Assignment2 Report

1. Using an example, show why deletion is not supported in the bit-vector form of the Bloom filter. Can you suggest a modification that would allow it?

2. Consider the following simple C program compiled for a 32 bit x86 Linux machine (64 bit just mean longer addresses with more zeroes to write down):

(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 0x4fac0 and foo to address 0x8082ab0.

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

(iii). 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.

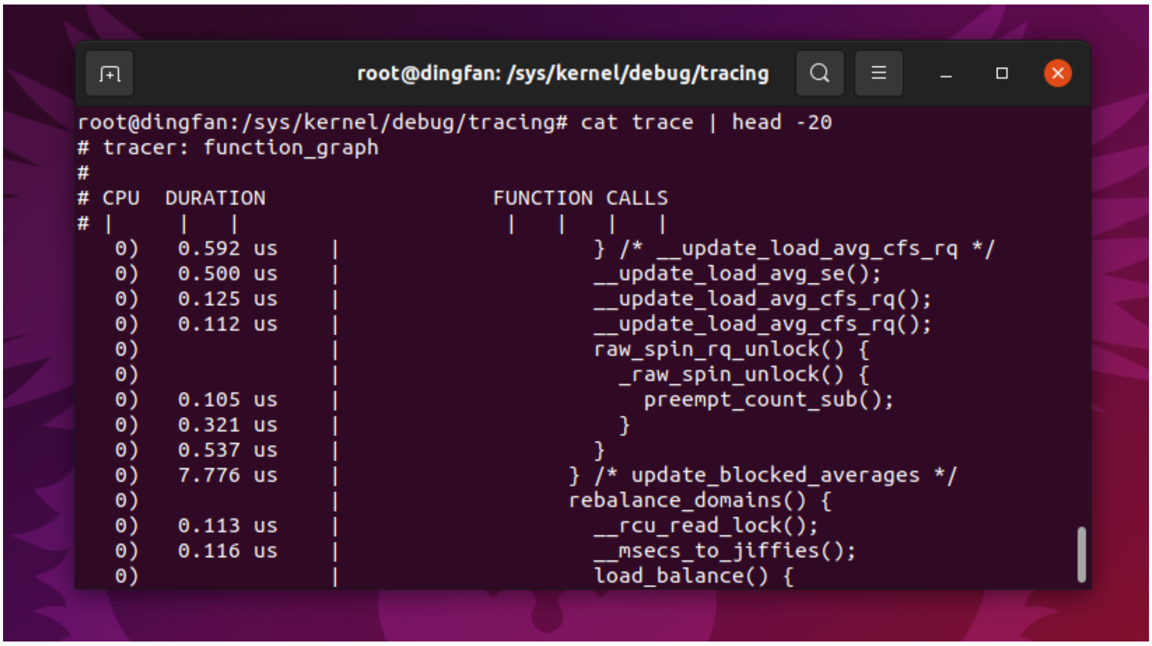
Task

Question 1: 6 marks, about 1 to 2 hours Answer the following questions:

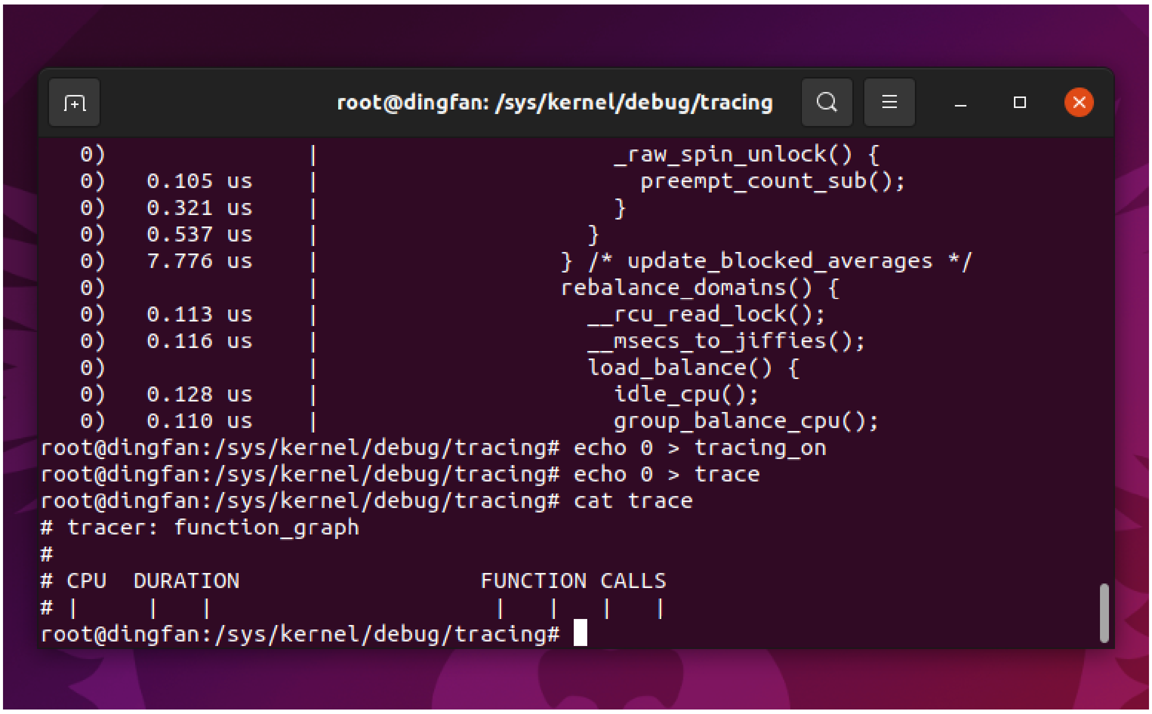
a. Ftrace uses memory to keep the trace record. Before starting a new trace, the old traces are still kept in the memory by ftrace has to be emptied. How can this be done? (1 mark)

solution:

- All the following commands are execute in the ftrace directory (`cd /sys/kernel/tracing`)
- As we can seen, before starting a new trace, the old traces are still kept in the memory.



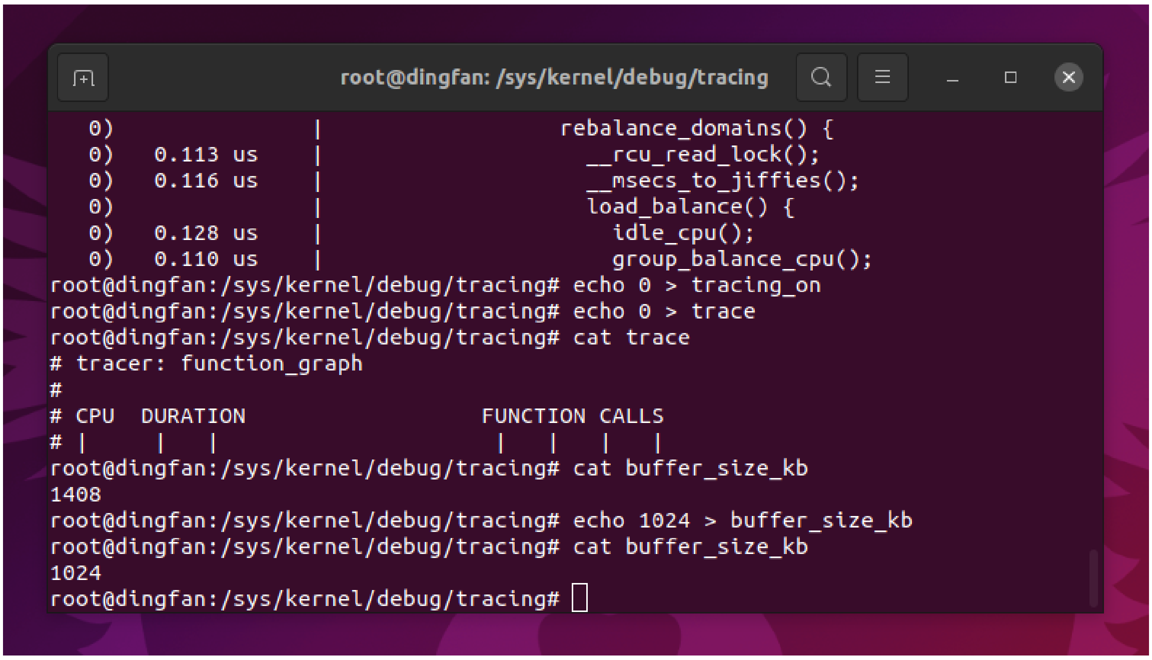
- In order to clear the old traces, it need 2 commands:
    - `echo 0 > tracing_on` : use this command to disable the tracer
    - `echo 0 > trace` : use this command to clear the old traces
- by command `cat trace` we can see that the trace file is empty



b. What is the command to change the maximum size of the trace file of ftrace? (1 mark)

solution:

