

2015 Spring - System Programming (002)

Kernel Driver Lab Report

2013-11438 함지웅(Hahm, Jiung)

1) Warming Up – Implement a kernel module that outputs the list of all parent processes up to the root when called through ioctl()

Introduction

Module is like a program that can be loaded into Linux Kernel through some command, and on demand. Commands for loading, unloading are basically 'insmod module_name.ko', 'rmmod module_name'.

These modules can execute privileged instructions that can only be executed in kernel mode without system calls. Typically user programs need system calls to do that.

Device drivers are one kind of kernel module, that supports methods to communicate with the virtual or real hardware. There are three types of device drivers.

'Character Device Drivers' communicates in character unit and does not need buffer for supporting.

'Block Device Drivers' communicates in block unit, using buffers. Hard disks are examples of 'Block Device Drivers'.

'Network Device Drivers' communicates via network protocols.

And each device driver has its major number and minor number. If drivers use same device, they have same major numbers. In other words, major numbers are different by device. Minor number is driver identification number, which is different for different drivers that use same device.

To use character device drivers, we have to know few things.

First, when user process calls(opens) character device driver, in module, init_module() must call register_chrdev() function to register character device driver and use it with no problems. And, for registration, major numbers and minor numbers should be set appropriately.

Second, user process can communicate with device driver via ioctl, which stands for 'Input and Output ConTrol'. By setting appropriate predefined macros as parameters of ioctl() function, we can read, write to the device drivers.

Problem Statement

We have to build Kernel module that should outputs the list of all parent processes up to the root(init process) when this module is called through ioctl function. We should call ioctl function in user process, and in device driver that is called, it should find the list of all parent processes and must return that information via ioctl to user process. We will use character device driver to do this, and output(list of all parent processes) will be obtained in form of character array. After that, parse the character array to readable form and print it out to stdout in user process.

Code files

chardev.c : Main module file coded in C.

chardev.h : Header file for chardev.c and ioctl_test.c.

ioctl_test.c : Code for user process.

chardev.c

This file is main file for kernel module. Includes various header files and necessary functions are implemented in this file.

Functions and Variables :

- 1) **Device_open** : Indicates weather device is open or not.
- 2) **Message[BUF_LEN]** : Array used for getting contents from user.
- 3) **Message_Ptr** : char pointer that points the character array(string) which device keeps.
- 4) **call_tree_Ptr** : char pointer that points the character array(string) that has generated by tracking parent of the process up to root. This will be passed to user.
- 5) **device_open()** : When device is open from process, do appropriate things, such as incrementing Device_Open to indicate how many process opened this device driver.
- 6) **device_release()** : When device is released from process, do appropriate things, such as decrementing Device_Open to indicate how many process opened this device driver and so on.
- 7) **string_adder(), int_to_str()** : Used to make process call tree string(char array).
- 8) **device_read()** : Generated process call tree. This is done by first get pid of current process and real_parent of current process. Then set task structure pointer to real_parent and do things again. This will be done till when current process is init process whose pid is 1. The call tree result will be made to string using functions mentioned in 7.
- 9) **device_write()** : First, gets the user input byte by byte using get_user() function. And make the Message_Ptr to point the result.
- 10) **device_ioctl()** : This function is called whenever a process tries to do an ioctl on device. Appropriate function with ioctl_num parameter(second parameter of this function) will be called.
- 11) **Fops** : file_operations structure which contains functions to be called when a process does something to the device. Pointer of this structure will be passed into register_chrdev() function as one of its argument at function init_module()
- 12) **init_module()** : Initialize the module. It registers character device by register_chrdev() function.
- 13) **cleanup_module()** : Unregister the device by unregister_chrdev() function.

chardev.h

This file declares some common constants that are used in both chardev.c and ioctl_test.c file. Major number and ioctl parameters are also defined here.

Definitions :

- 1) **MAJOR_NUM** : Defines the major number to be used when registering the device.
- 2) **IOCTL_SET_MSG** : ioctl parameter for setting the message the character device contains.
- 3) **IOCTL_GET_MSG** : ioctl parameter for getting the message the character device contains. In this problem, when this is used as ioctl parameter, instead of message the character device contains, process call tree will be get in form of character array.
- 4) **IOCTL_GET_NTH_BYTE** : ioctl parameter for getting the specific character which has given number(from user) as index.
- 5) **DEVICE_FILE_NAME** : Defined device file name.

ioctl_test.c

This is the user file that opens 'chardev' device and communicates to get the process tree from device driver. It uses ioctl() function to do that.

