

“Oh! You need to jump start, and you need to be safe too... Otherwise you will meet with an accident.” If you consider that a warning from a coach to a pupil, it isn't all that wrong. Because it is a real-life situation! Then, what about software experts who design operating systems based on real-time situations?

Coming to the development of an OS, we know it can be classified as a general-purpose operating system (GPOS) and a real-time operating system (RTOS).

As the name indicates, a GPOS is aimed at desktop and/or server applications, wherein it is involved with time lines, schedules or real-life tasks. It deals with simple computational problems like mathematical manipulations or computations that can be handled by the

Arithmetic Logic Unit (ALU) of the memory unit. It is said to be the same with the execution of other user processes as well.

Imagine a scenario in which your real-time machine begins counting down to a nuclear explosion-like situation and, suddenly, you need to monitor events on a large scale to handle critical situations within timelines. And we certainly need to look at deadlines. Apart from carrying out such a real-life task, you need to work on privileges and carry out tasks. Here, such services get the privilege of working at the kernel level.

Most real-life situations are such that the user needs to meet demands and deadlines—you need to respond to real-life situations. In a democracy you can't satisfy everybody. Sometimes, autocratic attitudes creep into a democratic set-up. Your democracy could be

# Enhance Performance by Running User Apps in Kernel Mode

In this article we learn the ins and outs of the Kernel Mode Linux (KML), a kernel patch that enables user programs to be executed as user processes, but with the privilege level of the kernel mode. The benefit? An often drastic performance enhancement of the application.



in danger due to such infiltrators and might go for a toss. Real-time systems need to equip themselves for similar situations. It is to be noted here that an operating system like Linux provides higher privileges to the kernel-level processes, while user processes are run with lower privileges. It's like a ruler and his people! If the people have to exercise higher tasks, they have to be given privileges.

To protect the kernel from user applications, the CPU executes user applications in user mode. And if they want to access kernel data structure, it will be executed in the kernel mode. Even though switching between user mode to kernel mode protects the kernel from crashing, there is an overhead in this switch that affects the performance of the system. Suppose we have a mechanism to execute user applications in the kernel mode directly and safely, we can eliminate the user-to-kernel mode or kernel-to-user mode-switching overhead.

Kernel Mode Linux (KML) has been developed in recent years with these features in mind. In this article we will give you an overview of its internals: how to use it in different system environments, the interrupt handling mechanism and problems of stack starvation; and we will also experiment to evaluate the performance enhancements while executing system calls in kernel mode, providing a brief analysis to support the work that is carried out. This article will also cover the building of the Kernel Mode Linux.

## Internals of KML

IA-32 architectures are based on a memory model that associates different areas of memory, called segments, with different usages (Figure 1). The code segment (CS) holds the instruction of a program. The stack segment (SS) contains the processor stack, and four data segments (DS, ES, FS, GS) are provided for holding data operands.

The privilege level of the code segment is determined by its segment descriptor. A segment descriptor has a field for specifying the privilege level of the segment. The IA-32 architecture uses a virtual memory management-based flat memory model that has segments like CS, DS, SS and FS.

Generally, the Linux kernel sets up two segments—one meant for kernel use and the other for ordinary user processes whose privilege level is at the user mode. The kernel mode has the segment set at 0xC0000000 as the segment descriptor. In order to execute a user process in the kernel mode, KML sets the CS register of the user process to the kernel mode segment. All system calls of programs executed under special protection of the "/trusted" directory execute them in the kernel mode. Shown below is a simple code to check whether the Kernel Mode Linux is working fine in /trusted, kernel source or user mode:

```
#include <stdio.h>
main () {
```

