Lab 4 - CPU Monitoring (Linux)

Objectives

- Offer an introduction to Performance Monitoring
- Present the main CPU metrics and how to interpret them
- Get you to use various tools for monitoring the performance of the CPU
- Familiarize you with the x86 Hardware Performance Counters

Contents

Tasks

Introduction

01. Performance Monitoring

Performance monitoring is the process of regularly checking a set of metrics and tracking the overall health of a specific system. Monitoring is tightly coupled with performance tuning, and a Linux system administrator should be proficient in these two subjects, as one of their main responsibilities is to identify bottlenecks and find solutions to help the operating system surpass them. Pinpointing a Linux system bottleneck requires a deep understanding of how various components of this operating system work (e.g. how processes are scheduled on the CPU, how memory is managed , the way that I/O interrupts are handled, the details of network layer implementation, etc). From a high level, the main subsystems that you should think of when tuning are CPU, Memory, I/O and Network.

These four subsystems are vastly depending on each other and tuning the whole system implies keeping them in harmony. To quote a famous idiom, “a chain is no stronger than its weakest link”. Thus, when investigating a system performance issue, all the subsystems must be checked and analysed.

Being able to discover the bottleneck in a system requires also understanding of what types of processes are running on it. The application stack of a system can be broken down in two categories:

- CPU Bound - performance is limited by the CPU
  - Requires heavy use of the CPU (e.g. for batch processing, mathematical operations, etc)
  - e.g. High volume web servers
- I/O Bound - performance is limited by the I/O subsystem
  - Requires heavy use of memory and storage system
  - An I/O Bound application is usually processing large amounts of data
  - An often behaviour is to use CPU resources for making I/O requests and to enter a sleeping state
  - e.g. Database applications

Before going further with the CPU specific metrics and tools, here is a methodical approach which can guide you when tuning the performance of a system:

- Understand the factors which affect the performance

- Create a baseline measurement with the normal performance of the system
- Reproduce the issue and compare the measurements with the baseline to narrow down the bottleneck to a specific subsystem
- Try a single change at a time and test the results

02. Introducing the CPU and CPU Metrics

Before looking at the numerous performance measurement tools present in the Linux operating system, it is important to understand some key concepts and metrics, along with their interpretation regarding the performance of the system.

The kernel contains a scheduler which is in charge of scheduling two types of resources: interrupts and threads. The resources are assigned by the scheduler with different priorities. The following list presents the priorities:

- User Processes - All the processes running in the user space - having the lowest priority in the scheduling mechanism
- System Processes - All kernel processing
- Interrupts - Devices announcing the kernel that they are done processing

Context Switches

While executing a process, the necessary set of data is stored in registers on the processor and cache. This group of information is called a context. Each thread owns an allotted time quantum to spend on the CPU, and when the time finishes or it is preempted by a higher priority task, a new ready to run process will be scheduled. When the next process is scheduled to run, the context of the current will be stored and the context of the new one is restored to the registers, this process being named context switch. Having a great volume of context switching is not desired because the CPU has to flush its register and cache each time, to gain room for the new process, which leads to performance issues.

The Run Queue

Each CPU preserves its own run queue of threads. In an ideal scenario, the scheduler would be constantly executing threads. Threads can be in different states: runnable - processes which are ready to be executed or in a sleep state - being blocked while waiting for I/O. If the system has performance issues or it’s overloaded, then the queue starts to fill up and a process thread will take longer to execute.

The same concept is known also as “load”. This term is measured by load average, which is a rolling average of the sum of the processes waiting to be processed and the processes waiting for uninterruptible task to be completed. Unix systems traditionally present the CPU load as 1-minute, 5-minute and 15-minute averages.

