International Journal of Hybrid Information Technology
Vol.8, No.10 (2015), pp.319-330
http://dx.doi.org/10.14257/ijhit.2015.8.10.29

Design of a Loadable Kernel Module based Process Performance Analyzer

Barun Kumar Parichha

Juniper Networks India R&D Lab,
Bangalore, INDIA 560 075
barun.parichha@gmail.com

Abstract

Performance analysis of process plays a significant role in improving the overall efficiency of any system. Usually, this task is accomplished either by system level commands or user space applications, based on procfs or sysfs file system. There are many creative techniques and popular commercial applications available to perform this management and fine tuning of process. But, these existing user space based mechanisms are limited in scope and often fail to provide the required process specific data to user. In order to avoid this limitation, we have proposed a new linux kernel module (LKM) based approach, which can be potentially used to track any process specific data in real time. In this approach, the user does not require any separate tool but the kernel module to gather all essential information. Further, the competence of this module based technique can be enhanced by incorporating additional functionalities and thereby making it a full-fledged multi process abstraction layer to cater the needs of any high performance embedded platform.

Index Terms: Performance Analysis, Kernel Module, System Security, Process, Device Driver

1. INTRODUCTION

LINUX is a multiuser, multitasking and multiprocessing operating system. It is fast, consistent, reliable and protects the users from problems related to viruses and other malwares. This OS is ubiquitous in almost all hardware platforms and architectures, ranging from small embedded boards, mobile handsets to sophisticated mainframes and Mars rovers [13]. Being an open-source [1], it has become the default platform for many developers and researchers. Linux has no license fees for most softwares unlike other operating systems. It provides support for all major file systems. It's ease of installation, upgradation, and facility of accessing different software components through central repositories makes this popular and user friendly. Major proprietary Windows softwares can also be used in Linux using the softwares like *wine* and *virtualbox*. Moreover there are many global developer communities, providing instant support for any Linux related issues.

With the increase in popularity [2] across different segments of society, it is crucial and challenging task to know the various system parameters of interest. There are various commands or applications available to know the system statistics of a process, some of them are mentioned in Table 1. The common idea behind these commands is that they read the existing *procfs* or *sysfs* to collect the required statistics and then present them in a user friendly way.

ISSN: 1738-9968 IJHIT

Table 1. Linux Performance Analysis Commands

Sl. NO	COMMAND	USAGE
1	top	cpu, memory usage of processes.
2	vmstat	virtual memory statistics of process.
3	lsof	list of open files by process.
4	tcpdump	network statistics
5	iptraf	real-time network statistics
6	netstat	list of open ports
7	strace	system call used by process
8	ps	process statistics
9	pmap	process memory usage
10	sar	collect system activities
11	wireshark	analyze network packets captured
12	ethereal	shows live network statistics

We observed several applications for process monitoring [3, 4, 7-9], and various methodologies for measuring process performance [10-12]. Some of those applications are good for network monitoring, some are for memory monitoring and some good for others. But hardly any single application was found, giving all real time process information needed by user on request. To overcome this limitation, we have proposed a generalized LKM based framework, which provides all process related information to user on the fly. This approach is potentially self-sufficient to give all process statistics, under a common kernel module interface. Moreover it facilitates to incorporate more features based on user and industry requirements.

The rest of the paper is organized as follows: First we give a short introduction to device driver and kernel in linux. Then we introduce our LKM based process monitoring architecture with results. Finally we conclude with future work and references.

2. DEVICE DRIVERS IN LINUX

In a linux machine, kernel is the heart of the whole system. It is a monolithic [5] type, with a very large and complex body of code, consisting of many device drivers, making a particular piece of hardware respond to a well defined internal programming interface, hiding the details of how the device works [6]. User space application talks to kernel through a set of system calls, which are mapped to device specific operations to act on the real hardware.

To a programmer, system calls are like other ordinary functions, but in reality this involves switching from user space to kernel space. These system calls behave as function level abstraction for the underlying hardware, hiding the detail level control flow. Everything in linux is a file, so the users just need to call the same API, no matter whether to access some hardware component or some normal text file. For detail understanding, one has to refer linux *virtual file system*(VFS) layer design and data structures in kernel. The VFS layer presents hardware as a collection file systems types in a hierarchical manner. The actual hardware elements are presented to user space through some device nodes under */dev* with major and minor numbers included.

