the underlying kernel. Moreover, they help to standardize and simplify the distribution and usage of kernel level routines across diversified kernels.

In order to make its capabilities and routines available to the user-level processes, the kernel facilitates system calls through which the kernel-level functions can be accessed in the user-space. A user-level process is an instance of a binary executable program file under execution. Every process has a certain unique state comprising of static and dynamic components. The static state of a process comprises of the binary executable program file itself; whereas the dynamic process-state comprises of the process’s address space (both of physical and virtual memory), CPU registers (both of user-space and kernel-space), I/O devices, files opened, sockets opened, pipes opened, signals received, blocked and pending, and many other things including system call under execution (if any) [2].

Sometimes, it may be required by the administrator, application developer, operating system kernel or system programs to determine the complete state information of a desired process. The paper describes a mechanism to determine system call information being executed by a desired process. Section 2 describes motivating scenarios concerned with determination of dynamic state information of a process. The problem statement and scope of the paper is in mentioned section 3. Proposed solution is discussed in section 4. The paper concludes with remarks and future directions in section 5.

2. MOTIVATION

As seen in the above section, process state comprises of static and dynamic state components of a process. The requirement of process state identification and checkpointing becomes substantial during many situations, some of which are listed here [9].

Sometimes, it may be desired to pause the execution of a partially executed process, preserve its current state, kill the process, and later on according to user’s convenience and needs, restore the preserved state in a new process, and resume the restored process for execution. Such a process checkpoint-restart mechanism becomes advantageous for process migration also [3].

Process migration and load balancing can be applied to a system of interconnected computers to distribute the load of overloaded workstations among idle or lightly-loaded workstations [1]. The load balancing activity may result into utilization of a plenty of distributed computing power that usually goes unused. Instead of letting some machines be overloaded with several tasks, it makes sense to relocate some of these tasks to under-utilized machines from the overloaded ones. This allows such tasks to complete quickly and improve the overall system performance [6], [7].

Another area of applications where process migration could be beneficial is- it may be used to relocate a distant process closer to the big data that it is processing, thereby ensuring that it spends most of its time in doing useful work; and not wasting its time performing communication between nodes for sake of accessing the required data.

Moreover, process migration may be beneficial in achieving fault tolerance. A process could be migrated from a partially failed workstation to some other healthy workstation in the network to avail process continuity. The same can be made applicable to the long-running processes whose workstations are to be disconnected from the network or turned-off later after some time because of lack of battery backup. Thus, the process migration can make the process available even after machine-failure or disconnection [4].

Furthermore, certain machines may possess better capabilities such as higher computing power, more amount of primary memory, availability of graphical processing unit, etc. which can be
utilized by moving a needy process to that machine. In this case, the process can be started on the source machine and then migrated to a more powerful machine later when it becomes available.

3. PROBLEM

The operating system maintains the dynamic state information of a process in a kernel-level process descriptor data structure task_struct. The LINUX process control block task_struct maintains process state information such as process id, process group id, address space, current execution state, memory regions, credentials, opened files information and many other details including process scheduling and management [8]. Being such information available in the process descriptor and other supporting kernel-level data structures, various mechanisms such as process forensic, process checkpointing and process migration may carry out their tasks. However, at the time of process state investigation, checkpointing and migration, chances are there that the process could be running in some kernel-space function such as some system call handler. However, such mechanisms need to determine the related information of the system call currently being executed by the desired process. Moreover, the kernel doesn’t maintain such information in the process descriptor and other supporting kernel-level data structures. Furthermore, the kernel doesn’t provide a kernel-level mechanism to determine whether the process is running in some system call handler or not.

It is desirable to avail kernel-level mechanism with which one can determine whether a process is running in some system call or not, and if the process is running some system call, then the related details about the system call should also be made available by the said mechanism. We have already proposed a partial solution overview of some of the algorithmic steps for the above described problem in our paper [5]. In this paper we have provided a comprehensive solution algorithm for the above said problem. Furthermore, a working implementation mechanism is also presented in this paper.

4. MECHANISM

We describe here the working of system calls first. For simplicity, we have considered here the open source operating system and the freely available kernel.

In order to provide its various services including efficiency, security and consistency to the user-space programs, the kernel facilitates a very powerful tool in the form of system calls. The user-space programs can avail the privileged kernel-level facilities and resources by means of system calls which normally appear as like as ordinary functions. Therefore, the system calls are the gateways for user-space programs to penetrate into the kernel space to avail its services. The system makes the system calls available to the user-space by means of software interrupts.

While the user-space program is running in user-space and invocation of system call causes a maskable interrupt or exception class transfer from user-mode to the kernel-mode; which happens as a result of execution of the instruction int 0x80. Here, the vector 0x80 is used for transferring the control from user mode to the kernel mode by means of software interrupt. The in memory data structure interrupt vector gets initialized during system boot time. Further, the number of system call is required to be communicated to the kernel-space which normally is achieved by means of setting the orig_ax register with the appropriate system call number; sometimes it also depends on the underlying architecture and the version of the kernel.