CPU Utilisation

The CPU Utilisation is a meaningful metric to observe how the running processes make use of the given processing resources. You can find the following categories the vast majority of performance monitoring tools:

- User time - the time percentage a CPU spends on user processes
    - High user time values are recommended because this usually means that the system carries out actual work
- System time - the time percentage a CPU spends on kernel threads and interrupts
    - High system time values could mean bottlenecks in the network and driver stack
- Waiting I/O - the time percentage a CPU waits for a I/O event to occur

- A system should not spend too much time waiting for I/O operations
- Idle time - the time percentage a CPU spends waiting for tasks
- Nice time - the time percentage spends on changing the priority and execution order of processes. It is often included in the user time

03. CPU Performance Monitoring

The Linux distributions have various monitoring tools available. Some of the utilities deal with metrics in a single tool, providing well formatted output which eases the understanding of the system performance. Other tools are specialized on more specific metrics and give us detailed information.

Some of the most important Linux CPU performance monitoring tools:

Tool	**Most useful function**
vmstat	System activity
top	Process activity
uptime, w	Average system load
ps, pstree	Displays the processes
iostat	Average CPU load
sar	Collect and report system activity
mpstat	Multiprocessor usage

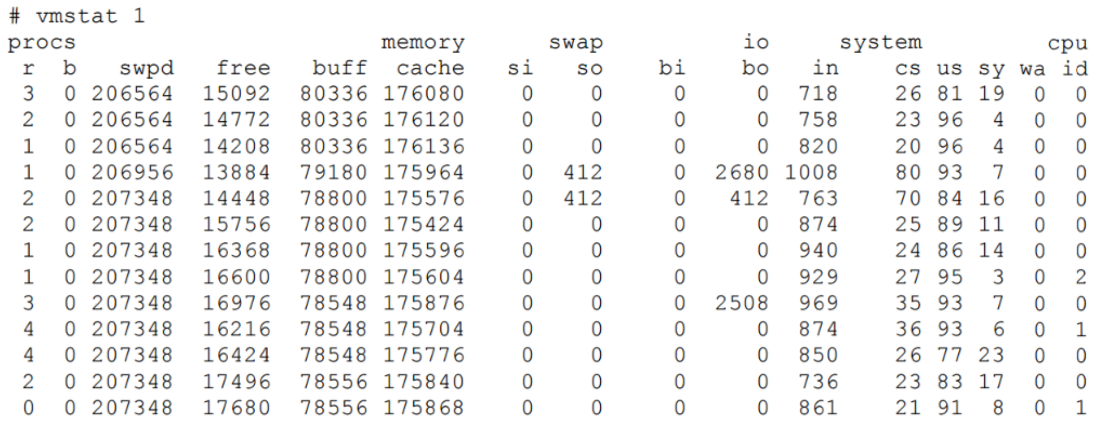
04. Examples

Understanding how well a CPU is performing is a matter of interpreting the run queue, its utilisation, and the amount of context switching performed. Although performance is relative to baseline statistics, in the absence of these statistics, the following general performance expectations of a system can be used as a guideline:

- Run Queues – A run queue should not have more than 3 threads queued per processor. For example, a dual processor system should not have more than 6 threads in the run queue.
- CPU Utilisation – A fully utilised CPU should have the following utilisation distribution:
  - 65% – 70% User Time
  - 30% – 35% System Time
  - 0% – 5% Idle Time
- Context Switches – The amount of context switches is directly relevant to CPU utilisation. As long as the CPU sustains the previously presented utilisation distribution, it is acceptable to have a high amount of context switches.

The following two examples give interpretations of the outputs generated by **vmstat**.