Basically kernel works in two modes called the *process context* and the *interrupt context*. Applications running in the system, use application specific APIs to accomplish the task. These APIs in turn call a set of system calls which uses the kernel on behalf of application process. In this mode, the kernel is said to be run in *process context*. Kernel also manages all the system hardware. When any part of hardware needs kernel

attention, it raises some hardware interrupts. These hardware interrupts in turn interrupts the kernel. In this mode kernel is said to be run in *interrupt context*. To provide synchronization across the whole system, kernel may disable some or all interrupts, till the actual interrupt processing gets completed. All the interrupts are mapped to some predefined interrupt number inside the interrupt Table. Entries in this table map to *interrupt servicing routine* (ISR), which performs the real operation on behalf of process or hardware.

Each process running in user space is represented in kernel by *struct task_struct*, which contains all information relevant to the process. Process often run with multiple threads in a multithreaded program. But in kernel all these threads are represented as independent *task struct*. Each of these threads include unique program counter, process stack and set of processor registers.

Life of a process starts with system call *fork()*, which creates a child process. The new child process is an exact copy of the parent process, but with unique child PID (process identifier). Often, after a *fork()* we require to run a new program. The *exec()* series of system calls are used for this purpose. In kernel, *fork()* is implemented using system call *clone()*. The program life cycle ends with system call *exit()*. The parent process can enquire the status of the terminated child process using *wait()* system call.

The kernel stores the list of all processes in a single circular liked list called the *task list*. Each element of this *task list* is a process specific descriptor of type *struct task_struct*. This *struct* is a very complex data structure having all OS parameters and data, kept under clean, well defined structure boundary. Our process monitoring system uses this kernel data structure to obtain the required data needed in monitoring the process.

3. LINUX REAL-TIME PROCESS MONITOR IMPLEMENTATION

Our process monitoring is broadly classified into two sections, the user space data collection and the kernel space module. A character device node *(/dev/myNode)* is used to communicate between these two units. The user space application opens the character node and passes appropriate request string to get the required information for the process.

3.1 User space Data Collection System

Several process parameters can be collected by our design, but for the simplicity and ease of illustration, we restrict to following list of process statistics in this work and depicted the procedure to retrieve them.

```
void show_opt ( )  {
      printf ( ” 1 . process name
                  2 . group id
                  3 . parent process
                  4 . process group leader
                  5 . child processes created
                  6 . process memory segments
                  7 . virtual memory mapping
                  8 . process priority
                  9 . process state
                 10 . cpu used by process
                 11 . total fault count of process
                 12 . process start time
                 13 . process link count  \n ” ) ;
     printf (” Enter Option : ” ) ;
```

```
}
```

Our user space code is based on C. As discussed before, the main body of the source code requests the LKM to get data through the character device node */dev/myNode*. We pass process id (PID) and request token as character string to module. This module checks our passed token and passes the result to user space application through the same buffer *buf* passed. Here we are not using any separate read system call after write to make it simple and faster.

```
int main ( )  {
   char buf [700] , opt [50] ;
   int  rt ;

   //open device node of LKM
   int fd=open(”/dev/myNode” ,O_RDWR) ;
   printf (”Enter Process PID : ” ) ;
   scanf (”%s”, buf ) ;

   //get option
   show_opt( ) ; scanf (”%s ”, opt ) ;

   //pass message string to LKM
   sprintf (buf ,”%s %s ”, buf , opt) ;
   rt = write (fd , buf , strlen(buf) ) ;

   //print result
   if ( rt ==1)
      printf (”%s ”, buf ) ;

   close(fd) ;
   return 0 ;
}
```

3.2 Kernel Module Code

3.2.1 Registering Char Driver

In our kernel code, first we register a dynamic character device driver and create the

device node */dev/myNode* in user space, through which user space application talks to module.

```
static struct class  * cl ;
static struct cdev  * cdev ;
dev_t dev ;
struct semaphore mysem;
static int start( )
{
    printk (KERN_ALERT  ”Initializing Module ” ) ;

   //get major no
   alloc_chrdev_region (&dev ,0 ,1 ,”myNode” ) ;

   //regisiter device
   cdev = cdev_alloc ( ) ;
   cdev→ops = &fops ;
   cdev→ owner = THIS_MODULE;
   cdev_add (cdev ,dev ,1) ;

   //creating device node
   cl = class_create (THIS_MODULE, ”myNode ” ) ;
   device_create (cl , NULL, cdev→dev , NULL, ”myNode ” ) ;

   //initialize semaphore
   sema_init (&mysem, 1) ;
   return 0 ;
}
```

