CS5250 – Advanced Operating Systems
AY2019/2020 Semester 2

Assignment 2

Deadline: Monday, 1 Mar 2021 • 11.59pm

1. Objectives

1. Learn how to use ftrace to trace kernel functions.
2. Learn how to add a kernel function in kernel.

2. Rules

1. It is fine to ask for "reasonable" amount of help from others, but ensure that you do all the tasks on your own and write the report on your own. The University's policy on plagiarism applies here and any breaches will be dealt with severely.
2. For this assignment, you are asked to finish three tasks, and write a report (check assignment section for more details).
3. Generate your report as a pdf file, name it as "*Name (Student Number) Assignment 2.pdf*" and upload your report in the IVLE folder "Submissions for Assignment 2" of CS5250.
4. The deadline is 1 Mar 2021. Late assignments lose 4 marks per day.

3. Assignment Tutorial

Part A: Learn the basic usage of ftrace

The key parts are:

1. the path of ftrace
2. how are the typical tracers look like and how to set the tracer
3. how to start and stop tracing
4. how to get the trace log and analyse it

You may check the official documentary of ftrace for details. The official documentary can be found online and in your linux kernel source code whose path is ./Documentation/trace/ftrace.txt.

https://www.kernel.org/doc/Documentation/trace/ftrace.txt

It is fine for you to search and follow other instructions as long as it works.

Use ftrace to trace a specific kernel function call stack

Tracer function_graph should be used for this task. The default configuration of this tracer is to trace all the kernel functions. However, users are more interested in the function call stack of a few specific functions in practice.

To do this, you should change the *set_graph_function* to set the function and *max_graph_depth* for the function stack depth.

Part B: Adding a kernel system call

Syscall can be viewed as a big table of function entries. Each index in the table corresponds with a kernel function. In C library most of the syscall are encapsulated in functions like fopen and fclose, so you do not have to deal with software interrupt or syscall index. The Linux headers provide a syscall function so you can call to an arbitrary kernel function with an index.

To add a custom function into the kernel that outputs strings into the kernel messages, you need to modify the kernel code and system call table, recompile the kernel and use a user test program to check whether the new kernel function works.

1. Add a file in the ***kernel/*** directory under your linux kernel source file. Remember to add a print function of ftrace so that you can also use ftrace to trace your function.

   An Example:

```
#include <linux/kernel.h>   /* for printk */
#include <linux/syscalls.h>  /* for SYSCALL_DEFINE1 macro */


SYSCALL_DEFINE1(printmsg, int, i)
{
    printk(KERN_DEBUG "Hello! This is A…(Your student ID) from %d", i);
    return 1;
}
```

   You can be creative about the print message, but remember to contain your student ID. You can also create a function with 2 functions as you like as long as the basic printing function is contained.

   The *kernel.h* is needed because `printk` is called. (`printf` is not available inside the kernel. Why?) The *syscalls.h* is for the SYSCALL_DEFINE1 macro. This macro will be expand into the definition of function and a few other facility for kernel debugging. The number “1” signifies the function take one parameter. Thus if later you want to create a syscall function that takes two parameters, you need to use SYSCALL_DEFINE2.

2. Add a line ***obj-y += printmsg.o*** in the Makefile under the ***kernel/*** directory so that the kernel will add your new program into kernel image.
3. Please find the location of the table of function entries on your own and add your function at the end of the table as a new entry.
   Hint : the table starts with **syscall**
4. Recompile the kernel and boot in the new kernel as Assignment 1.
5. Build a user mode program to calls your special kernel function. Compile and run the program.

   An Example:

```
#include <linux/unistd.h>
#define __NR_printmsg    <the_index>

int printmsg(int i){
    return syscall(__NR_ printmsg, i);
}
int main(int argc, char** argv)
{
    printmsg(668);
    return 0;
}
```

   Replace the index with the index you input in step 3.
6. Use dmesg | tail to see the kernel log. Use ftrace and choose appropriate tracer to see the ftrace log. Give a screenshot of dmesg|tail and the first lines of ftrace log.