Example A - Sustained CPU Utilisation

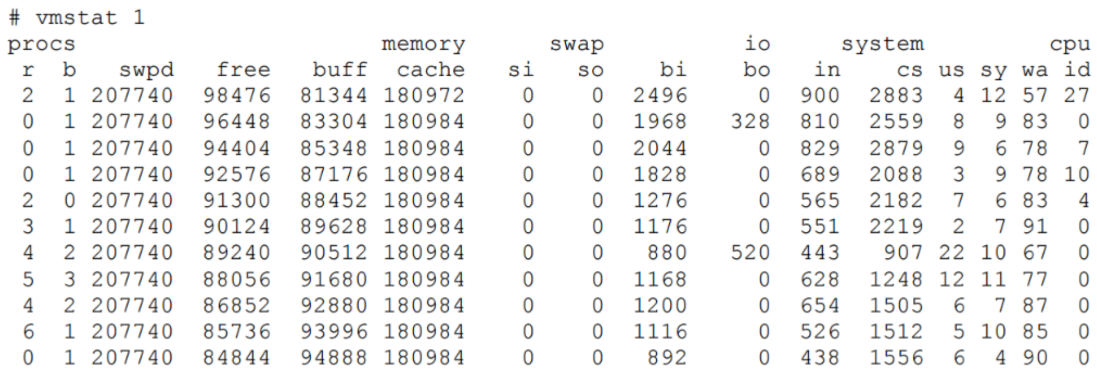


```
# vmstat 1
procs                        memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy wa id
 3  0 206564  15092  80336 176080    0    0     0     0  718    26 81 19  0  0
 2  0 206564  14772  80336 176120    0    0     0     0  758    23 96  4  0  0
 1  0 206564  14208  80336 176136    0    0     0     0  820    20 96  4  0  0
 1  0 206956  13884  79180 175964    0  412     0  2680 1008    80 93  7  0  0
 2  0 207348  14448  78800 175576    0  412     0   412  763    70 84 16  0  0
 2  0 207348  15756  78800 175424    0    0     0     0  874    25 89 11  0  0
 1  0 207348  16368  78800 175596    0    0     0     0  940    24 86 14  0  0
 1  0 207348  16600  78800 175604    0    0     0     0  929    27 95  3  0  2
 3  0 207348  16976  78548 175876    0    0     0  2508  969    35 93  7  0  0
 4  0 207348  16216  78548 175704    0    0     0     0  874    36 93  6  0  1
 4  0 207348  16424  78548 175776    0    0     0     0  850    26 77 23  0  0
 2  0 207348  17496  78556 175840    0    0     0     0  736    23 83 17  0  0
 0  0 207348  17680  78556 175868    0    0     0     0  861    21 91  8  0  1
```

The following observations can be made based on this output:

- There are a high amount of interrupts (**in**) and a low amount of context switches (**cs**). It appears that a single process is making requests to hardware devices.
- To further prove the presence of a single application, the user (**us**) time is constantly at 85% and above. Along with the low amount of context switches, we deduce that the process comes on the processor and stays on the processor.
- The run queue is just about at the limits of acceptable performance. On a couple occasions, it goes beyond acceptable limits.

Example B - Overloaded Scheduler



```
# vmstat 1
procs                        memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy wa id
 2  1 207740  98476  81344 180972    0    0  2496     0  900  2883  4 12 57 27
 0  1 207740  96448  83304 180984    0    0  1968   328  810  2559  8  9 83  0
 0  1 207740  94404  85348 180984    0    0  2044     0  829  2879  9  6 78  7
 0  1 207740  92576  87176 180984    0    0  1828     0  689  2088  3  9 78 10
 2  0 207740  91300  88452 180984    0    0  1276     0  565  2182  7  6 83  4
 3  1 207740  90124  89628 180984    0    0  1176     0  551  2219  2  7 91  0
 4  2 207740  89240  90512 180984    0    0   880   520  443   907 22 10 67  0
 5  3 207740  88056  91680 180984    0    0  1168     0  628  1248 12 11 77  0
 4  2 207740  86852  92880 180984    0    0  1200     0  654  1505  6  7 87  0
 6  1 207740  85736  93996 180984    0    0  1116     0  526  1512  5 10 85  0
 0  1 207740  84844  94888 180984    0    0   892     0  438  1556  6  4 90  0
```

The following observations can be made based on this output:

- The amount of context switches is higher than interrupts, suggesting that the kernel has to spend a considerable amount of time context switching threads.
- The high volume of context switches is causing an unhealthy balance of CPU utilisation. This is evident by the fact that the wait on IO percentage is extremely high and the user percentage is extremely low.
- Because the CPU is blocked waiting for I/O, the run queue starts to fill and the amount of threads blocked waiting on I/O also fills.