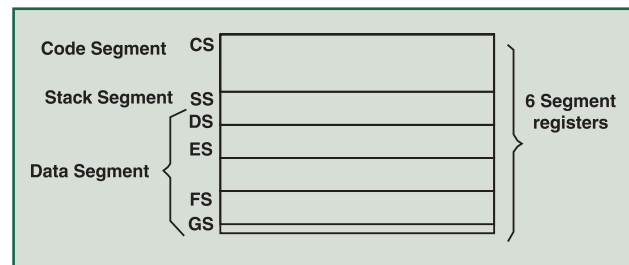


Figure 1: IA-32 memory segments

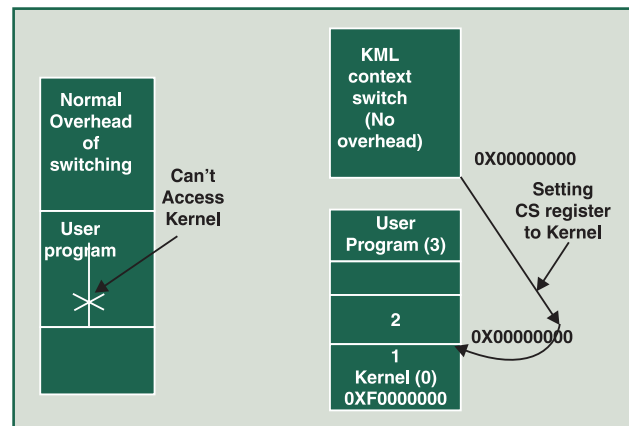


Figure 2: Kernel protection by Linux (diagram on left); kernel-level privileges and KML for context switching (right)

```
int r;
r = *(int *)0xC0000000;
printf ("It is working fine in KML\n");
return r;
}
```

## How to use KML for AMD64 and IA-32 architectures

To use KML, all you have to do is patch the original source of Linux with the KML patch, and enable the Kernel Mode Linux option at the configuration phase, as is also done with other kernel patches. Then install and reboot the system for Linux-KML option execution (Figure 2). The KML patch is available from the KML website. As of now, KML only runs successfully on the IA-32 and AMD64 platforms.

System call invocations are set automatically into fast, direct function calls, without modifying user programs on IA-32 system architectures using the GNU C library. They have mechanisms to choose one of the several methods that the kernel provides for system call invocation, and KML is provided with direct function calls to invoke system calls.

A patch for GNU C Library is available for AMD64, since it does not have a mechanism similar to IA-32 mentioned earlier. Note that the kernel-mode user processes can invoke system calls rapidly, which are automatically translated to function calls.

To execute user programs in the kernel mode, the KML

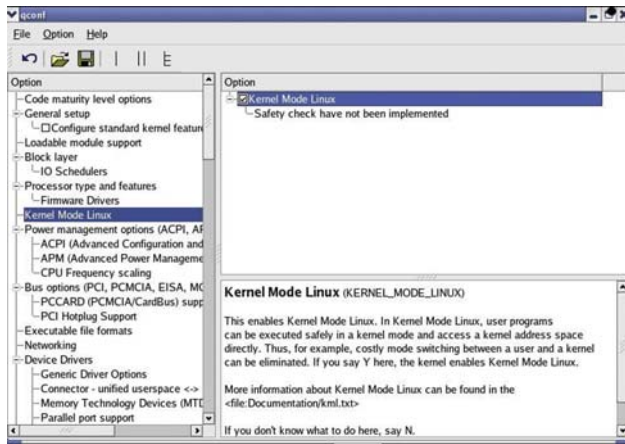


Figure 3: Kernel mode configuration

has a special *start\_thread* (*start\_kernel\_thread*) routine, which is called in while processing *execve* and sets the registers of a user process to specified initial values. The original *start\_thread* routine sets CS segment register to *\_\_USER\_CS*. The *start\_kernel\_thread* routine sets the CS register to *\_\_KERNEL\_CS*. Thus, a user program started as a user process is executed in the kernel mode.

## Interrupt handling

The program counter of an IA-32 CPU is a pair of a segment and an offset in the segment—the CS segment register and the EIP register, respectively. A privilege level of a running program is determined by the segment that is pointed by the CS segment register. For example, if the segment pointed by the CS segment register is defined as the kernel mode, the program is executed in the kernel mode.