- The file `buffer_size_kb` is used to modify the size of the internal trace buffers.
- `cat buffer_size_kb` to read the current value
- If we want to change the the maximum size of the trace file of ftrace, use command echo a number to the file like `echo 1024 > buffer_size_kb`
- After execute the command `echo 1024 > buffer_size_kb`, we can use the command `cat buffer_size_kb` to check the maximum size of the trace file; it is 1024 now

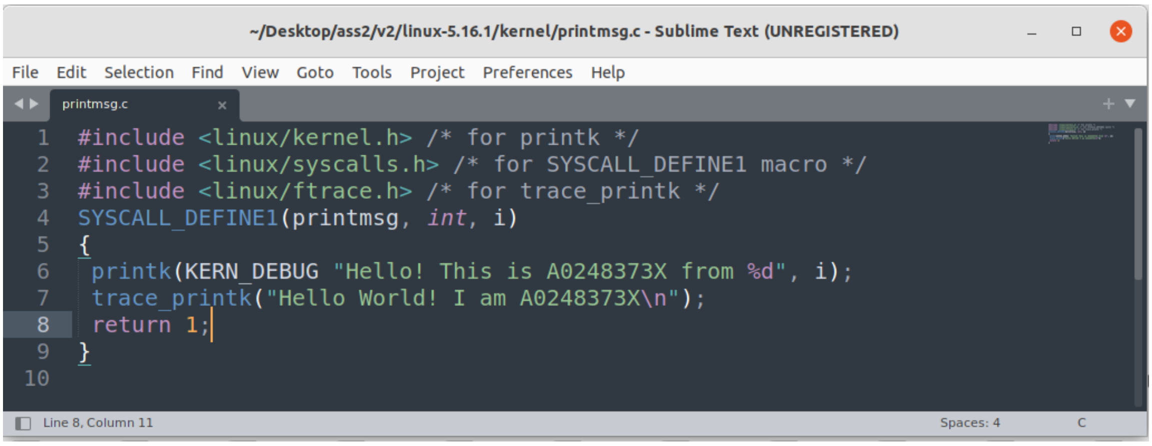


c. Assuming you are changing the kernel code to insert a printk-like code to output hello world in ftrace and the message given by the print code is the only thing you want to trace, show your code (with comments) and also the tracer that should use in ftrace. You should provide a shell script that will run the tracing. (2 marks)

solution:

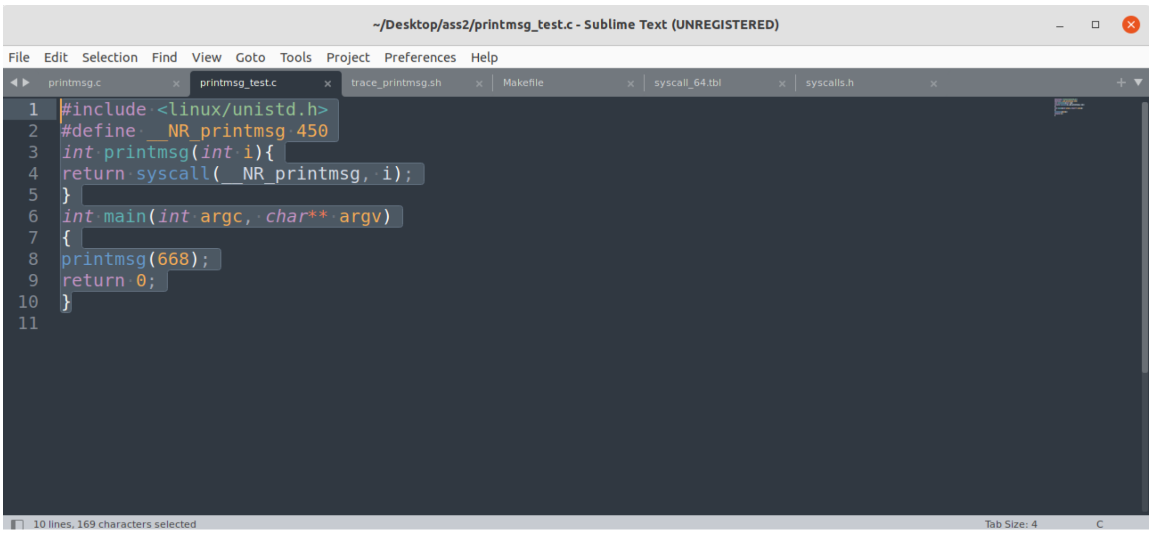
- In this question we use the new kernel function add to kernel in the tutorial as the changed kernel function
- Following is the new kernel function(which has been added to the kernel; the new kernel has been recompiled and booted as the linux system kernel)

```
#include <linux/kernel.h> /* for printk */
#include <linux/syscalls.h> /* for SYSCALL_DEFINE1 macro */
#include <linux/ftrace.h> /* for trace_printk */
SYSCALL_DEFINE1(printmsg, int, i)
{
   printk(KERN_DEBUG "Hello! This is A0248373X from %d", i);
   trace_printk("Hello World! I am A0248373X\n");
   return 1;
}
```

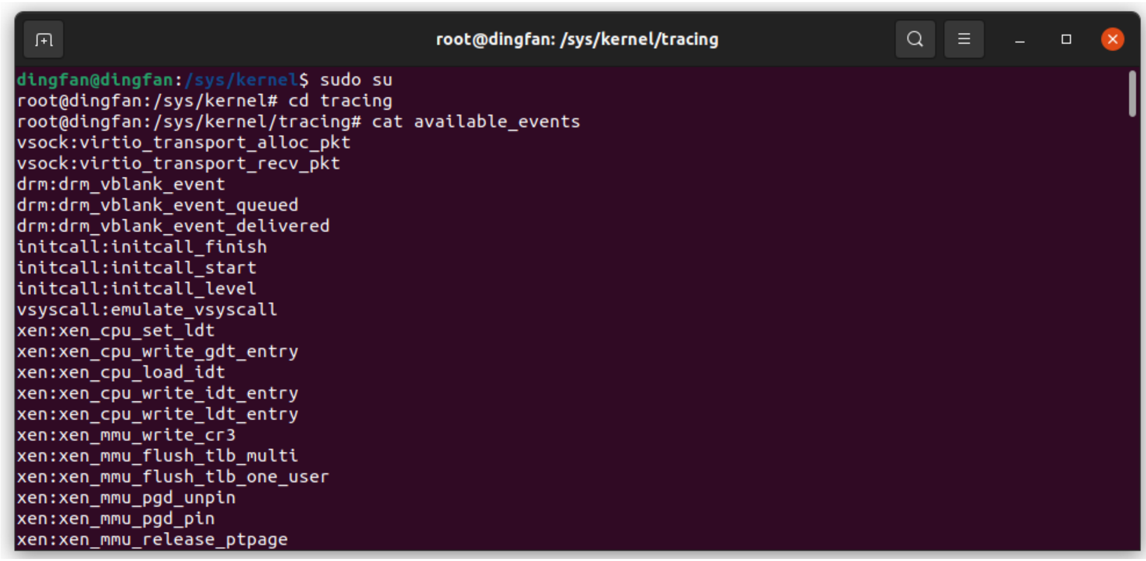


- We need a a user mode program to calls the new special kernel function
    - the user mode test program is as followed:
    - use command `gcc printmsg_test.c -o printmsg_test.o` compile the test program into the directory `“~/Desktop/ass2/”`