These examples are from Darren Hoch's *Linux System and Performance Monitoring*.

Tasks

01. [20p] Vmstat

The **vmstat** utility provides a good low-overhead view of system performance. Since **vmstat** is such a low-overhead tool, it is practical to have it running even on heavily loaded servers when it is needed to monitor the system's health.

[10p] Task A - Monitoring stress

Run **vmstat** on your machine with a 1 second delay between updates. Notice the CPU utilisation (info about the output columns here).
In another terminal, use the **stress** command to start N CPU workers, where N is the number of cores on your system. Do not pass the number directly. In stead, use command substitution.
[10p] Task B - How does it work?

Let us look at how **vmstat** works under the hood. We can assume that all these statistics (memory, swap, etc.) can not be normally gathered in userspace. So how does **vmstat** get these values from the kernel? Or rather, how does any process interact with the kernel? Most obvious answer: ***system calls***.

```
$ strace vmstat
```

“All well and good. But what am I looking at?”

What you *should* be looking at are the system calls after the two **write**s that display the output header (hint: it has to do with **/proc/** file system). So, what are these files that **vmstat** opens?

```
$ file /proc/meminfo
$ cat /proc/meminfo

$ man 5 proc
```

The manual should contain enough information about what these kernel interfaces can provide. However, if you are interested in *how* the kernel generates the statistics in **/proc/meminfo** (for example), a good place to start would be meminfo.c (but first, SO2 wiki).
02. [20p] Mpstat

Open fact_rcrs.zip and look at the code.
[10p] Task A - Python recursion depth

Try to run the script while passing 1000 as a command line argument. Why does it crash?

Luckily, python allows you to both retrieve the current recursion limit *and* set a new value for it. Increase the recursion limit so that the process will never crash, regardless of input (assume that it still has a reasonable upper bound).
[10p] Task B - CPU affinity

Run the script again, this time passing 10000. Use **mpstat** to monitor the load on each *individual* CPU at 1s intervals. The one with close to 100% load will be the one running our script. Note that the process might be passed around from one core to another.
Stop the process. Use **stress** to create N-1 CPU workers, where N is the number of cores on your system. Use **taskset** to set the CPU affinity of the N-1 workers and then run the script again. You should notice that the process is scheduled on cpu0.
Note: to get the best performance when running a process, make sure that it stays on the same core for as long as possible. Don't let the scheduler decide this for you, if you can help it. Allowing it to bounce your process between cores can drastically impact the efficient use of the cache and the TLB. This holds especially true when you are working with servers rather than your personal PCs. While the problem may not manifest on a system with only 4 cores, you can't guarantee that it also won't manifest on one with 40 cores. When running several experiments in parallel, aim for something like this:
Click to display ⇲
03. [20p] Zip with compression levels

The **zip** command is used for compression and file packaging under Linux/Unix operating system. It provides 10 levels of compression, where:

- **level 0** : provides no compression, only packaging
- **level 6** : used as default compression level
- **level 9** : provides maximum compression

```
$ zip -5 file.zip file.txt
```

[10p] Task A - Measurements

Write a script to measure the compression rate and the time required for each level. Use the following files:

- two largest bitmaps from here
- this large text file here

[10p] Task B - Plot

Fill the data you obtained into the **python3** script in plot.zip. Make sure you have **python3** and **python3-matplotlib** installed.

