



BAHIR DAR UNIVERSITY
BAHIR DAR INSTITUTE OF TECHNOLOGY
FACULTY OF COMPUTING
DEPARTMENT OF SOFTWARE ENGINEERING
OPERATING SYSTEM AND SYSTEM PROGRAMMING
INDIVIDUAL ASSIGNMENT

SYSTEM CALL:

**int perf_event_open(struct perf_event_attr *attr, pid_t pid, int cpu, int group_fd,
unsigned long flags)**

NAME=Betelhem Belete

ID Number=BDU 1306134

Submitted to: Lec Wendemu Baye

Submission date:19/11/2014EC

1. What / Why / How, this system call?

➤ what:

A call to `perf_event_open()` creates a file descriptor that allows measuring performance information. Each file descriptor corresponds to one event that is measured; these can be grouped together. `perf` provides access to the Performance Monitoring Unit in the CPU, and thus allows us to have a close look at the behavior of the hardware and its associated events. In addition, it can also monitor software events, and create reports out of the data that is collected. either to measure multiple events simultaneously.

➤ Why:?

Perf- A Performance Monitoring and Analysis Tool for Linux

When we talk of performance in computing, we refer to the relationship between our resources and the tasks that they allow us to complete in a given period of time.

In a day of high competition between companies, it is important that we learn how to use what we have at the best of its capacity. The waste of hardware or software resources, or the lack of ability to know how to use them more efficiently, ends up being a loss that we just can't afford if we want to be at the top of our game.

At the same time, we must be careful to not take our resources to a limit where sustained use will yield irreparable damage.

In this article we will introduce you to a relatively new performance analysis tool and provide tips that you can use to monitor your Linux systems, including hardware and applications. This will help you to ensure that they operate so that you are capable to produce the desired results without wasting resources or your own energy.

➤ How:

Perf works on the Model Specific Registers of your CPU for measurements like cycles or branch-misses or so. A special Part called PMU(Performance Measurement Unit) is counting all kinds of events.

So if you measure just a few features of your program, there is actually no overhead, because the CPU's PMU works independently from the actual computation.

If you exceed the Register count of your PMU, the measurement cycles through the features to measure. Perf annotates this with [XX %].

2. Briefly describe about the list of parameters and flags

Given a list of parameters, `perf_event_open()` returns a file descriptor, for use in subsequent system calls (`read(2)`, `mmap(2)`, `prctl(2)`, `fcntl(2)`, etc.).

A call to `perf_event_open()` creates a file descriptor that allows measuring performance information. Each file descriptor corresponds to one event that is measured; these can be grouped together to measure multiple events simultaneously.

Events can be enabled and disabled in two ways: via `ioctl(2)` and via `prctl(2)`. When an event is disabled it does not count or generate overflows but does continue to exist and maintain its count value.

Events come in two flavors: counting and sampled. A counting event is one that is used for counting the aggregate number of events that occur. In general, counting event results are gathered with a

read(2) call. A sampling event periodically writes measurements to a buffer that can then be accessed via mmap(2).

parameters

The pid means process id. this argument allow specifying which process to monitor.

The cpu arguments allow specifying which CPU to monitor:

- pid == 0 and cpu == -1

This measures the calling process/thread on any CPU.

- pid == 0 and cpu >= 0

This measures the calling process/thread only when running on the specified CPU.

- pid > 0 and cpu == -1

This measures the specified process/thread on any CPU.

- pid > 0 and cpu >= 0

This measures the specified process/thread only when running on the specified CPU.

- pid == -1 and cpu >= 0

This measures all processes/threads on the specified CPU. This requires CAP_SYS_ADMIN capability or a /proc/sys/kernel/perf_event_paranoid value of less than 1.

- pid == -1 and cpu == -1

This setting is invalid and will return an error.

The group_fd argument allows event groups to be created. An event group has one event which is the group leader.

The leader is created first, with group_fd = -1.

The rest of the group members are created with subsequent perf_event_open() calls with group_fd being set to the file descriptor of the group leader.

A single event on its own is created with group_fd = -1 and is considered to be a group with only 1 member.

An event group is scheduled onto the CPU as a unit: it will be put onto the CPU only if all of the events in the group can be put onto the CPU. This means that the values of the member events can be meaningfully compared---added, divided (to get ratios), and so on---with each other, since they have counted events for the same set of executed instructions.

The flags argument is formed by ORing together zero or more of the following values:

- PERF_FLAG_FD_CLOEXEC:

This flag enables the close-on-exec flag for the created event file descriptor, so that the file descriptor is automatically closed on `execve`.