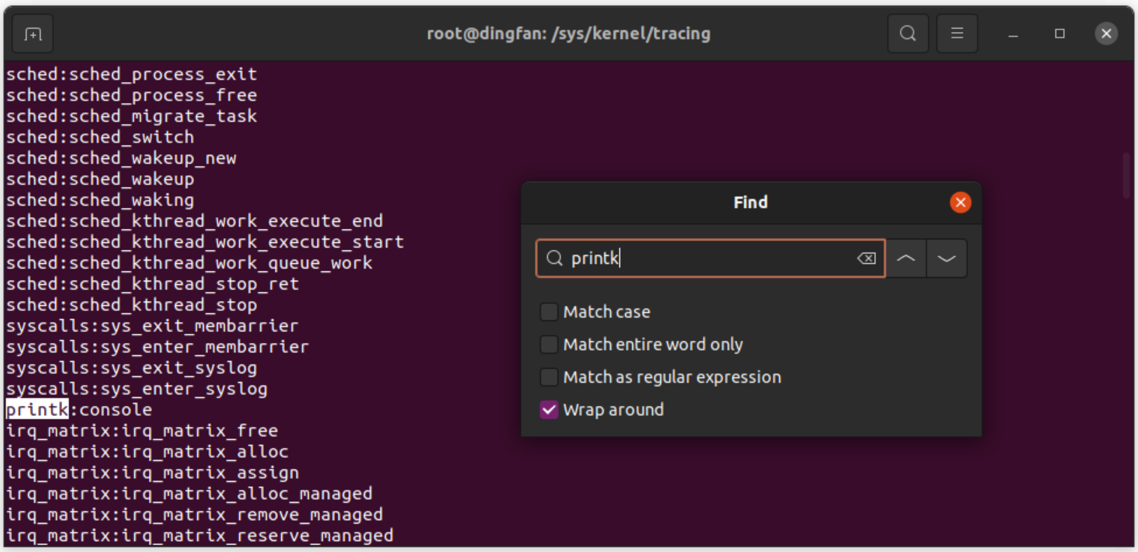
```
#include <linux/unistd.h>
#define __NR_printmsg 450
int printmsg(int i){
return syscall(__NR_printmsg, i);
}
int main(int argc, char** argv)
{
printmsg(668);
return 0;
}
```



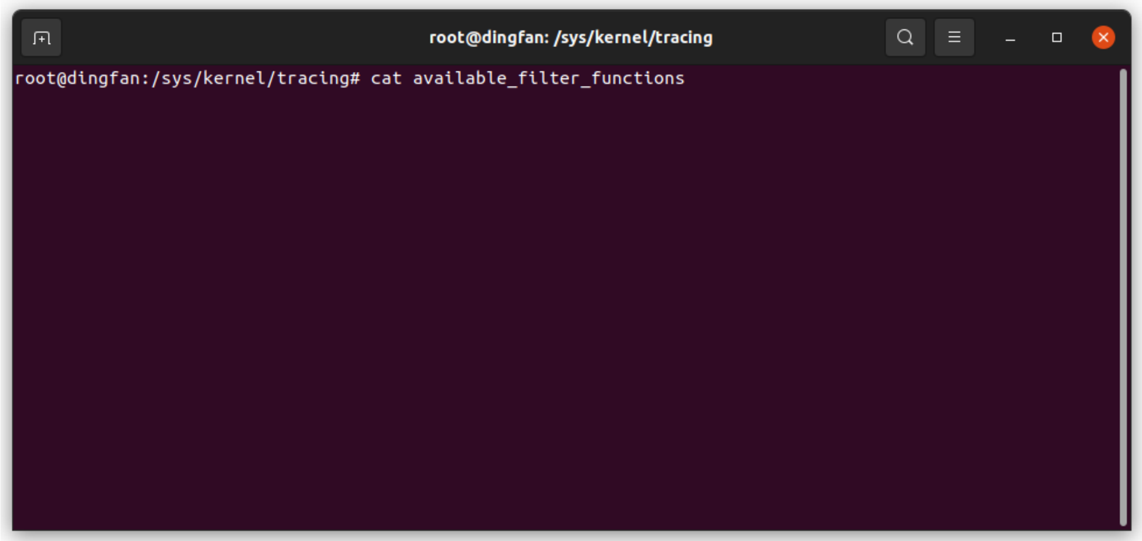
- Because the message given by the print code in function printmsg is the only thing I want to trace, we need to use the `set_event` and `set_ftrace_filter` in ftrace
- By use command `cat available_events` , I lookup the available events can be used in tracing



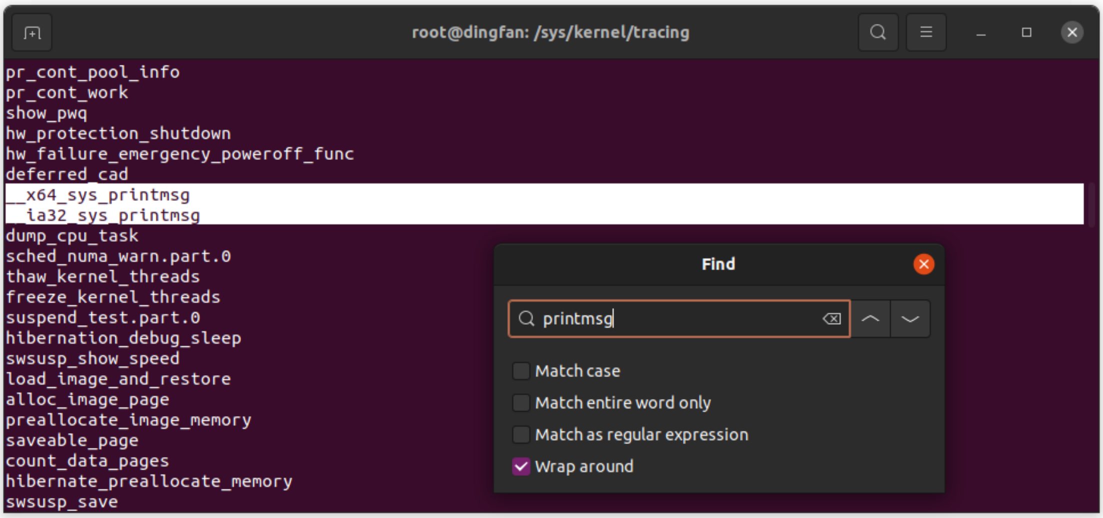
    - I need to use the event `printk`



- By use command `cat available_filter_functions`, I lookup the available function filter in ftrace



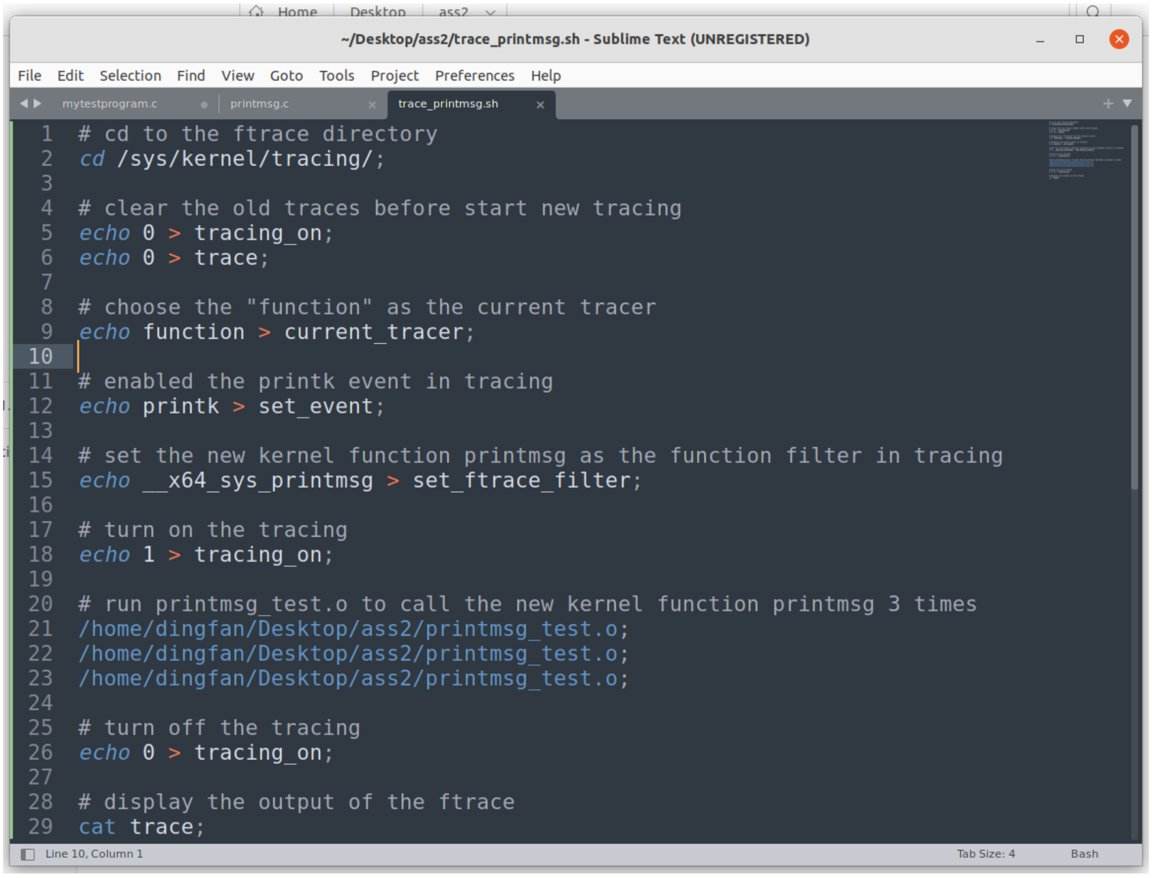
- I need to use the function filter `__x64_sys_printmsg`