‘Interrupt’ is more than a simple mechanism for coordinating I/O transfers. It enables the transfer of control from one program to another, to be initiated by an event external to the computer. Generally, an interrupt is identified by an unsigned 1-byte integer, called a vector. The vector ranges between 0 and 255. The first 32 (0-31) vectors are exceptions and non-maskable interrupts. The range from 32-47 is assigned to maskable interrupts and is generated by IRQs (0-15 IRQ line numbers). The last range, from 48-255, is used to identify software interrupts; an example of this is the interrupt 128 (*int 0x80* assembly instructions), which is used to implement system calls.

At initialisation, the kernel sets the target address of the software interrupt to the address of a special routine that handles system calls. To invoke system calls, a user program executes a special instruction, *int 0x80*. Then, the system-call handling routine in the kernel is executed in kernel mode. The routine performs a context switch—that is, it saves the content of the registers of the user program. Finally, it calls the kernel function that implements the system service specified by the user program. In this case, the scheduling, signalling and paging are as usual as ordinary processes.

## Stack starvation in the kernel mode and its solution

The original Linux kernel is supplied with the ability to handle exceptions (page faults) and interrupts on the IA-32 CPU. When an interrupt occurs, an IA-32 CPU stops execution of the running program, saves the execution context of that program, and executes the interrupt handling routine.

As mentioned earlier, in an interrupt-handling mechanism, if the user programs are executed in the kernel mode, there are limitations to such implementations. For example, if a user program in the kernel mode accesses its stack, which is not mapped by page tables of a Main Memory Unit (MMU), a page fault occurs first. In case of the IA-32 CPU, it tries to interrupt the running program and jumps to a page-fault handler. Since it is running in the kernel mode, the CPU cannot accomplish the work and there is no stack to save the context to. Since the process is executed in the kernel mode, the CPU can never switch the memory stack to the kernel stack. In an attempt to generate a special interrupt called the double fault, the CPU fails in its execution since there is no stack to save the running context. This problem of failure in the CPU’s performance and its resetting is known as ‘stack starvation’.

The stack starvation problem is handled in IA-32 CPUs by exploiting its task management facility. The IA-32 task management facility, which is provided to support process management of the kernel, can switch between processes with only one instruction. This facility has become obsolete and has been almost forgotten nowadays.

In IA-32 CPUs, it saves the execution context of the interrupted program to a task data structure instead of to memory stacks. Since tasks managed by the IA-32 CPU can be set to the Interrupt Descriptor Table (IDT), it is possible to restore the context from the task data structure specified in the IDT.

## Limitations of KML

Although kernel mode user processes are ordinary user processes, they are bound by the following limitations:

1. A program shouldn’t modify the CS, DS, SS or FS segment registers. The current KML for IA-32 assumes that these segment registers are not modified by kernel-mode user processes, and it uses them internally.
2. A program shouldn’t perform privileged actions improperly. In kernel mode, programs can perform any privileged action. However, if the program performs such actions in a way that is inconsistent with the kernel, the system will be in an undefined state. For example, if we use a CLI program to disable the interrupt globally and call an infinite loop, the system will hang. In the worst case scenario the system will crash.
3. KML is only available for IA-32 /AMD-64 architecture.

## Build a KML-enabled kernel

KML is available as a patch from the KML website (<http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml>). Apply the patch to the Linux kernel. We used Linux 2.6.17 for our experiments.

1. patch -p1 <.../kml\_2.6.17\_001.diff
2. make xconfig

The screenshot of KML configuration is shown in Figure

3.

After it is configured we need to build the modules. At a shell prompt, run the following commands:

1. make modules
2. make modules\_install
3. make install

The *make install* command is the final stage of the kernel building and it will also update the boot loader.