Setting the close-on-exec flags at creation time, rather than later with `fcntl`, avoids potential race conditions where the calling thread invokes `perf_event_open()` and `fcntl` at the same time as another thread calls `fork` then `execve`.

➤ `PERF_FLAG_FD_NO_GROUP`:

This flag tells the event to ignore the `group_fd` parameter except for the purpose of setting up output redirection using the `PERF_FLAG_FD_OUTPUT` flag.

➤ `PERF_FLAG_FD_OUTPUT`

This flag re-routes the event's sampled output to instead be included in the `mmap` buffer of the event specified by `group_fd`.

➤ `PERF_FLAG_PID_CGROUP`:

This flag activates per-container system-wide monitoring.

A container is an abstraction that isolates a set of resources for finer-grained control (CPUs, memory, etc.).

In this mode, the event is measured only if the thread running on the monitored CPU belongs to the designated container (cgroup).

The cgroup is identified by passing a file descriptor opened on its directory in the `cgroupfs` file system.

For instance, if the cgroup to monitor is called `test`, then a file descriptor opened on `/dev/cgroup/test` (assuming `cgroupfs` is mounted on `/dev/cgroup`) must be passed as the `pid` parameter.

cgroup monitoring is available only for system-wide events and may therefore require extra permissions.

The `perf_event_attr` structure: provides detailed configuration information for the event being created.

The fields of the `perf_event_attr` structure are described in more detail below:

`type`

This field specifies the overall event type. It has one of the following values:

`PERF_TYPE_HARDWARE`:

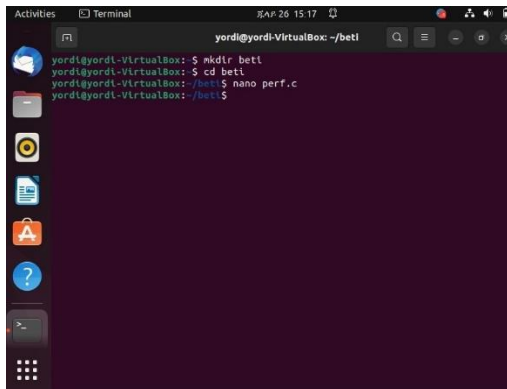
This indicates one of the "generalized" hardware events provided by the kernel. See the `config` field definition for more details.

`PERF_TYPE_SOFTWARE`:

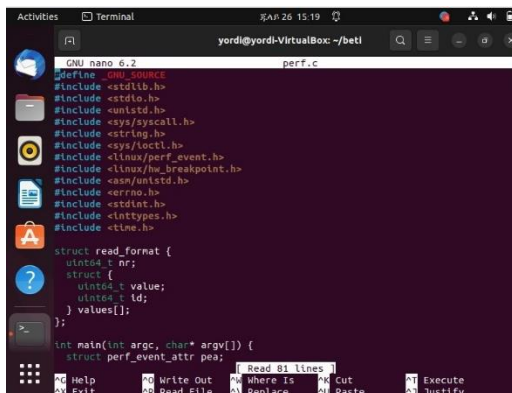
This indicates one of the software-defined events provided by the kernel (even if no hardware support is available).

3. List the flags, their purpose with code implementation (give Example source code with output).

To declare a data structure to read multiple counters at once. You have to declare different set of fields depending on what flags you pass to `perf_event_open`. In our case, we passed **PERF_FORMAT_ID** flag which adds id field. This will allow us to distinguish between different counters.