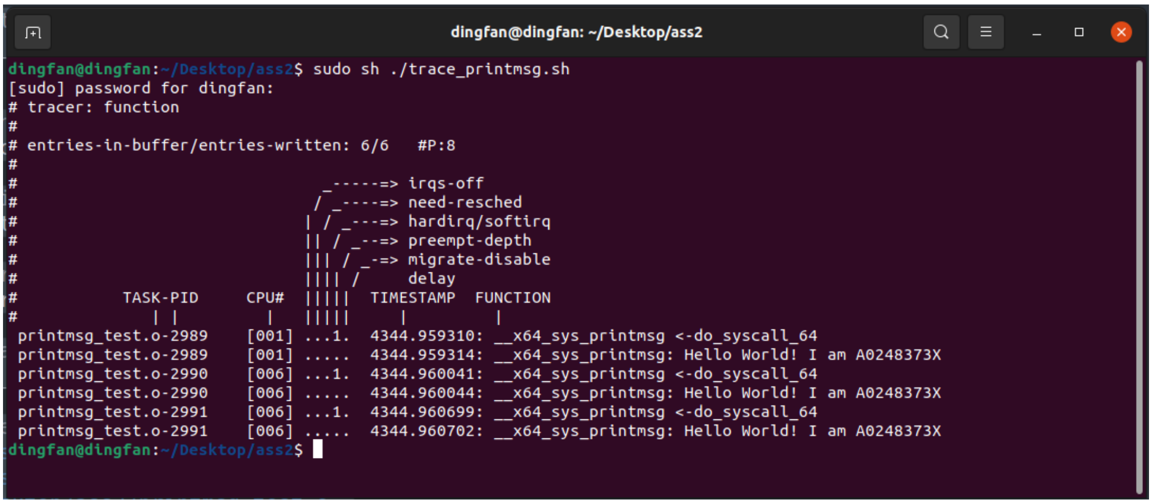
- The shell script code (with comment) that used to run the the tracing is as followed:
  - the tracer I use here is `function`

```
# cd to the ftrace directory
cd /sys/kernel/tracing/;

# clear the old traces before start new tracing
echo 0 > tracing_on;
echo 0 > trace;

# choose the "function" as the current tracer
echo function > current_tracer;

# enabled the printk event in tracing
echo printk > set_event;
```

```
# set the new kernel function printmsg as the function filter in tracing
echo __x64_sys_printmsg > set_ftrace_filter;

# turn on the tracing
echo 1 > tracing_on;

# run printmsg_test.o to call the new kernel function printmsg 3 times
/home/dingfan/Desktop/ass2/printmsg_test.o;
/home/dingfan/Desktop/ass2/printmsg_test.o;
/home/dingfan/Desktop/ass2/printmsg_test.o;

# turn off the tracing
echo 0 > tracing_on;

# display the output of the ftrace
cat trace;
```



- use command `sudo sh trace_printmsg.sh` to run the shell script for tracing
    - As I can seen, the message given by the print code in function printmsg is the only thing I want to trace

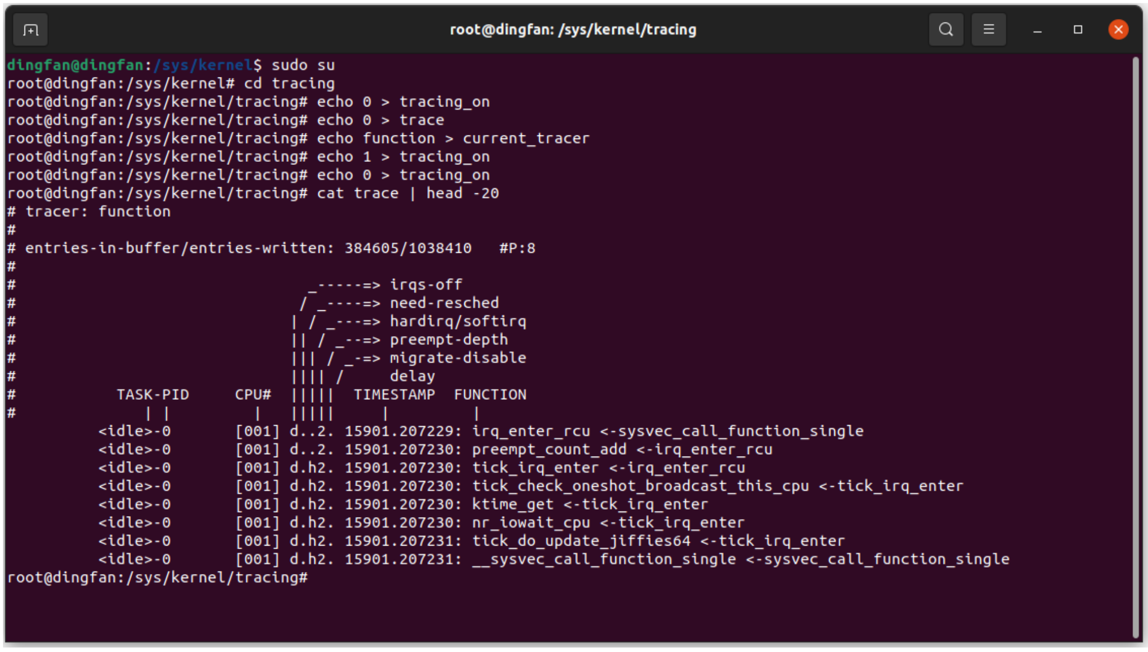


```
dingfan@dingfan: ~/Desktop/ass2
dingfan@dingfan:~/Desktop/ass2$ sudo sh ./trace_printmsg.sh
[sudo] password for dingfan:
# tracer: function
#
# entries-in-buffer/entries-written: 6/6   #P:8
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| / _-=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
 printmsg_test.o-2989    [001] ...1.  4344.959310: __x64_sys_printmsg <-do_syscall_64
 printmsg_test.o-2989    [001] .....  4344.959314: __x64_sys_printmsg: Hello World! I am A0248373X
 printmsg_test.o-2990    [006] ...1.  4344.960041: __x64_sys_printmsg <-do_syscall_64
 printmsg_test.o-2990    [006] .....  4344.960044: __x64_sys_printmsg: Hello World! I am A0248373X
 printmsg_test.o-2991    [006] ...1.  4344.960699: __x64_sys_printmsg <-do_syscall_64
 printmsg_test.o-2991    [006] .....  4344.960702: __x64_sys_printmsg: Hello World! I am A0248373X
dingfan@dingfan:~/Desktop/ass2$
```

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)

solution:

- use the function tracer to trace the kernel(all functions) for about few seconds. All the commands and the screenshots of the first 20 lines of the trace file is as followed:
  - the command `cat trace | head 20` is used to show the first 20 lines of the trace file



```
root@dingfan: /sys/kernel/tracing
dingfan@dingfan:/sys/kernel$ sudo su
root@dingfan:/sys/kernel# cd tracing
root@dingfan:/sys/kernel/tracing# echo 0 > tracing_on
root@dingfan:/sys/kernel/tracing# echo 0 > trace
root@dingfan:/sys/kernel/tracing# echo function > current_tracer
root@dingfan:/sys/kernel/tracing# echo 1 > tracing_on
root@dingfan:/sys/kernel/tracing# echo 0 > tracing_on
root@dingfan:/sys/kernel/tracing# cat trace | head -20
# tracer: function
#
# entries-in-buffer/entries-written: 384605/1038410   #P:8
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| / _-=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
          <idle>-0       [001] d..2. 15901.207229: irq_enter_rcu <-sysvec_call_function_single
          <idle>-0       [001] d..2. 15901.207230: preempt_count_add <-irq_enter_rcu
          <idle>-0       [001] d.h2. 15901.207230: tick_irq_enter <-irq_enter_rcu
          <idle>-0       [001] d.h2. 15901.207230: tick_check_oneshot_broadcast_this_cpu <-tick_irq_enter
          <idle>-0       [001] d.h2. 15901.207230: ktime_get <-tick_irq_enter
          <idle>-0       [001] d.h2. 15901.207230: nr_iowait_cpu <-tick_irq_enter
          <idle>-0       [001] d.h2. 15901.207231: tick_do_update_jiffies64 <-tick_irq_enter
          <idle>-0       [001] d.h2. 15901.207231: __sysvec_call_function_single <-sysvec_call_function_single
root@dingfan:/sys/kernel/tracing#
```

- **analyse the basic structure: of the first 20 lines of the trace file**

1. `tracer: function` :indicate the current tracer is the function, it depends on what kind of tracer I use when tracing the kernel, it can also set as other tracer like function_graph

2. `entries-in-buffer/entries-written:3846005/1038410 #p:8` : show the total number of events in the buffer(`entries-in-buffer`) which is 384605 here; it also show the total number of entries that were written in buffer (`entries-written:3846005`) which is 1038410 here; what's more The difference (between `entries-in-buffer` and `entries-written`) is the number of entries that were lost due to the buffer filling up (384605 - 1038410 = 110200 events lost).

3. `TASK-PID` : is part of the header (The header explains the content of the events.), Task indicate the tracing name of the task process; PID indicate its Process Identifier(PID) of the process.

4. `CPU`: indicate the CPU which the process was running on.

5. the latency format:

    a. `irqs-off`: it indicate whether the interruption is disabled when tracing. 'd' interrupts are disabled. '.' otherwise.

    b. `need-resched`: it indicate the information of schedule.'N' both TIF_NEED_RESCHED and PREEMPT_NEED_RESCHED is set, 'n' only TIF_NEED_RESCHED is set, 'p' only PREEMPT_NEED_RESCHED is set, '.' otherwise.

    c. `hardirq/softirq`: it indicate the information of interrupt mechanism: 'Z' - NMI occurred inside a hardirq 'z' - NMI is running 'H' - hard irq occurred inside a softirq. 'h' - hard irq is running 's' - soft irq is running '.' - normal context.

    d. `preempt-depth`: it is the The level of preempt_disabled

6. `Timestamp`: is the time at which the function was entered. the timestamp is in <secs>

7. `Function`: indicate the function name which is traced; It also indicate which function call this function. For example: `irq_enter_cpu <- sysvec_call_function_single` means the function `irq_enter_cpu` is called by `sysvec_call_function_single`

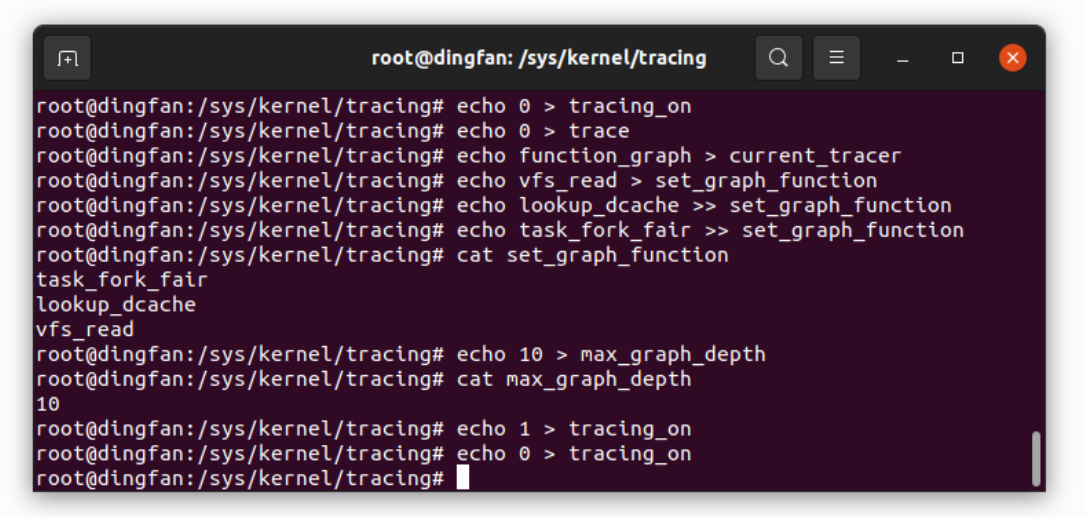
Question2: 6 marks, about 30 minutes to 1 hour

Use the function_graph tracer in ftrace to trace the function call stack of three kernel functions, vfs_read, lookup_dcache and task_fork_fair

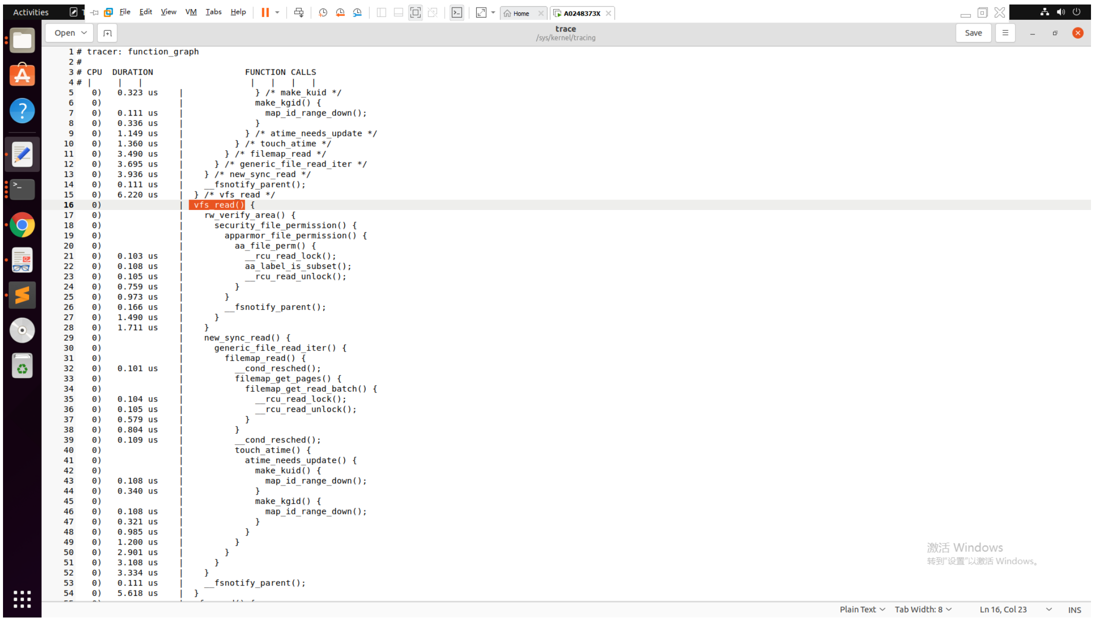
and set the max function call depth to record as 10. Give the complete commands you use and analyse the trace result. Note: the three functions should be traced at the same time.