Once *Linux-KML* is installed successfully, we need to create the */trusted* directory to execute any time-critical program directly in kernel mode, which shows a drastic performance improvement compared to the general Linux kernel.

## Experimental results

A system can provide very-high resolution time measurements through the time-stamp counter, which counts the number of instructions since boot. For Pentium-class systems, we can get the number of clock cycles elapsed since the last reboot with the following C code:

```
#include <sys/time.h>
unsigned long long rdtsc ()
{
    unsigned long long dst;
    __asm__ __volatile__ ("rdtsc":"=A" (dst));
}

int main (void) {
    int i, nano;
    unsigned long long int start, end;
    for (;;)
    {
        start = rdtsc();
        /* execute your system calls here*/
        end = rdtsc();
        nano = (end-start)/1.8;
        printf("TSC difference is : %d
bsec\n", nano);
    }
    return 0;
}
```

The Intel processor includes a CLK input pin, which receives the clock signal of an external oscillator. Starting with Pentium, many recent Intel microprocessors include a 64-bit time stamp counter register that can be read by means of the *rdtsc* assembly language instruction. The *rdtsc* instruction (read time-stamp counter) can be used to access the time-stamp counter and provide accurate real-

**TABLE 1: DIFFERENT SYSTEM CALLS EXECUTION TIME TABLE (100 LOOP COUNTS)**

System calls	/trusted(ns)	other(ns)	%
open-close	201028	268248	25.3
link and unlink	195484	269451	27.45
dup	199806	263891	24.2
signal	206759	269671	23.33

**TABLE 2: DIFFERENT SYSTEM CALLS EXECUTION TIME TABLE WITH DIFFERENT LOOP COUNTS**

System calls	/trusted(ns)	User programme(ns)	%	Loop count
getpid	3111	22904	86.4	100
getpid	30102	272788	88.96	1000
getppid	32431	228217	85.8	1000
getppid	361771	22029226	98	10000
getuid	3211	22989	86	100
getuid	32424	269782	88	1000

time readings. Moreover, the *rd\** operation modifies the parameters directly (without using pointer direction). The *rdtsc* is defined in the */usr/src/linux-2.6.x/asm-i386/msr.h* file for access to the machine-specific register. In the x86 assembly language, the *rdtsc* instruction is a mnemonic for read time stamp counter. In general, *rdtsc* can be implemented as follows for a 32-bit processor:

```
static __inline__ unsigned long long
rdtsc(void)
{
    unsigned long long int x;
    __asm__ __volatile__ (".byte 0x0f, 0x31":
    "=A" (x));
    return x;
}
```

Here *\_\_asm\_\_ volatile* takes care of accessing from registers (A) pointing to x (variable). It is an inline assembler instruction that works, for example, similar to a process that registers rise in temperature—each time it records a new value of the variable just as if there is a temperature rise. Since there is a time counter that needs to be specified, *rdtsc* takes care of the actual time stamp in the register with the declared *dst* variable.

This can be executed using loop counts, and the system time is measured for better evaluation. A program with system call invocations and *get pid* measuring the system time included in a *for* loop is as follows:

```
#include <sys/time.h>
#include <stdio.h>
unsigned long long rdtsc ()
```