Functions and Variables :

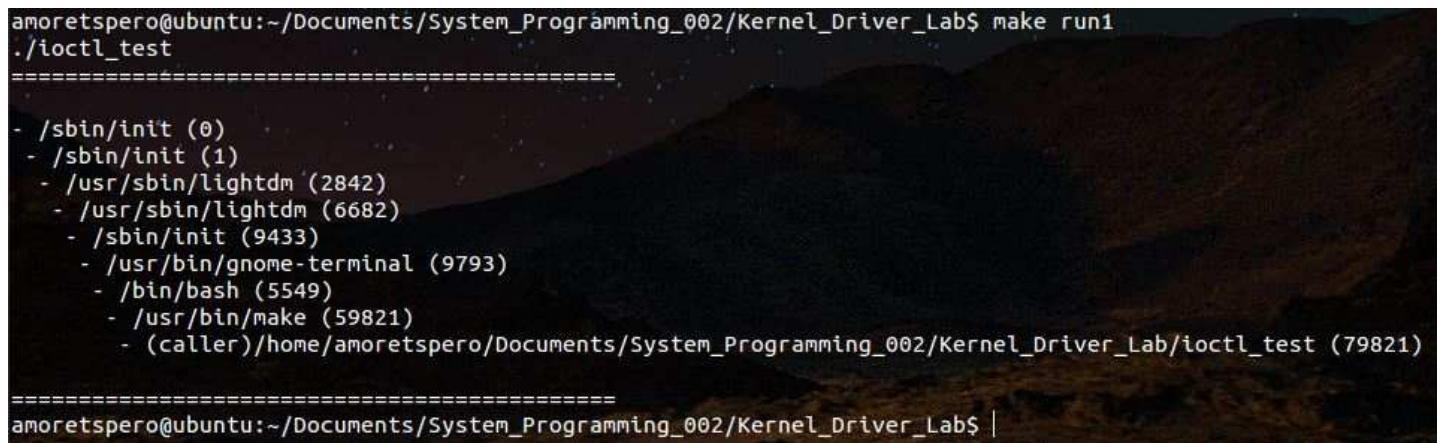
- 1) **string_formatter()** : Formats given string to increase readability.

- 2) **ioctl_set_msg()** : Passes the given string to device driver via `ioctl()` to be set as device driver's string.
- 3) **ioctl_get_msg()** : Get the process call tree from device driver and format it with `string_formatter()` function and at last print it to `stdout`.
- 4) **ioctl_get_nth_byte()** : Gets specified byte from the string that device driver has and by `putchar()` function, print it out to `stdout` until the end of string.(String has to be null terminated)
- 5) **main()** : Open device, and do the things such as get process call tree and print it via `ioctl_get_msg()` function.

Result

- 1) **System** : Core i7 4790K @ 4.0GHz(turbo boost : 4.4GHz), 4GB DDR3 RAM(Virtual),
Ubuntu Linux (Kernel version : 3.13) on VMware Workstation 10

- 2) **Result** :



```
amoretspero@ubuntu:~/Documents/System_Programming_002/Kernel_Driver_Lab$ make run1
./ioctl_test
=====
- /sbin/init (0)
- /sbin/init (1)
- /usr/sbin/lightdm (2842)
- /usr/sbin/lightdm (6682)
- /sbin/init (9433)
- /usr/bin/gnome-terminal (9793)
- /bin/bash (5549)
- /usr/bin/make (59821)
- (caller)/home/amoretspero/Documents/System_Programming_002/Kernel_Driver_Lab/ioctl_test (79821)
=====
amoretspero@ubuntu:~/Documents/System_Programming_002/Kernel_Driver_Lab$ |
```

Image 1. - Screen shot taken after building the module by 'make', and running command 'make run1'

Comment

- 1) Since Linux kernel has been gone through some rough changes from 2.6.x to 3.13.x, There are some functions used in this problem that does not exists anymore or has return type changed, etc.
After Linux kernel version 2.6.39 and after ones, `ioctl` functions does not need inode pointer to be one of its parameters. And return type of function has changed to `long`.
In Linux kernel version 3.13.x, `unregister_chrdev()` has return type 'void'. Which was 'int' in 2.4.x.
- 2) As we can see in Image 1, 'init' process has pid 1 definitely, but when we get that process's real parent with `mm` structure, parent's name is given as 'init' which is same with its child.
The process with pid zero is scheduler(swapper), which is responsible for paging and part of the kernel rather than a user-mode process. Plus, for `mm` struct of process with pid zero, when we access its real parent, that is always `init` process itself, and pid will be always zero.
In addition, Linux's all process has its parent process and initialized by `fork()` system call or `exec()` system call, but except swapper process, which has no parent process and executed manually.