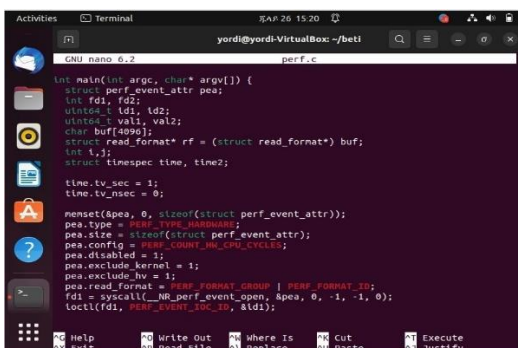
```
yordi@yordi-VirtualBox: ~/beti
yordi@yordi-VirtualBox: $ mkdir beti
yordi@yordi-VirtualBox: $ cd beti
yordi@yordi-VirtualBox: ~/beti $ nano perf.c
yordi@yordi-VirtualBox: ~/beti $
```



```
GNU nano 6.2 perf.c
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <string.h>
#include <sys/unistd.h>
#include <linux/perf_event.h>
#include <linux/hw_breakpoint.h>
#include <asm/unistd.h>
#include <errno.h>
#include <stdint.h>
#include <inttypes.h>
#include <time.h>

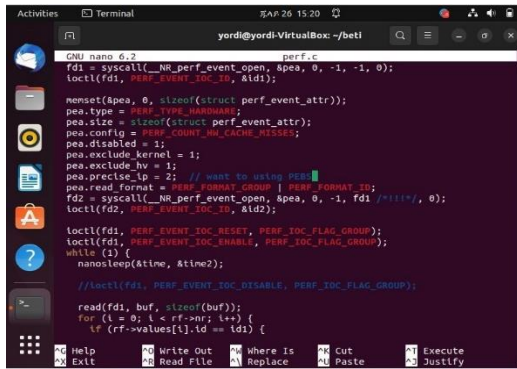
struct read_format {
    uint64_t nr;
    struct {
        uint64_t value;
        uint64_t id;
    } values[];
};

int main(int argc, char* argv[]) {
    struct perf_event_attr pea;
```



```
int main(int argc, char* argv[]) {
    struct perf_event_attr pea;
    int fd1, fd2;
    uint64_t id1, id2;
    uint64_t val1, val2;
    char buff[4096];
    struct read_format* rf = (struct read_format*) buf;
    int i, j;
    struct timespec time, time2;
    time.tv_sec = 1;
    time.tv_nsec = 0;

    memset(&pea, 0, sizeof(struct perf_event_attr));
    pea.type = PERF_TYPE_HARDWARE;
    pea.size = sizeof(struct perf_event_attr);
    pea.config = PERF_COUNT_HW_CPU_CYCLES;
    pea.disabled = 1;
    pea.exclude_kernel = 1;
    pea.exclude_hv = 1;
    pea.read_format = PERF_FORMAT_GROUP | PERF_FORMAT_ID;
    fd1 = syscall(__NR_perf_event_open, &pea, 0, -1, -1, 0);
    ioctl(fd1, PERF_EVENT_IOC_ID, &id1);
```



```
GNU nano 6.2 perf.c
fd1 = syscall(__NR_perf_event_open, &pea, 0, -1, -1, 0);
fcntl(fd1, PERF_EVENT_IOC_ID, &id1);

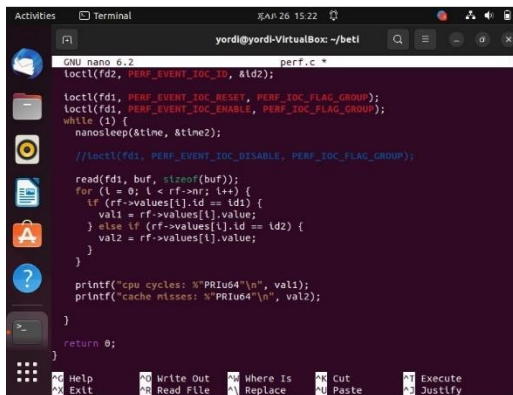
memset(&pea, 0, sizeof(struct perf_event_attr));
pea.type = PERF_TYPE_HARDWARE;
pea.size = sizeof(struct perf_event_attr);
pea.config = PERF_COUNT_HW_CACHE_MISSES;
pea.disabled = 1;
pea.exclude_kernel = 1;
pea.exclude_hv = 1;
pea.precise_ip = 2; // want to using PEBB
pea.read_format = PERF_FORMAT_GROUP | PERF_FORMAT_ID;
fd2 = syscall(__NR_perf_event_open, &pea, 0, -1, fd1, 0);
fcntl(fd2, PERF_EVENT_IOC_ID, &id2);

fcntl(fd1, PERF_EVENT_IOC_RESET, PERF_IOC_FLAG_GROUP);
fcntl(fd1, PERF_EVENT_IOC_ENABLE, PERF_IOC_FLAG_GROUP);
while (1) {
    nanosleep(&time, &time2);

    //fcntl(fd1, PERF_EVENT_IOC_DISABLE, PERF_IOC_FLAG_GROUP);

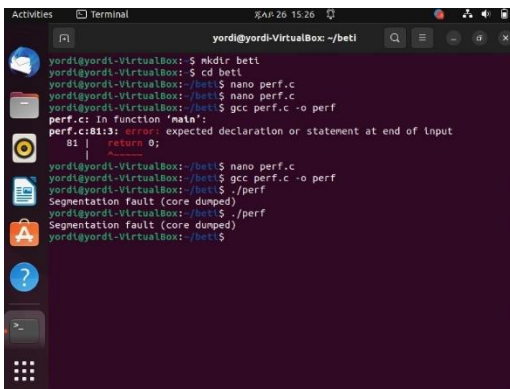
    read(fd1, buf, sizeof(buf));
    for (l = 0; l < rf->n; l++) {
        if (rf->values[l].id == id1) {