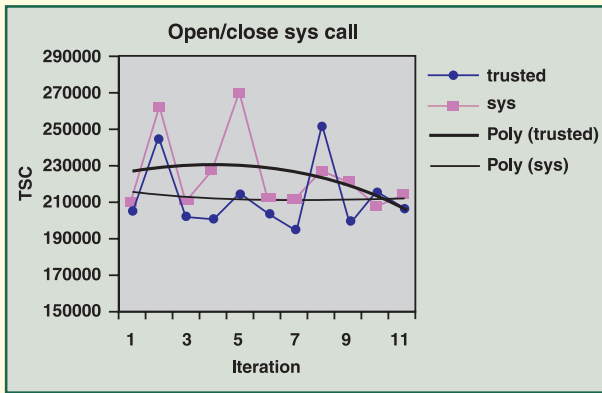


Figure 4: Results of open/close () syscall

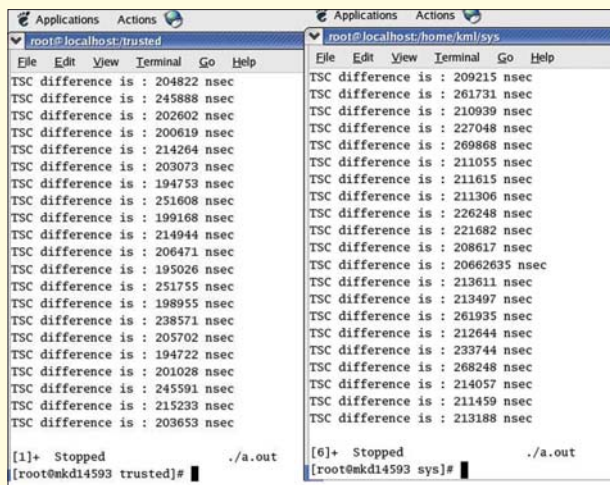


Figure 5: Open and close system call invocations

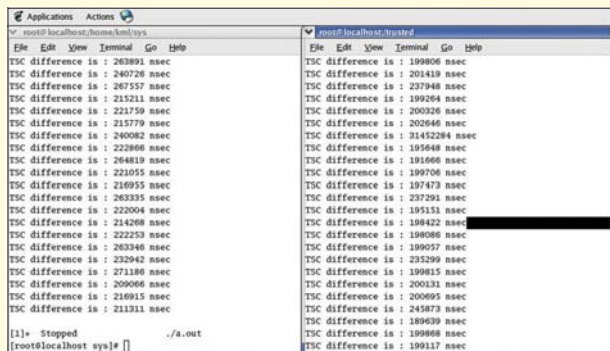


Figure 6: Dup system call invocation

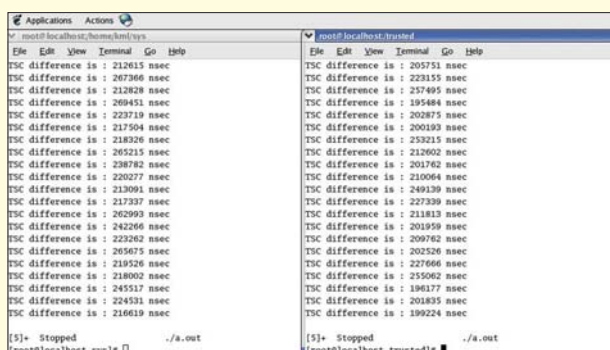


Figure 7: Link and unlink system calls

```
{
    unsigned long long dst;
    __asm__ __volatile__ ("rdtsc;" : "A" (dst));
    for (;;)
    {
        start = rdtsc();
        for(i=0; i<=10000; i++);
        getpid();
        end = rdtsc();
        nano = (end-start)/1.8;
        printf("TSC difference is : %d\n", nano);
    }
    return 0;
}
```

The recorded time for evaluating system performance for the kernel and user program is shown in Table 1 and Table 2.

## Results of some of the system call invocations using KML

The trend lines (Poly(trusted) and Poly(sys)) show a reduction in system call execution time using KML. Figure 4 shows time TSc diff vs the number of cycles of executing system calls in user programs and kernel mode. A polyline trend shown for both assumes a set of variations as the CPU ticks. Since the variation in system time ticking resembles a polyline, a polyline trend is drawn for both kernel and user execution. It shows a reduction in execution time and a performance improvement by 27 per cent, as evident from Table 1. Since Open-Close (Figure 5), Dup (Figure 6) and other system calls (Figure 7) are involved with many kernel functions, apart from switching from the user to the kernel mode, the performance difference with and without KML is expected to be less as evident from Table 2. The *getpid*, *getppid*, *getuid* function calls won't use much of the kernel resources since they just read the corresponding value from the task-struct structure. Here, the only overhead is the user-to-kernel switching, and thus the performance of these