2) Get serious - Implement a kernel module to manage the processor's performance monitoring unit (PMU)

Introduction

Intel has been providing MSR(stands for Model Specific Register)s since P6 processors. Intel Processors provides a variety of machine specific registers that are used to control and report on processor performance. Operations on these registers are not permitted in user applications, which is executed in the highest(outer most) performance ring.(Time stamp counter is only one exception.) But if we execute instructions in ring 0 privileges, we can read these MSRs and even write some binary data to them. We read MSR values by RDMSR instruction and write values to MSRs by WRMSR instruction. Basically, MSRs are a group of registers available primarily to OS or codes running in privilege level 0. As the name suggests, MSR's number and functions are different for each processors, mainly processor architecture or generations.

In these MSRs, there are Performance Monitoring Counters, which is introduced since introduction of Intel P6, Pentium processors. The information fetched from these registers can be used to monitor the performance and system, compiler tunings and many others. In the aspect of compatibility of performance monitoring, there are two classes. First class performance monitoring mechanism varies from generations and models of processors. These are called 'non-architectural'. Though they have less compatibility between different architectures of processors, have more events available for monitoring. Second class performance monitoring mechanism does not vary with architectures and these are called 'architectural'. This class has less kinds that are available for monitoring but consistent across processor architectures.

Problem Statement

We should implement a kernel module that writes to and reads the MSRs that are used to monitor performance monitoring. Read and write order will be issued within the user process, named 'msrtest' and will be passed via ioctl() function, executed in kernel mode in module we build.(named 'msrdrv') The result of reading MSRs will be passed to user process with ioctl() function. The goal is to first read some registers'(eax, ecx, edx) values and initialize the performance monitoring counters(except time stamp counter). Then enable the monitoring counters and execute some computations, which will be calculation of PI by Monte-Carlo method in my implementation. After that, performance monitoring counters will be stopped again and read some register(eax, ecx, edx), time stamp counter will be read and difference of time stamp count with start of execution will be calculated. At last, data will be read from performance monitoring counters.

Code files

msrdrv.c : Main file for kernel module, coded in C

msrdrv.h : Header file that specifies some predefined values.

msrtest.c : Code for user process that issues read and write MSRs.

msrdrv.c

This file is the kernel module file for module that will when called through ioctl() by user process, read or write some binary data to MSRs.

Functions and Variables

1) **MODULE_LICENSE** : Sets the license of this kernel module. Dual license with BSD and GPL is set.

2) **msrdrv_open()** : Defines what will be done when some process opens this driver file. Since we are only going to read or write to MSRs, this function may do nothing.

- 3) **msrdrv_release()** : Defines what will be done when the device file is released.
- 4) **msrdrv_read()** : Defines what will be done when process that opened this driver file try to read something from this driver.
- 5) **msrdrv_write()** : Defines what will be done when process that opened this driver file try to write something to this driver.
- 6) **msrdrv_ioctl()** : Defines what will be done when process that opened this driver file calls ioctl() function to communicate with this driver.
- 7) **msrdrv_dev()** : This will have the value that returned by MKDEV() function and will be used to register this driver. (MKDEV() function takes major number and minor number as its parameters and returns dev_t value.)
- 8) **msrdrv_cdev()** : This will have the value that cdev_alloc() function returns and will be used to initialize the character driver we are using via cdev_init() function. (cdev_alloc() function takes no arguments and returns allocated cdev structure. cdev_init() function takes cdev and file operations pointer and initializes character driver.)
- 9) **msrdrv_fops** : This structure defines the file operations of our driver. In this driver, 'owner', 'read', 'write', 'open', 'release', 'unlocked_ioctl', 'compat_ioctl(NULL)' has been defined and set to appropriate values.
- 10) **read_msr()** : With parameter of ecx value, reads MSR specified by ecx and return result as long long type.
- 11) **read_eax()** : read current eax value and returns it.
- 12) **read_ecx()** : read current ecx value and returns it.
- 13) **read_edx()** : read current edx value and returns it.
- 14) **write_msr()** : With parameters of ecx, eax, edx, writes to MSR specified by ecx the value edx:eax to that.
- 15) **read_tsc()** : read current time stamp counter value and returns result as long long type.
- 16) **msrdrv_ioctl()** : Performs ioctl functions specified by ioctl_param.
- 17) **msrdrv_init()** : Register and initializes driver.
- 18) **msrdrv_exit()** : Unregister driver.
- 19) **module_init()** : When module has to be initialized, call msrdrv_init() function.
- 20) **module_exit()** : When module has to be exited, call msrdrv_exit() function.