04. [40p] Hardware counters

A significant portion of the system statistics that can be generated involve hardware counters. As the name implies, these are special registers that count the number of occurrences of specific events in the CPU. These counters are implemented through **Model Specific Registers** (MSR), control registers used by developers for debugging, tracing, monitoring, etc. Since these registers may be subject to changes from one iteration of a microarchitecture to the next, we will need to consult chapters 18 and 19 from Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3B.
The instructions that are used to interact with these counters are RDMSR, WRMSR and RDPMC. Normally, these are considered privileged instruction (that can be executed only in ring0, aka. kernel space). As a result, acquiring these information from ring3 (user space) requires a context switch into ring0, which we all know to be a costly operation. The objective of this exercise is to prove that this is not necessarily the case and that it is possible to configure and examine these counters from ring3 in as few as a couple of clock cycles.
Before getting started, one thing to note is that there are two types of performance counters:

1. Fixed Function Counters
   - each can monitor a single, distinct and predetermined event (burned in hardware)
   - are configured a bit differently than the other type
   - are not of interest to us in this laboratory
2. General Purpose Counters
   - can be configured to monitor a specific event from a list of over 200 (see chapters 19.1 and 19.2)

Download hw_counter.zip.
Here is an overview of the following five tasks:

- **Task A**: check the version ID of your CPU to determine what it's capable of monitoring.
- **Task B**: set a certain bit in CR4 to enable ring3 usage of the RDPMC instruction.
- **Task C**: use some ring3 tools to enable the hardware counters.
- **Task D**: start counting L2 cache misses.
- **Task E**: use RDPMC to measure the cache misses for a familiar program.

[5p] Task A - Hardware Info

First of all, we need to know what we are working with. Namely, the microarchitecture *version ID* and the *number of counters* per core. To this end, we will use cpuid (basically a wrapper over the CPUID instruction.) All the information that we need will be contained in the 0AH leaf (might want to get the raw output of **cpuid**):

- **CPUID.0AH:EAX[15:8]** : number of general purpose counters
- **CPUID.0AH:EAX[7:0]** : version ID
- **CPUID.0AH:EDX[7:0]** : number of fixed function counters

Point out to your assistant which is which in the **cpuid** output.

[5p] Task B - Unlock RDPMC in ring3

This is pretty straightforward. All you need to do is set the **Performance-Monitor Counter Enable** bit in CR4. Naturally, this can't be done from ring3. As such, we provide a kernel module that does it for you (see *hack_cr4.c*.) When the module is loaded, it will set the aforementioned bit. Similarly, when the module is unloaded, it will revert the change. Try compiling the module, loading and unloading it and finally, check the kernel message log to verify that it works.

```
$ make
$ sudo insmod hack_cr4.ko
$ sudo rmmod hack_cr4
$ dmesg
```