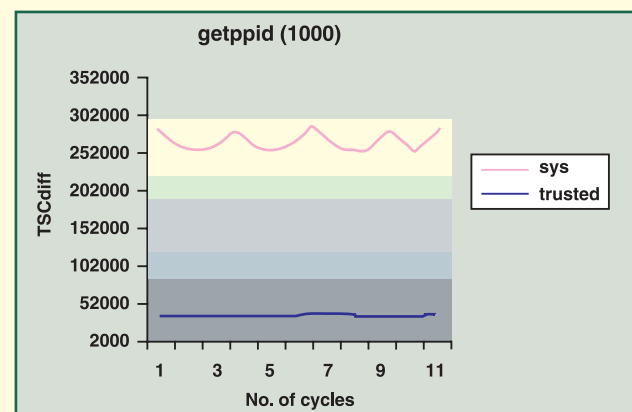



Figure 8: shows getpid ()

system calls in KML is very high. They return a value for each of those calling functions. The graphical output (Figure 4) for the kernel-level execution using KML is steady with a value of 30,102 against varying user program execution times, as is evident from the graph with an 80-90 per cent performance enhancement in case of *getppid* (Figure 10).

## In a nutshell

Kernel Mode Linux provides a platform to go beyond real-time operations by executing user programs in kernel mode without the overhead of switching between user mode and kernel mode. Even though interrupt handling and stack starvation includes a little effort for interrupt

handling, KML's performance varies from 5-30 per cent in case of open-close sort of system calls. Function calls like *getpid*, *geppid*, *getuid* show a performance enhancement by 80-90 per cent over normal user execution. It can also be observed that improvement is steady for several cycles of execution of these system calls. Currently, it works on AMD-64 and IA-32 platforms, and has to be made available on other architectures as well. Running user applications in real-time also needs to be tested on KML—it can be further built in developing cognition applications on Linux. **END** 

## REFERENCES

- *Kernel Mode Linux* by Toshiyuki Maeda on, may 2003 Linux Journal, Issue 109, <http://www.linuxjournal.com/article/6516>
- *Kernel Korner—Kernel Mode Linux for AMD64* by Toshiyuki Maeda in August 2005 in Linux Journal, Issue 136, <http://www.linuxjournal.com/article/8023>
- *Safe Execution of User Programs in Kernel Mode Using Typed Assembly Language* by Toshiyuki Maeda. A master thesis submitted to The Graduate School of the University of Tokyo, February 5, 2002, pp 21-44.
- *Computer Organisation* by Carl Hamacher, Zvonko Vranesic, Sifat Zaky, Chapter 3, Mc Graw Hill Publication. pp 104-140
- KML Home Page—<http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/>

*By: Ramesha J and Dr B. Thangaraju. Ramesha has done his M. Tech computer cognition technology from department of studies in computer science, University of Mysore. Previously, He was with ADA, ASTRA (IISc), NIAS. This work was carried out at Wipro Technologies as a M.tech project trainee on a project "cognition applications in embedded systems using UNIX". Presently, he is working as a Senior Faculty in the Department of Computer Science and Engineering at East West Institute of Technology, Bangalore. Thangaraju is a PhD in Physics and has worked as a research associate for five years at the IISc. Presently, he is working as a senior consultant with Wipro Technologies. He works in the areas of Linux internals, the Linux kernel, device drivers and embedded and real-time Linux.*

### Acknowledgements

*We would like to thank Thoshiyuki Maeda for his valuable work on Kernel Mode Linux. Ramesha also wishes to thank Talent Transformation, Wipro Technologies for the support given for carrying out these studies and experiments.*

support@  
efyindia.com

Do you have a query,  
suggestion or a complaint?

You can e-mail it to  
**support@efyindia.com** and we  
will take care of the rest.