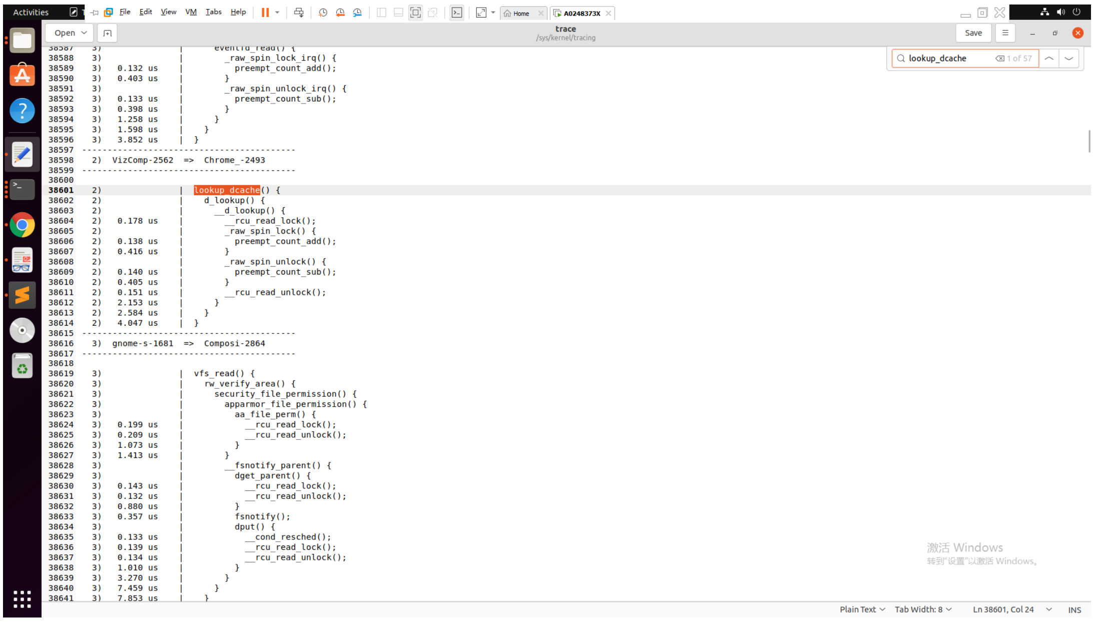
solution:

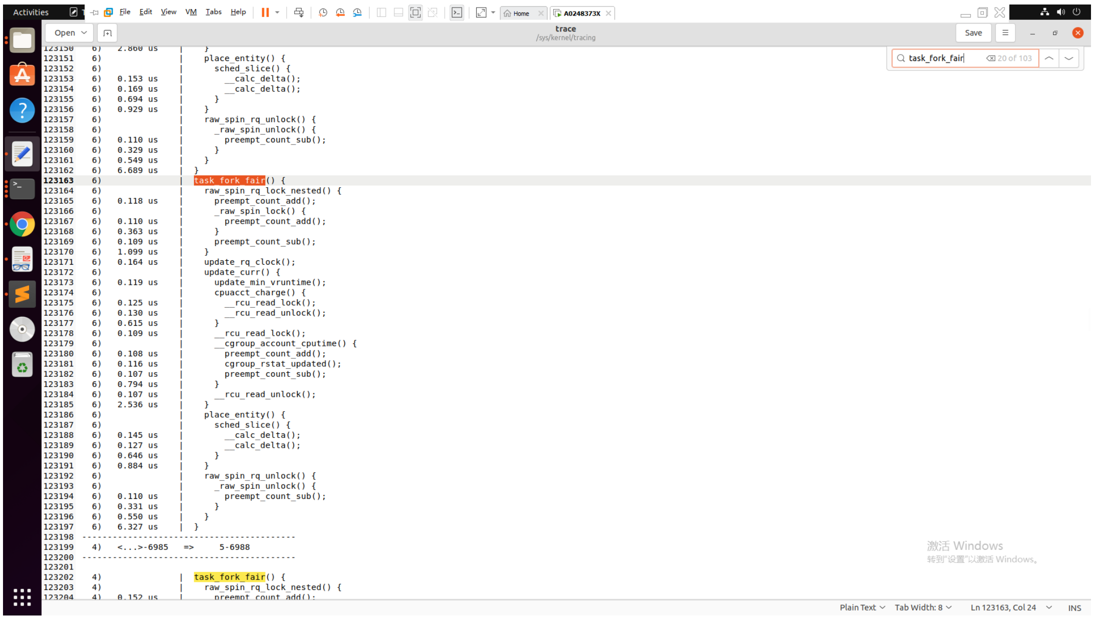
- step0: `cd sys/kernel/tracing` cd to the ftrace directory
- step1:
  - `echo 0 > tracing_on` to turn off the old tracing
  - `echo 0 > trace` to clean the old traces
- step2: `echo function_graph > current_tracer` set function_graph as the current tracer
- step3: set set_graph_function to only trace vfs_read, lookup_dcache, task_fork_fair 3 functions and the functions they call
  - `echo vfs_read > set_graph_function` add the vfs_read to set_graph_function
  - `echo lookup_dcache >> set_graph_function` append the lookup_dcache to set_graph_function
  - `echo task_fork_fair >> set_graph_function` append the task_fork_fair to set_graph_function
  - can use the command `cat set_graph_function` to check the setting of set_graph_function
- step4: set the max function call depth to record as 10
  - `echo 10 > max_graph_depth`
- step5: turn on the tracing
  - `echo 1 > tracing_on`
- step6: turn off the tracing after tracing for several seconds
  - `echo 0 > tracing_on`



- step7: display the trace result
    - `cat trace` or `gedit trace`
- the trace result
    - As I can seen in the following screenshot: the three kernel functions **vfs_read, lookup_dcache and task_fork_fair** we want to trace in ftrace are in the trace file







- analyse the trace result:
  - the first line: `tracer: function` :indicate the current tracer is the function_graph
  - `CPU` : indicate the CPU which the process was running on.
  - `Duration` : indicate the function's time of execution
  - `Function calls` : indicate both entry and exit of the functions that we want to trace: this column also includes the comments print by frace_printk

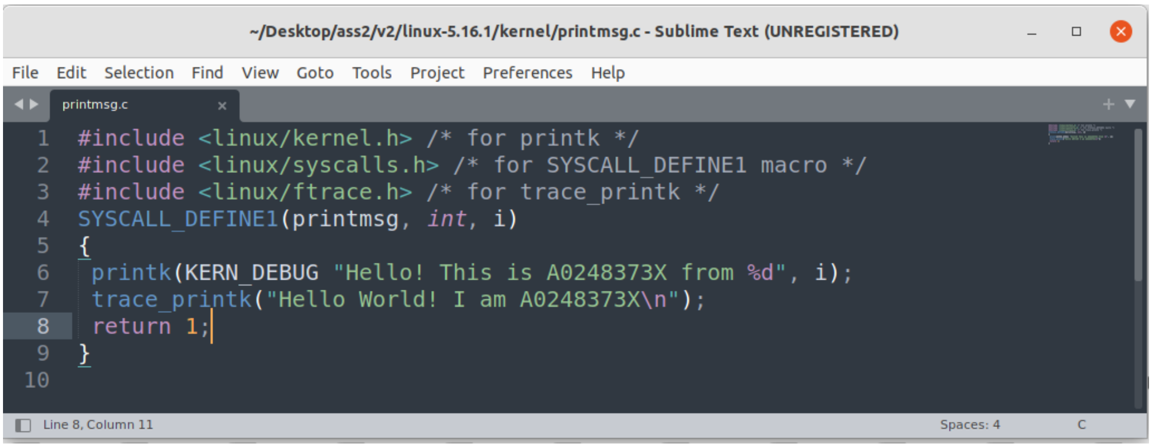
Question 3: 8 marks, about 2 to 3 hours

a. Finish the steps for adding a kernel function into kernel in the tutorial. Supply the two C files. (2 marks)