[10p] Task C - Configure IA32_PERF_GLOBAL_CTRL

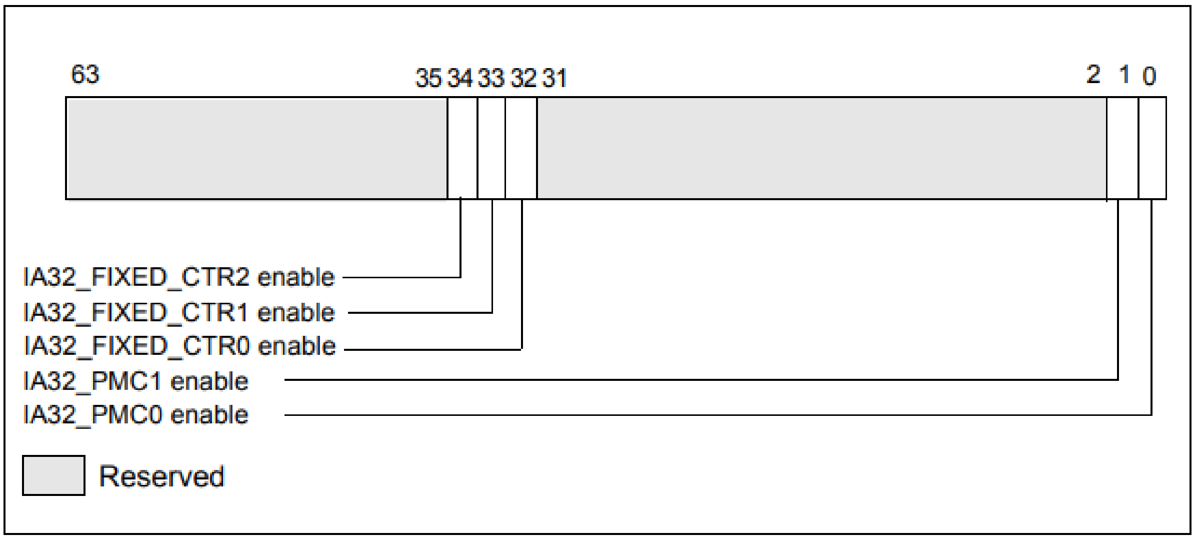


Figure 18-3. Layout of IA32_PERF_GLOBAL_CTRL MSR

The **IA32_PERF_GLOBAL_CTRL** (0x38f) MSR is an addition from *version 2* that allows enabling / disabling multiple counters with a single WRMSR instruction. What happens, in layman terms, is that the CPU performs an AND between each EANBLE bit in this register and its counterpart in the counter's original configuration register from *version 1* (which we will deal with in the next task.) If the result is 1, the counter begins to register the programmed event every clock cycle. Normally, all these bits should be set by default during the booting process but it never hurts to check. Also, note that this register exists for each logical core.

If for CR4 we had to write a kernel module, for MSRs we have user space tools that take care of this for us (rdmsr and wrmsr) by interacting with a driver called **msr** (install **msr-tools** if it's missing from your system.) But first, we must load this driver.

```
$ lsmod | grep msr
$ sudo modprobe msr
$ lsmod | grep msr
   msr                    16384  0
```

Next, let us read the value in the IA32_PERF_GLOBAL_CTRL register. If the result differs from what you see in the snippet below, overwrite the value (the **-a** flag specifies that we want the command to run on each individual logical core).

```
$ sudo rdmsr -a 0x38f
   70000000f
$ sudo wrmsr -a 0x38f 0x70000000f
```