msrdrv.h

This header file is used for msrdrv.c and msrtest.c files. It contains some constants' definitions and enum type and structure.

- 1) **DEV_NAME** : Has value of "msrdrv" and this defines the character device's name.
- 2) **DEV_MAJOR** : Major number for this character device.
- 3) **DEV_MINOR** : Minor number for this character device.
- 4) **MsrOperation** : Defines macros to identify each MSR operations we will use.
- 5) **MsrInOut** : Structure that contains op(MsrOperation), ecx(specifies MSR), eax(low part of double word), edx(high part of double word), value(quad word that contains result value).

msrtest.c

Source code for the user process that will open the character device driver 'msrdrv' and communicate with that to get desired informations. This process will calculate PI value with Monte-Carlo method to see core cycle and time stamp goes by.

- 1) **totalPoints, calculatePi(), getDistanceFromOrigin(), calc_pi()** : Used to calculate PI value by Monte-Carlo method with number of trials specified at 'totalPoints'. This process will try 2^{26} times to find approximate value of PI.
- 2) **loadDriver()** : Opens the character device driver 'msrdrv'.
- 3) **closeDriver()** : Closes the character device driver 'msrdrv'.
- 4) **main()** : Defines MsrInOut structures for start of MSRs and stoppage of MSRs and for reading eax, ecx, edx

and time stamp counter. Plus, main() function prints the results in stdout.

MSR values

1) Time Stamp Counter :

This value can be read from time stamp counter register, with instruction RDTSC. TSC is 64-bit model-specific counter.

(Read with instruction RDTSC)

2) Longest Latency Cache Reference :

This value represents the Longest Latency Cache requests from specified core. In Intel Haswell generation i7 CPUs, LLC is L3 Cache, whose size is 8MB.

(Read from IA32_PERFECTSEL0 – UMASK 4FH, EVENT 2EH)

3) Longest Latency Cache Miss :

This value represents the Longest Latency Cache request from specified core that have failed and results in cache miss.

(Read from IA32_PERFECTSEL1 – UMASK 41H, EVENT 2EH)

4) UnHalted Core Cycles :

This counts the number of clock cycles that is dependent on actual CPU clock speed. If UMASK value is changed to 01H, that MSR will give UnHalted Core Cycles 'REF_XCLK' which counts reference clock cycles.

(Read from IA32_PERFECTSEL2 – UMASK 00H, EVENT 3CH)

5) Branch Misses Retired :

This counts the branch miss predictions for retired instructions.

(Read from IA32_PERFECTSEL3 – UMASK 00H, EVENT C5H)

6) Instruction Retired :

This counts the number of instructions that retire from execution. For instructions that consist of multiple micro operations, this event counts the retirement of the last micro operations of that instruction.

(Read from IA32_FIXED_CTR0)

7) CPU Clock Unhalted – Core :

This counts the number of core cycles. In Intel CPU generations Sandy-Bridge and after ones, This is same as UnHalted Core Cycles.

(Read from IA32_FIXED_CTR1)

8) CPU Clock Unhalted – Ref :

This counts the number of reference core cycles. In Intel CPU generations Sandy-Bridge and after ones, This is same as UnHalted Core Cycles REF_XCLK.