```



```
        }
        if (rf->values[l].id == id2) {
            val2 = rf->values[l].value;
        }
    }
    printf("cpu cycles: %PRIu64\n", val1);
    printf("cache misses: %PRIu64\n", val2);
}
return 0;
}
```

Press: Ctr + x and then Y to save it and press enter.



```
yordi@yordi-VirtualBox:~$ mkdir bet1
yordi@yordi-VirtualBox:~$ cd bet1
yordi@yordi-VirtualBox:~/bet1$ nano perf.c
yordi@yordi-VirtualBox:~/bet1$ gcc perf.c -o perf
perf.c: In function 'main':
perf.c:81:13: error: expected declaration or statement at end of input
81 |     return 0;
    |
yordi@yordi-VirtualBox:~/bet1$ nano perf.c
yordi@yordi-VirtualBox:~/bet1$ gcc perf.c -o perf
yordi@yordi-VirtualBox:~/bet1$ ./perf
Segmentation fault (core dumped)
yordi@yordi-VirtualBox:~/bet1$ ./perf
Segmentation fault (core dumped)
yordi@yordi-VirtualBox:~/bet1$
```

Type `gcc perf.c -o perf` then press enter. Then enter `./perf` and press enter.

It returns a message which says segmentation fault(core dumped).

First I thought this happens because perf sometimes doesn't work on OS that is installed in VM virtual boxes but that was not the problem.

The problem was Segmentation fault (core dumped) is when the system tries to access a page of memory that doesn't exist. Core dumped means when a part of code tries to perform read and write operation on a read-only or free location. This can be solved by performing the following steps command line:

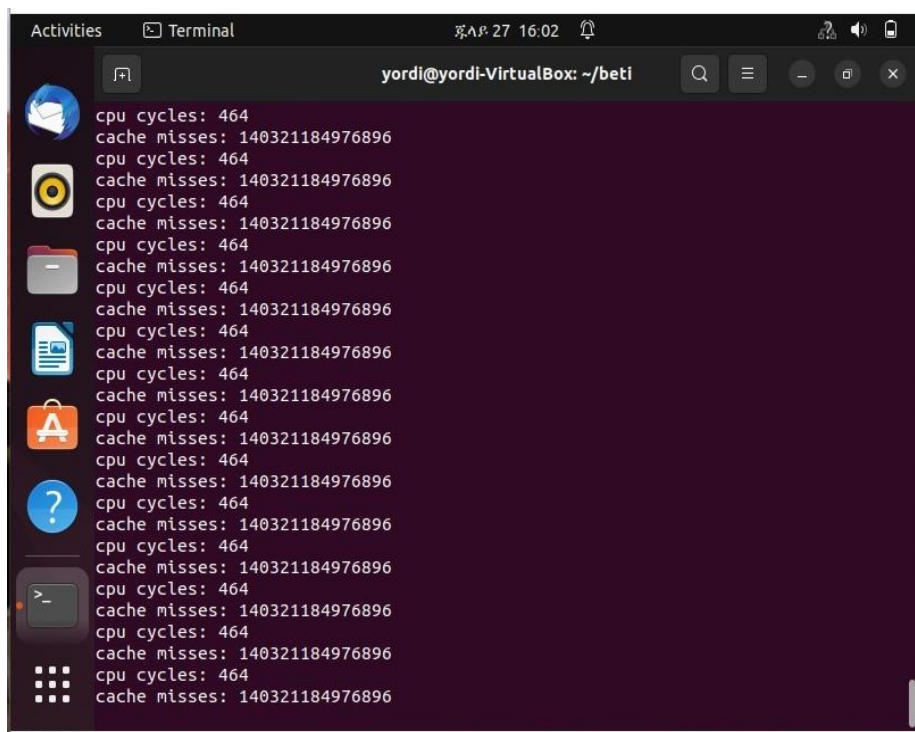
⇒ Removing the lock files present at different locations.

- ⇒ Remove repository cache.
- ⇒ Update and upgrade the repository cache.
- ⇒ Then upgrade the distribution, this will update the packages.
- ⇒ Finally find the broken packages and delete them forcefully.

After this I try to implement it and it gives me the following out put:

```
cpu cycles: 464 // Just have about 464 CPU cycles in a second
cache misses: 140321184976896 // and the specified number of data in is not currently in cache
memory.
```

[illegible]



Reference:

Linux manual page

<https://stackoverflow.com>

<https://blog.opstree.com>

<https://www.javatpoint.com>

<https://man7.org>

<https://linux.die.net>

<https://classes.engineering.wustl.edu>