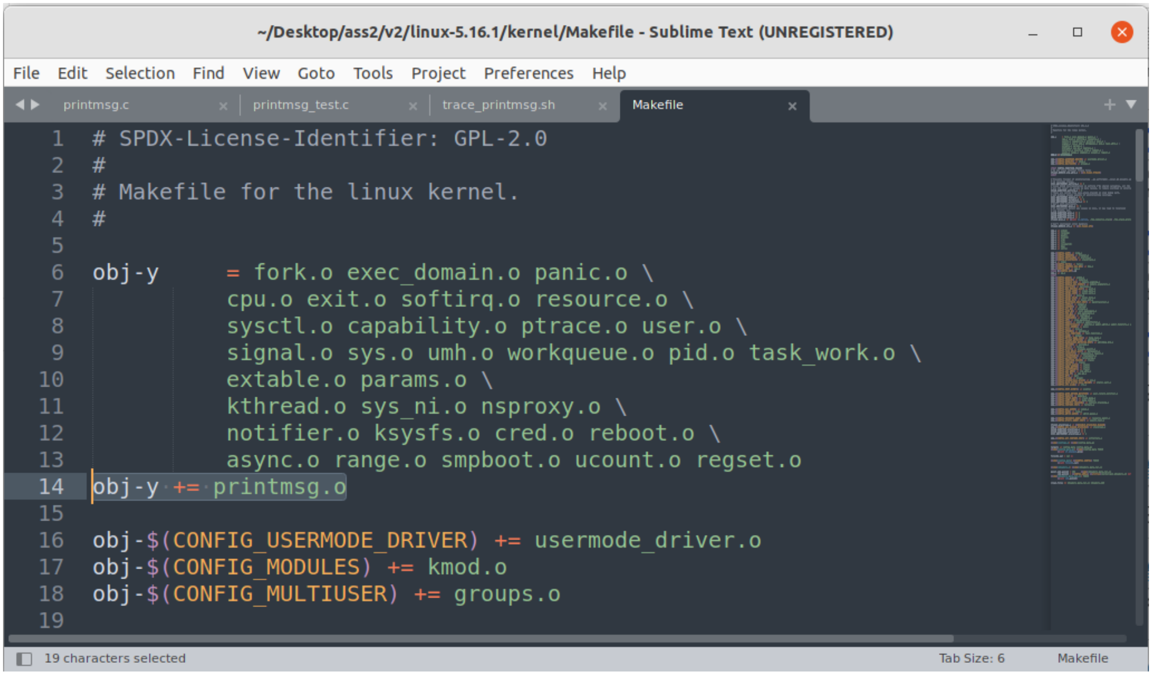
solution:

- step1:Add a file in the kernel/ directory under the linux kernel source file
    - the name of the new kernel function is `printmsg`
    - the message include my student ID `A0248373X`

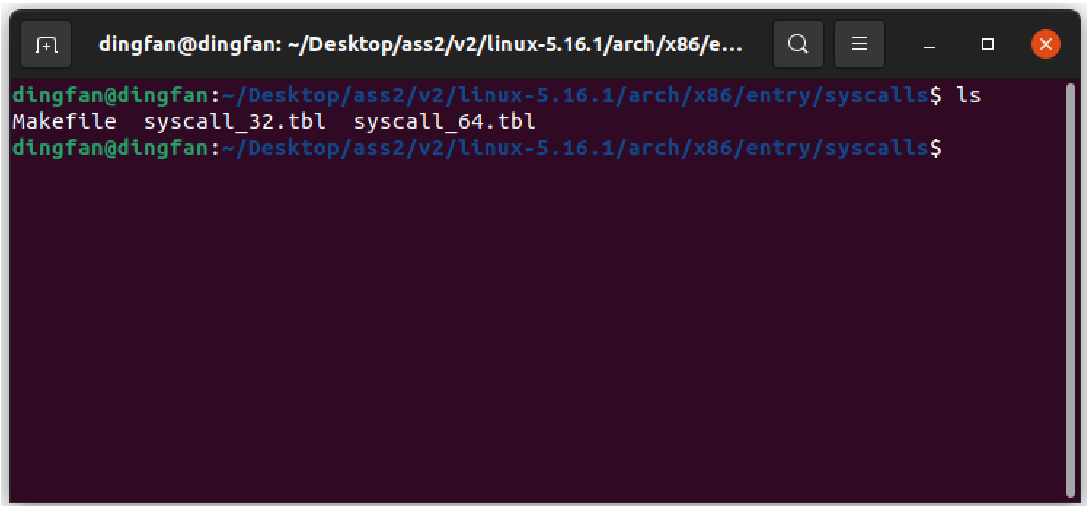
```
#include <linux/kernel.h> /* for printk */
#include <linux/syscalls.h> /* for SYSCALL_DEFINE1 macro */
#include <linux/ftrace.h> /* for trace_printk */
SYSCALL_DEFINE1(printmsg, int, i)
{
   printk(KERN_DEBUG "Hello! This is A0248373X from %d", i);
   trace_printk("Hello World! I am A0248373X\n");
   return 1;
}
```



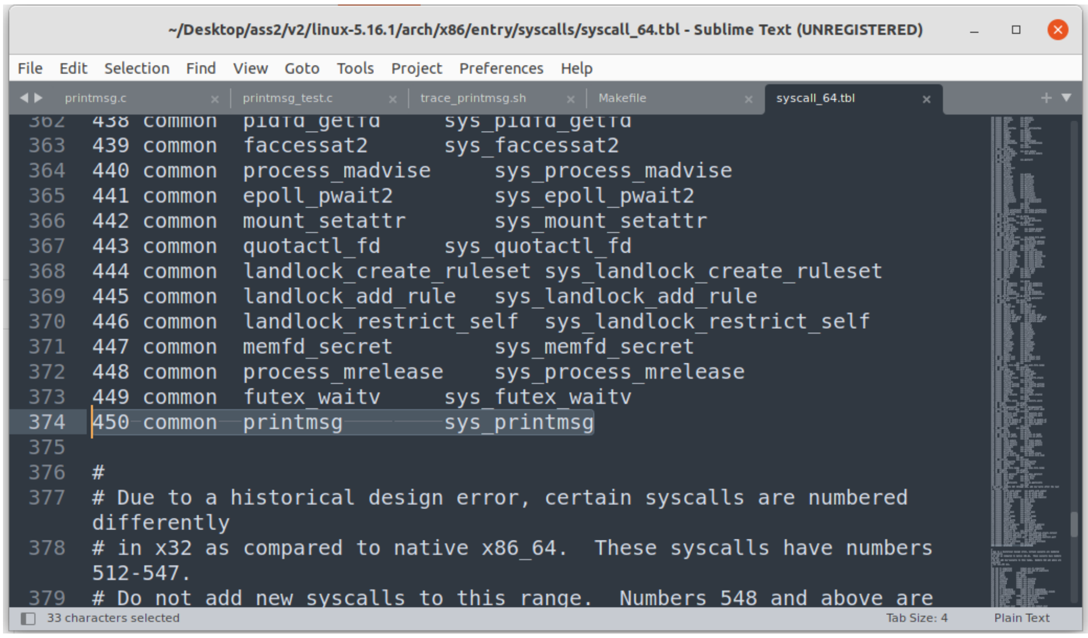
- step2: Add a line `obj-y += printmsg.o` in the Makefile under the kernel/ directory so that the kernel will add my new program into kernel image.