Semaphore *mysem* is used here for synchronization from simultaneous access to module. As per the design, the module services single request at a time, while other requests have to wait before getting access. However this singleton approach can be changed easily with minor code changes for simultaneous access from multiple user applications.

```
static void leave( )
{
   printk (KERN_ALERT, ”Leaving Module ” ) ;
   device_destroy (cl, dev) ;
   class_destroy (cl) ;
   cdev_del (cdev) ;
   unregister_chrdev_region (cdev→dev , 1) ;
}
module init ( s t a r t ) ;
module exit ( leave ) ;
```

3.2.2 Driver Interface for User Space

All access to driver module like *open, write, close* to */dev/myNode* occurs through kernel interface *struct file_operations*. Semaphore *mysem* is used for singleton access to module.

```
struct file_operations fops = {
    . owner = THIS_MODULE,
    . open = myopen ,
    . release = myrelease ,
    . write = mywrite ,
};
```

```
int myopen (struct inode  * inode ,  struct file  * file )
{
   //block from simultaneous access
   down_interruptible (&mysem) ;
   return 0 ;
}

int myrelease (struct inode * inode , struct file   * file )
{
   //release semaphore
   up(&mysem) ;
   return 0 ;
}
```

```
#define MAX 700
static char mybuff [MAX] = { 0 } ;

int mywrite (struct  file  * file , char * buff, size_t count, loff_t  * pos)
{
   int val , rpid ;
   char *  to, * val ;

   //copy buffer received
   memset (mybuff, 0, 100) ;
   strncpy (mybuff , buff , count ) ;

   //get PID and OPTION passed
   to = &mybuff [0] ;
   val = strsep (&to , ” ” ) ;
   rpid = simple_strtol (val, NULL, 10) ;
   val = strsep (&to, ” ”) ;
   opt = simple_strtol (val ,NULL, 10) ;

   return ret ;
}
```

3.2.3 Main logic to process query

Inside kernel handler *mywrite* we find *task_struct* of process, whose PID is passed. For more detail and implementation of *task_struct* refer linux documentation [1, 5, 6].

```
//find the task_struct
for_each_process(task) {
   //compare process pid
```



```
    if (rpid == task->pid) {
        ret =1;
        break;
    }
}
```

Then we find out the information needed based on the option value received from user space. The kernel logic to evaluate various process information is shown below, but for simplicity detail code snapshot is omitted in this literature.

Case 1: Finding Process Name

```
strncpy(buff,task->comm, strlen(task->comm));
```

Case 2: Finding Group Id

```
snprintf(buff,sizeof(buff), "%d ",task->tgid);
```

Case 3: Finding Parent Process

```
snprintf(buff,MAX, "Process:%s with Pid=%d ",
            task->parent->comm,
            task->parent->pid);
```

Case 4: Finding Group Leader

```
snprintf(buff,MAX, "Process:%s with Pid=%d ",
            task->group_leader->comm,
            task->group_leader->pid);
```

Case 5: Finding the List of Child Processes Created

```
list_for_each(trav, &(task->children)){
    t = list_entry(trav, struct task_struct, sibling);
    count = strlen(buff);
    snprintf(buff+ count, MAX, "%d ", t->pid);
}
```

Case 6: Finding Process Memory Segments

```
snprintf (buff,MAX,
            "Code Segment: 0x%lx - 0x%lx \n
              Date Segment: 0x%lx - 0x%lx \n
              Heap Segment: 0x%lx \n
              Stack Segment: 0x%lx\n",
              task->mm->start_code,
              task->mm->end_code,
              task->mm->start_data,
              task->mm->end_data,
              task->mm->start_brk,
              task->mm->start_stack);
```

Case 7: Finding Process Virtual Memory Mapping

```
snprintf(buff, MAX, "\nTotal no of vmas = %d\n", task->mm->map_count);
for (vma = task->mm->mmap; vma; vma = vma->vm_next) {
   count = strlen(buff);
   snprintf(buff+ count, MAX, "0x%lx - 0x%lx\n", vma->vm_start, vma->vm_end);
}
```