(Read from IA32_FIXED_CTR2)

Result

- 1) System : Core i7 4790K @ 4.0GHz(turbo boost : 4.4GHz), 4GB DDR3 RAM(Virtual),
Ubuntu Linux (Kernel version : 3.13) on VMware Workstation 10
- 2) Result :

```
amoretsperso@ubuntu:~/Documents/System_Programming_002/Kernel_Driver_Lab$ make run2
./msrtest

=====
Driver has been loaded!

Before starting instructions :
eax : 0000000000000005, ecx : 0000000046fce820, edx : 0000000046fce830
time stamp : 0078236412558854

=====
Performance Monitoring Unit has been reset and started.

Now calculating PI value by Monte-Carlo Method with n=2^26

Estimating Pi Using Monte-Carlo Method

Pi is estimated as: 3.141833
Time : 2.023485
Calculation has been finished.

Performance Monitoring Unit has been stopped.

=====
After executed instructions :
eax : 0000000000000005, ecx : 0000000046fce820, edx : 0000000046fce830
time stamp : 0078244511172471

time stamp difference : 0000008098613617

=====
Results :
Longest Latency Cache Reference:      16705
Longest Latency Cache Miss:          2733
UnHalted Core Cycles:                 8810119575
Branch Misses Retired:               18790803
Instruction Retired:                  11928788932
CPU Clock UnHalted - Core:           8832483580
CPU Clock UnHalted - Ref:            8029531480

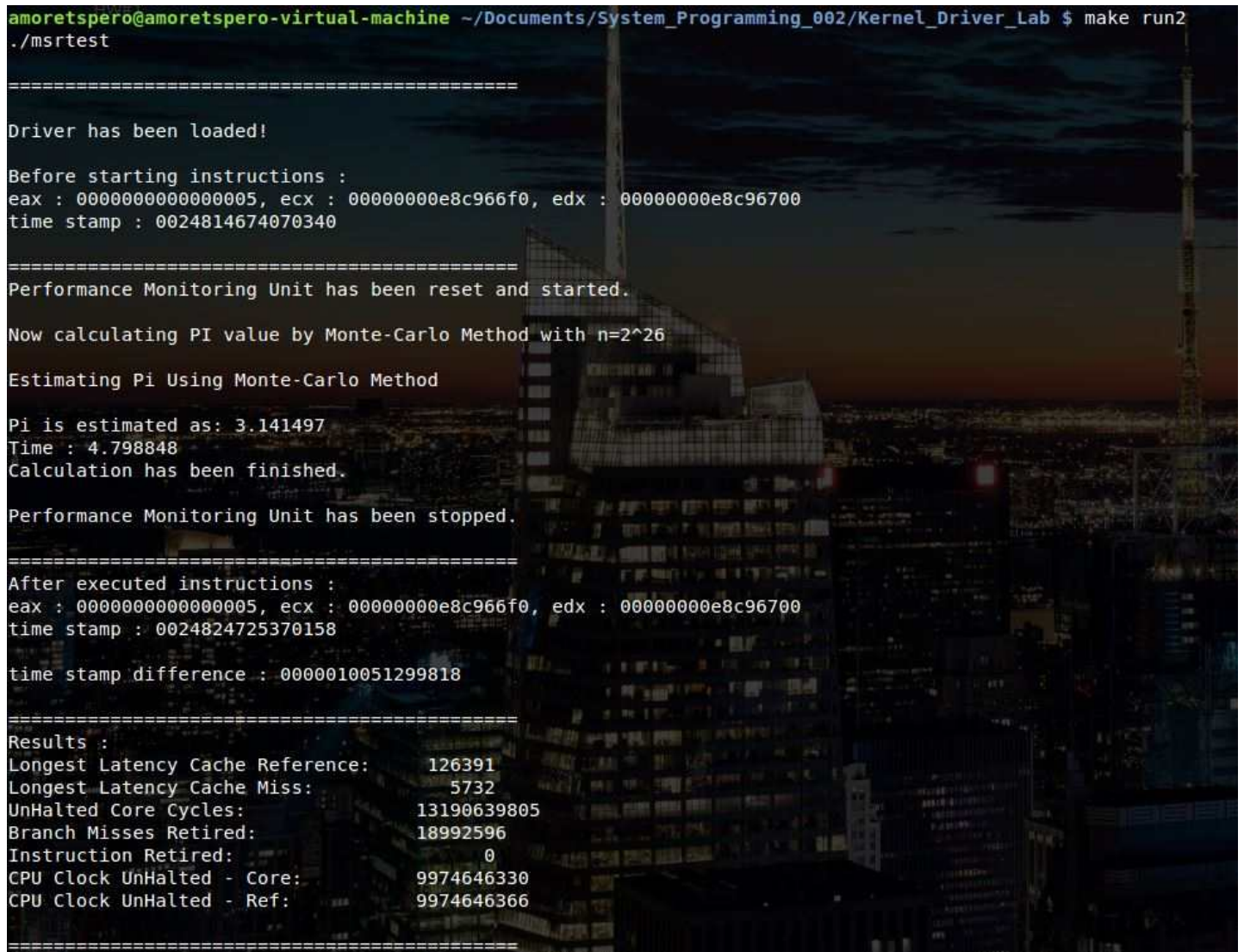
=====
```