The execution flow of system calls is shown in figure 1[5]. As described above, the instruction int 0x80, is responsible to raise a software interrupt 128. It represents a call gate to the kernel’s specific service, thereby causing execution mode switching from user-mode to the kernel-mode. In the kernel-space, for the interrupt 128, a specific handle routine is registered to carry out execution of the requested system call. It suggests that during the time of process forensic or
migration, if the process was passing through the stage of execution of some system call, then the CPU would just have executed the instruction 0xcd 0x80 before jumping to the kernel space, and therefore, the previous value of the instruction pointer in the user-space would point to the corresponding mode switching instruction.

<table>
<thead>
<tr>
<th>User mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
</tbody>
</table>
|   int age;
|   scanf("%d",&age); |
| }         |
| libc      |
| %orig_ax=sys_read; |
| int 0x80   |
| library   |
| kernel mode |
| 0x80       |
| IDT        |
| system_call() |
| {          |
|   fn=syscalls[%orig_ax]; |
| }         |
| syscalls table |

![Figure 1: The system call mechanism](image)

Here we suggest a mechanism for the steps suggested in [5]:

1. Obtain the process descriptor of the desired process whose forensic-investigation is to be done, or which is to be migrated.
2. Determine the current state of the process’s register-context from the obtained process control block.
3. Fetch the value of orig_ax register of the desired process.
   The value obtained may represent some system call. Determine the validity of the system call number. In case of a valid system call number, the process could be running a system call, otherwise the process was not doing a system call.
4. Obtain the previous instruction code of the instruction pointer in order to determine the two code values at the two memory addresses ptr_to_registers.ip-2 and ptr_to_registers.ip-1.
   a. Calculate the address of page global directory and verify its in memory validity.
   b. Calculate the address of the page upper directory from the global directory address and verify its in memory validity.
   c. Determine the page middle directory and verify its in memory validity.
   d. Determine the page table entry and verify its in memory validity.
   e. Swap in the required page if it is currently not available in memory.
5. Determine the addresses for the ptr_to_registers.ip-2 and ptr_to_registers.ip-1.
6. Fault in the page(s) containing the physical addresses determined in step 5 and obtain the values stored in those two locations.
7. Determine the nature of instruction from the values fetched in above step-6 to verify whether the process is running in some system call or not.
Some of the foremost implementation steps for the function system_call_forensic( ) are shown below [10], [11]:

```c
#include<linux/pagemap.h>
#include<linux/sched.h>
#include<linux/module.h>
#include<linux/kernel.h>
...

int system_call_forensic(pid_t pid)
{
    /*pid is process-id of the desired process which is to be migrated or whose forensic investigation is to be done*/

    struct task_struct* pcb; //represents the process control
        //block
    struct pt_regs* regs;   //represents the register context
    unsigned long k_addr;  // represents kernel physical address
    pcb = find_task_by_pid(pid); //fetch pcb
...
    regs = task_pt_regs(pcb);

    k_addr = regs.ip;
    if(*regs).orig_ax <= 0 || (*regs).orig_ax >= NR_syscalls)
    {
        printk(KERN_DEBUG "The process is not running system call\n");
        ...
        return 0;
    }

    //obtain the addresses of directory levels and verify their validity
    pgd_t* globaldir = pgd_offset(pcb->mm, k_addr);
    ...
    pud_t* upperdir = pud_offset(globaldir, k_addr);
    ...
    pmd_t* middledir = pmd_offset(upperdir, k_addr);
    ...
    pte_t* pte = pte_offset_map(middledir, k_addr);
    ...

    /*determine the presence of the page in memory, otherwise fault in the same.*/
    if(!pte_present(*pte))
    {
        //lock the table
        spin_lock(page_table_lock);
        //fault in the page
        ...
        spin_unlock(page_table_lock);
    }
```
char* physical_address = page_address(pte_page(*pte)) +
    k_addr & ~PAGE_MASK;

pte_unmap(pte);

if the type of instruction stored at physical address is
associated with software interrupt
    return (*regs).orig_ax;
else
    return -1;
}

5. CONCLUDING REMARKS

A working implementation by means of a kernel module is suggested here to determine
whether a desired process is running in some system call or not. The suggested mechanism returns
the system call number if the process is running in a system call. The suggested mechanism may be
enhanced further to work with the modern approach of system call gateway i.e. sysenter. Moreover,
the suggested mechanism involves traversal through page directories at many levels; which in future
may be optimized further to obtain better performance.

REFERENCES

   direction forward in checkpoint/restart implementations for fault tolerance”, Proceedings of 19th
   IEEE International Symposium on Parallel and Distributed Processing.
   Platform”, International Journal of Computer Applications in Engineering, Technology &
   Sciences.
   Journal of Advanced Research in Engineering & Technology, ISSN Print: 0976-6480,
   34343-2, 2008.
    Systems”, International Journal of Advanced Research in Engineering & Technology,
    ISSN Print: 0976-6480, ISSN Online: 0976-6499, Vol. 5, Issue 5.