Case 8: Finding Process Priority

```
snprintf(buff,MAX,
            "Static Priority(nice)=%d \n
             Dynamic Priority=%d \n
            Normal Priority=%d \n",
            task->prio,
            task->static_prio,
            task->normal_prio);
```

Case 9: Finding Process Scheduling Policy

```
if(task->policy == TASK_RUNNING)
   snprintf(buff,MAX, "TASK_RUNNING");
else if(task->policy == TASK_INTERRUPTIBLE)
   snprintf(buff,MAX, "TASK_INTERRUPTIBLE");
else if(task->policy == TASK_UNINTERRUPTIBLE)
   snprintf(buff,MAX, "TASK_UNINTERRUPTIBLE");
else if(task->policy == TASK_STOPPED)
   snprintf(buff,MAX, "TASK_STOPPED");
else if(task->policy == EXIT_ZOMBIE)
   snprintf(buff,MAX, "TASK_ZOMBIE");
```

Case 10: Finding Cpu Used

```
snprintf(buff,MAX,"\nCPU used = %d\n", task_cpu(task));
```

Case 11: Finding Major and Minor Faults

```
snprintf(buff,MAX,
          "No of Major faults = %d \n
           No of Minor faults = %d \n",
           task->maj_flt,
           task->min_flt);
```

Case 12: Finding Process Start Time

```
snprintf(buff,MAX,"TIME: %.2lu:%.2lu:%.2lu:%.6lu \r\n",
                          (task->start_time.tv_sec / 3600) % (24),
                          (task->start_time.tv_sec / 60) % (60),
                          task->start_time.tv_sec % 60,
                          task->start_time.tv_nsec / 1000);
```

Case 13: Finding Process Link Count

```
snprintf(buff,MAX,"\nLink count = %d \n", task->link_count);
```

3.2.3 Kernel Makefile

Given below the kernel makefile in it's simplest form to build the loadable kernel module. This Makefile builds testLKM.c and generates module testLKM.ko.


```
obj-m += test.o
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

4. RESULTS

After the module gets inserted, the user is expected to get process information of any arbitrary process of interest, which is shown as an example for PID 16526.

```
narmada$ sudo insmod testLKM.ko
narmada$ sudo ./a.out

Enter Process PID : 16526

1. process name
2. group id
3. parent process
4. group leader
5. child processes
6. process memory segments
7. process virtual memory mapping
8. process priority
9. process state
10. cpu used
11. No. of major and minor faults
12. start time
13. link count

Enter Options : 1
  gnome-pty-helper

Enter Options : 6
   Code Segment : 0x400000 -  0x4027c4
   Date Segment : 0x602e18 -  0x603200
   Heap Segment : 0xb68000
   Stack Segment : 0x7fff9f6112e0

Enter Options : 7
   Total no of vmas = 5

   0x7f3ca70f9000 -  0x7f3ca7105000
   0x7f3ca7105000 -  0x7f3ca7304000
   0x7f3ca7304000 - 0x7f3ca7305000
   0x7f3ca7305000 -  0x7f3ca7306000
   0x7f3ca7306000 -  0x7f3ca74bb000