Image 2. Screen shot taken after building the module by 'make', and running command 'make run2'

Comments

- 1) Time stamp counter counts almost same with CPU Clock Unhalted Ref(there may be some difference with CPU generations.). The CPU I used to test is i7-4790K, whose uncore ratio is x40 and turbo ratio is x44. So, after dividing time stamp difference and CPU Clock Unhalted Ref with $\text{time} \times (10^9) = 2.023485 \times 10^9$, we get 4.0023(GHz), 3.9682(GHz), which are almost same with uncore frequency of i7-4790K, which is 4.00(GHz). Unhalted Core Cycles and CPU Clock Unhalted Core will count the actual clock cycles of CPU, which is affected by turbo boost. So, after dividing both value with $\text{time} \times (10^9) = 2.023485 \times 10^9$, we get 4.3539(GHz), 4.3650(GHz), which are almost same with max turbo frequency of i7-4790K, which is 4.40(GHz).

2) For some unknown reason (at least I can't find out), there are some difference in result when CPU changes. I have tested this problem with i7-3612QM CPU as well, but the 'Instruction Retired' prints out zero value. Plus, in i7-4790K, 'CPU Clock UnHalted – Core' used to count 'turbo-boost enabled' clock cycles, but in i7-3612QM, it counts the reference clock cycles, which turbo boost has no effect. Within my knowledge the 'architectural' MSRs including IA32_PERFEVTSELx and IA32_FIXED_CTRx will function same for Ivy-Bridge CPU(i7-3612QM) and Haswell CPU(i7-4790K). The other things results in the way they should be. This will be the question to be solved.



```
amoretspero@amoretspero-virtual-machine ~/Documents/System_Programming_002/Kernel_Driver_Lab $ make run2
./msrtest

=====

Driver has been loaded!

Before starting instructions :
eax : 0000000000000005, ecx : 00000000e8c966f0, edx : 00000000e8c96700
time stamp : 0024814674070340

=====

Performance Monitoring Unit has been reset and started.

Now calculating PI value by Monte-Carlo Method with n=2^26

Estimating Pi Using Monte-Carlo Method

Pi is estimated as: 3.141497
Time : 4.798848
Calculation has been finished.

Performance Monitoring Unit has been stopped.

=====

After executed instructions :
eax : 0000000000000005, ecx : 00000000e8c966f0, edx : 00000000e8c96700
time stamp : 0024824725370158

time stamp difference : 0000010051299818

=====

Results :
Longest Latency Cache Reference:    126391
Longest Latency Cache Miss:        5732
UnHalted Core Cycles:              13190639805
Branch Misses Retired:             18992596
Instruction Retired:                0
CPU Clock UnHalted - Core:         9974646330
CPU Clock UnHalted - Ref:          9974646366

=====
```

Image 3. Screen shot taken after building the module by 'make', and running command 'make run2' with i7-3612QM CPU on Linux Mint.

Reference

- 1) Intel 64 and IA-32 Architectures Software Developer's Manual – Combined Volumes : 1, 2A, 2B, 2C, 3A, 3B and 3C. Published in January 2015 (Order number : 325462-053US)
- 2) www.tldp.org/LDP/intro-linux/html/sect_04_02.html, accessed in 2015.04.14.
- 3) http://www.hpc.ut.ee/dokumendid/ips_xe_2015/vtune_amplifier_xe/documentation/en/help/, accessed in 2015.04.14.
- 4) <http://lxr.free-electrons.com/>, for linux references, accessed in 2015.04.
- 5) <http://www.mindfruit.co.uk/2012/11/intel-msr-performance-monitoring-basics.html>, accessed in 2015.04