4. Tasks (20 marks)

1. 6 marks, about 1 to 2 hours
   Answer the following questions:
   a. Ftrace uses memory to keep the trace record. Before starting a new trace, the old traces are still kept in the memory by ftrace has to be emptied. How can this be done? (1 mark)
   b. What is the command to change the maximum size of the trace file of ftrace? (1 mark)
   c. Assuming you are changing the kernel code to insert a printk-like code to output hello world in ftrace and the message given by the print code is the only thing you want to trace, show your code (with comments) and also the tracer that should use in ftrace. You should provide a shell script that will run the tracing. (2 marks)

d. Use the ***function*** tracer to trace the kernel for about a few seconds. Give the screenshots of the first 20 lines of the trace file and analyse the basic structure of it. (2 marks)

2. 6 marks, about 30 minutes to 1 hour
   Use the ***function_graph*** tracer in ftrace to trace the function call stack of three kernel functions, ***vfs_open, vfs_read and vfs_write*** and set the max function call depth to record as 10. Give the complete commands you use and analyse the trace result.
   Note: the three functions should be traced **at the same time**.

3. 8 marks, about 2 to 3 hours
   a. Finish the steps for adding a kernel function into kernel in the tutorial. Supply the two C files. (2 marks)
   b. What is the full path of the system call table? Show the line you added in the system call table. (2 marks)
   c. Show the screenshot of the “`dmesg | tail`” containing the print message of your new kernel function. (2 marks)
   d. Give the first 20 lines of the trace file of ftrace and analyse it. (2 marks)

Part C: Additional exercises (5 marks each)

1. Using an example, show why deletion is not supported in the bit-vector form of the Bloom filter.
2. Consider the following simple C program compiled for a <u>32 bit x86</u> Linux machine:

```
#include <stdio.h>

int globvar = 42;

int foo(int arg)
{
    return globvar + arg;
}

main()
{
    foo(10);
    printf("%d\n", globvar);
}
```

The disassembled code for the “.o” file is:

```
[wongwf@asura ~]$ objdump -d PIC-example.o

PIC-example.o:     file format elf32-i386


Disassembly of section .text:

00000000 <foo>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8b 15 00 00 00 00       mov    0x0,%edx
   9:   8b 45 08                mov    0x8(%ebp),%eax
   c:   01 d0                   add    %edx,%eax
   e:   5d                      pop    %ebp
   f:   c3                      ret

00000010 <main>:
  10:   55                      push   %ebp
  11:   89 e5                   mov    %esp,%ebp
  13:   83 e4 f0                and    $0xfffffff0,%esp
  16:   83 ec 10                sub    $0x10,%esp
  19:   c7 04 24 0a 00 00 00    movl   $0xa,(%esp)
  20:   e8 fc ff ff ff          call   21 <main+0x11>
  25:   a1 00 00 00 00          mov    0x0,%eax
  2a:   89 44 24 04             mov    %eax,0x4(%esp)
  2e:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  35:   e8 fc ff ff ff          call   36 <main+0x26>
  3a:   c9                      leave
  3b:   c3                      ret
```

And the *relocation section* of the ".o" file is:

```
Relocation section '.rel.text' at offset 0x1f8 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000005  00000901 R_386_32          00000000   globvar
00000021  00000a02 R_386_PC32        00000000   foo
00000026  00000901 R_386_32          00000000   globvar
00000031  00000501 R_386_32          00000000   .rodata
00000036  00000c02 R_386_PC32        00000000   printf
```

(i) Show the resultant binary of the function "foo" after the first THREE (3) relocation is applied, assuming that the linker decides to allocate **`globvar`** to the address **`0x4fa80`** and **`foo`** to address **`0x80817e0`**.

(ii) What do you think the 4th relocation record is for?

(ii) Show what the GOT and PLT may look like at runtime (before the program begins execution, assuming lazy binding). Write down your assumptions about where things (such as the function **`printf`**) are allocated.

Hope you can have fun and learn something from the assignment.