[10p] Task D - Configure IA32_PERFEVENTSELx

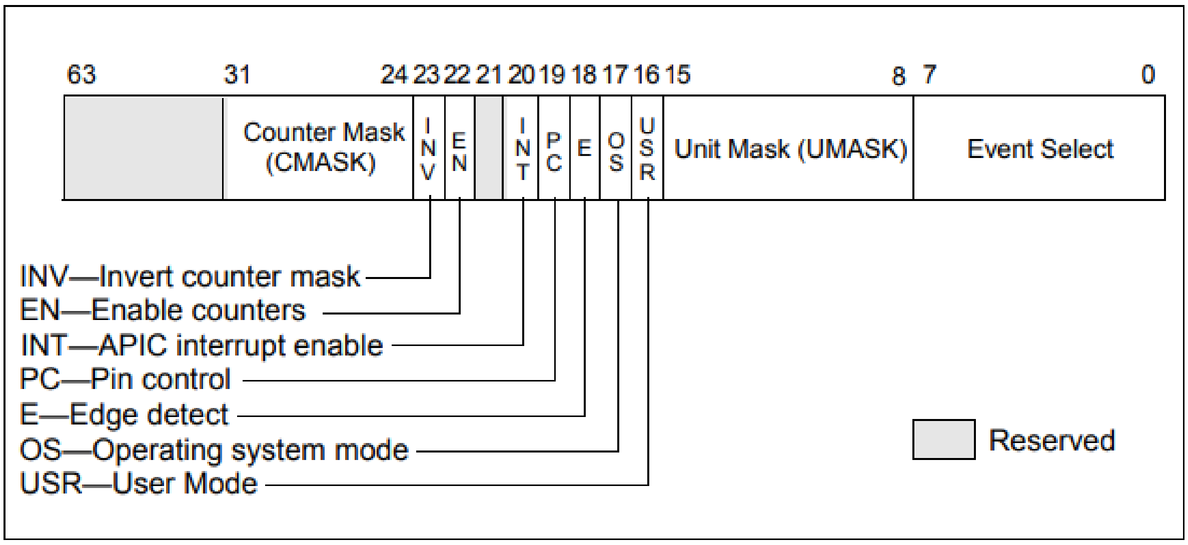


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

The **IA32_PERFEVENTSELx** are MSRs from *version 1* that are used to configure the monitored event of a certain counter, its enabled state and a few other things. We will not go into detail and instead only mention the fields that interest us right now (you can read about the rest in the Intel manual.) Note that the *x* in the MSR's name stands for the counter number. If we have 4 counters, it takes values in the 0:3 range. The one that we will configure is IA32_PERFEVENTSEL0 (0x186). If you want to configure more than one counter, note that they have consecutive register number (i.e. 0x187, 0x188, etc.).
As for the register flags, those that are not mentioned in the following list should be left cleared:

- **EN** (enable flag) = **1** starts the counter
- **USR** (user mode flag) = **1** monitors only ring3 events.2)
- **UMASK** (unit mask) = **??** depends on the monitored event (see chapter 19.2)
- **EVSEL** (event select) = **??** depends on the monitored event (see chapter 19.2)

Before actually writing in this register, we should verify that no one is currently using it. If this is indeed the case, we might also want to clear **IA32_PMC0** (0xc1). PMC0 is the actual counter that is associated to PERFEVENTSEL0.

```
$ sudo rdmsr -a 0x186
   0
$ sudo wrmsr -a 0xc1 0x00
$ sudo wrmsr -a 0x186 0x41????
```

For the next (and *final* task) we are going to monitor the number of L2 cache misses. Look for the **L2_RQSTS.MISS** event in table 19-3 or 19-11 (depending on CPU version id) in the Intel manual and set the last two bytes (the unit mask and event select) accordingly. If the operation is successful and the counters have started, you should start seeing non-zero values in the PMC0 register, increasing in subsequent reads.
[10p] Task E - Ring3 cache performance evaluation

As of now, we should be able to modify the **CR4** register with the kernel module, enable all counters in the **IA32_PERF_GLOBAL_CTRL** across all cores and start an **L2 cache miss** counter again, across all cores. What remains is putting everything into practice.
Take *mat_mul.c*. This program may be familiar from an ASC laboratory but, in case it isn't, the gist of it is that when using the naive matrix multiplication algorithm (O(n^3)), the frequency with which each iterator varies can wildly affect the performance of the program. The reason behind this is (in)efficient use of the CPU cache. Take a look at the following snippet from the source and keep in mind that each matrix buffer is a continuous area in memory.

```
for (uint32_t i=0; i<N; ++i)            /* line   */
    for (uint32_t j=0; j<N; ++j)        /* column */
        for (uint32_t k=0; k<N; ++k)
            r[i*N + j] += m1[i*N + k] * m2[k*N + j];
```

What is the problem here? The problem is that i and k are multiplied with a large number N when updating a certain element. Thus, fast variations in these two indices will cause huge strides in accessed memory areas (larger than a cache line) and will cause unnecessary cache misses. So what are the best and worst configurations for the three fors? The best: i, k j. The worst: j, k, i. As we can see, the configurations that we will monitor in *mat_mul.c* do not coincide with the aforementioned two (so… not great, not terrible.) Even so, the difference in execution time and number of cache misses will still be significant.
Which brings us to the task at hand: using the **RDPMC** instruction, calculate the number of L2 cache misses for each of the two multiplications without performing any context switches (hint: look at gcc extended asm and the following macro from *mat_mul.c*).

```
#define rdpmc(ecx, eax, edx)    \
    asm volatile (              \
        "rdpmc"                 \
        : "=a"(eax),            \
          "=d"(edx)             \
        : "c"(ecx))
```

A word of caution: remember that each logical core has its own PMC0 counter, so make sure to use taskset in order to set the CPU affinity of the process. If you don't the process may be passed around different cores and the counter value becomes unreliable.

```
$ taskset 0x01 ./mat_mul 1024
```