Enter Options : 11
   No of Major f a u l t s =0
   No of Minor f a u l t s =745
```

Being implemented entirely in kernel space, this approach is a simple and low-overhead process for the performance evaluation of a system. In comparison with other approaches, it takes less time to retrieve the data. Here, one just need to call a single system-call from userspace application. Based on the requirement, further features of interest can be added to this approach and hence can be incorporated in any high or middle level languages.

5. EVALUATION

To validate our design, we selected a system having 2.5GHz, 64-bit, i5-processor with 4GB RAM, running Ubuntu12 Linux distribution with kernel version-3.13. We measured the overall time required to capture the various process parameters and it's contribution towards the overall system load. The time was measured in microseconds, using libc call *gettimeofday*. We called *gettimeofday* before passing the request string to kernel and again immediately after retrieving the result. The difference between both time values, gives the time required to capture the process parameters. Table 2 shows this final result in microseconds for all the parameters collected.

Table 2. Time Taken to Get Various Process Parameters

Sl. No.	Parameters	Time(in microseconds)
1	process name	82
2	group id	120
3	parent process	71
4	group leader	101
5	child process	147
6	memory segments	110
7	process priority	92
8	process state	83
9	cpu used	83
10	major and minor faults	96
11	start time	94
12	link count	83

The time taken for most of the parameters was found to be less than 100 microseconds, except for retrieving the list of child processes, because it involves lookup in multiple process task entries. This approach's contribution on system load was found to be negligible as it's entire logic is in kernel space. Hence, we did not include this data in this paper.

6. FUTURE WORK

This paper depicts the procedure to collect some simple process specific data without depending on any user space application or file system. However, many advance features can be included in the later versions as a future work. Some of them are mentioned below.

1. Client server based multisystem single server data collection approach to be added.
2. Changing the process parameters through this kernel interface.
3. System alarm implementation, when process parameters exceeds threshold.
4. Finding of different benchmarks, such as top 10 processes thrashing and, occupying more memory *etc*.
5. Including these new important process parameters to *proc* file system, so that some

script can process them in hourly, daily and monthly basis.

7. CONCLUSION

Our LKM based process monitoring system has the potential to provide user, the total health of any process, running in the system. Understanding of these real time system parameters, can effectively identify the processes, which are malfunctioning and need immediate user attention. It ensures the safety of the whole system, from the analysis of kernel level data collected. Further, it can be used as a potential solution for advance detection and poisoning of the machine from the applications running. Numerous features can be added to this procedure to extend it's functionalities and there by making it more applicable for a full fledged monitoring tool.

REFERENCES

[1] K. Source, "The Linux Kernel Archives", https://www.kernel.org/.
[2] Wikipedia, "Usage share of operating systems", **(2014)** August 25, http://www.netmarketshare.com/operating-systemmarket-share.aspx.
[3] A. Kutlu and T. Aydogan, "Performance Analysis of MicroNet: A Higher Layer Protocol for Multiuser Remote Laboratory", IEEE Transactions on Industrial Electronics, vol. 56, **(2009)** December, pp. 4784-4790.
[4] L. Bello, O. Mirabella and A. Raucea, "Design and Implementation of an Educational Testbed for Experiencing With Industrial Communication Networks", IEEE Transactions on Industrial Electronics, vol. 54, **(2007)** December, pp. 3122-3133.
[5] R. Love, Linux Kernel Development, Novell Press, **(2006)**.
[6] A. Rubini and J. Corbet, Linux Device Drivers, 3rd ed, O'Reilly Media, **(2005)**.
[7] J. Zhang and K. Chen, "Performance analysis towards a KVM-Based embedded real-time virtualization architecture", IEEE Transactions on Computer Sciences and Convergence Information Technology, **(2010)** November, pp. 421-426.
[8] R. Ranokphanuwat and S. Kittitornkun, "Performance analysis and improvement of SNPHAP on Multi-core CPUs", IEEE Transactions on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), **(2013)** May, pp. 1-6.
[9] A. Ansari, A. Chattopadhayay and S. Das, "A Kernel Level VFS Logger for Building Efficient File System Intrusion Detection System", IEEE Transactions on Computer and Network Technology (ICCNT), **(2010)**, pp. 273-279.
[10] V. V. Rubanov and E. A. Shatokhin, "Runtime Verification of Linux Kernel Modules Based on Call Interception", IEEE Transactions on Software Testing, Verification and Validation (ICST), **(2011)**, pp. 180-189.
[11] M. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study", IEEE Transaction on Software Maintenance, **(2000)** October, pp. 131-142.
[12] M. M. Lehman, "Laws of Software Evolution Revisited", Proceedings of the 5th European Workshop on Software Process Technology, **(1996)**, pp. 108-124.
[13] F. Hartman and S. Maxwell, "Driving the Mars Rovers", http://www.linuxjournal.com/article/7570/.

Author

Barun Kumar Parichha, received his B.E. degree in computer science and engineering in 2003 and the M.S. degree in computer science and engineering from Indian Institute of Technolgy, Madras, India, in 2010. He is currently working as a software engineer in JunOS kernel team for Juniper Networks R&D center, Bangalore, India. From 2011 to 2012, he was a Research Assistant in the Indian Institute of Science, Bangalore, India. His research interest includes high performance computing, OS customization, network performance improvement and embedded engineering, using efficient hardware and software modelling and reorganization.

The author can be contacted at his personal mail-id

barun.parichha@gmail.com or his facebook account.