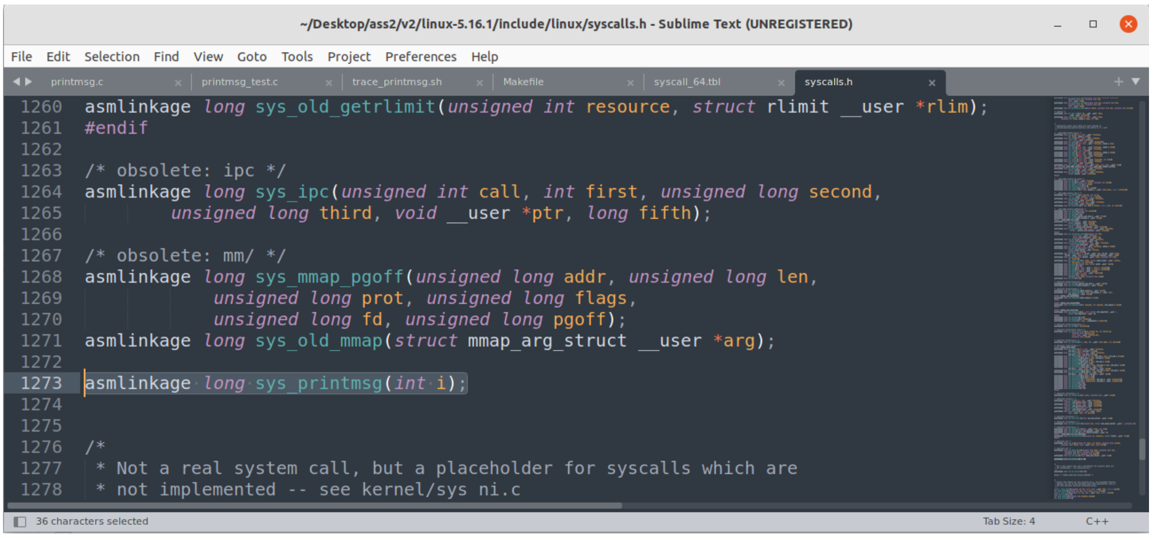
- step3:find the location of the table of function entries and add the new function at the end of the table as a new entry
    - the location of the able of function entries is `linux-5.16.1/arch/x86/entry/syscalls/syscall_64.tbl` in the linux kernel source code



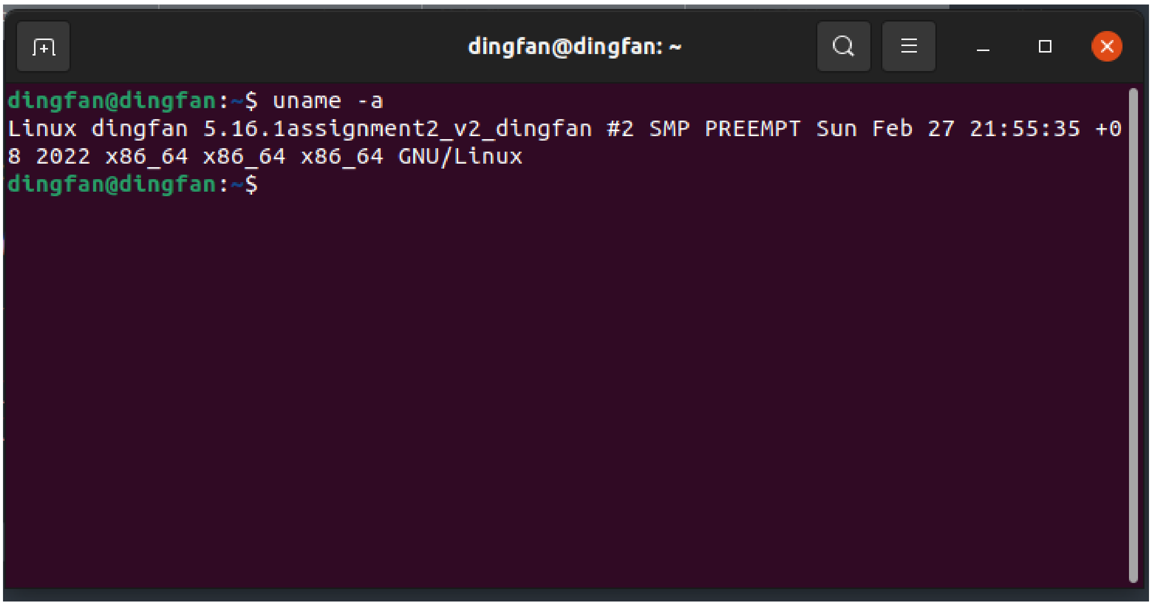
    - add a new line in as the screenshot shows: `450 common printmsg sys_printmsg`



- step4: add `asmlinkage long sys_printmsg(int i);` in file `linux-5.16.1/include/linux/syscall.h`

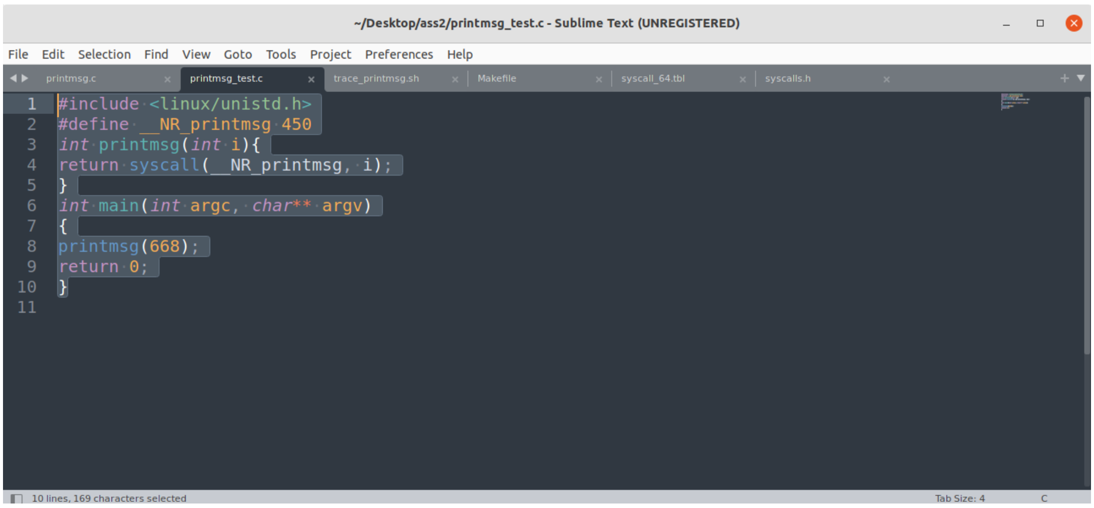


- step5: Recompile the kernel and boot in the new kernel as Assignment 1.



- step6:Build a user mode program to calls the new kernel function printmsg. Compile and run the program.
    - the user mode program is `printmsg_test`, the index is `450`

```
#include <linux/unistd.h>
#define __NR_printmsg 450
int printmsg(int i){
return syscall(__NR_printmsg, i);
}
int main(int argc, char** argv)
{
printmsg(668);
return 0;
}
```

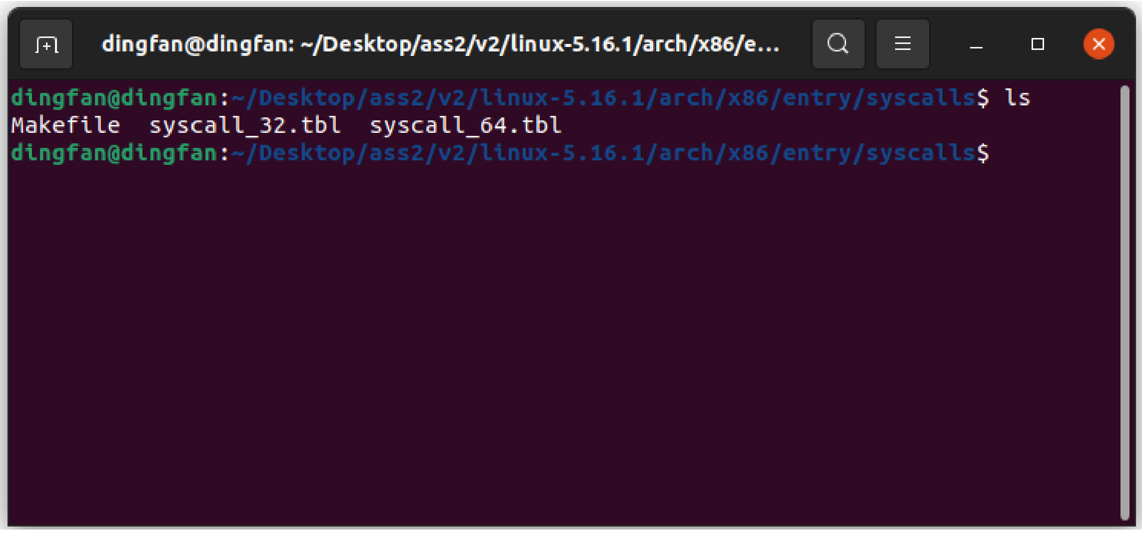


- use command `gcc printmsg_test.c -o printmsg_test.o` compile the test program into the directory `“~/Desktop/ass2/”`

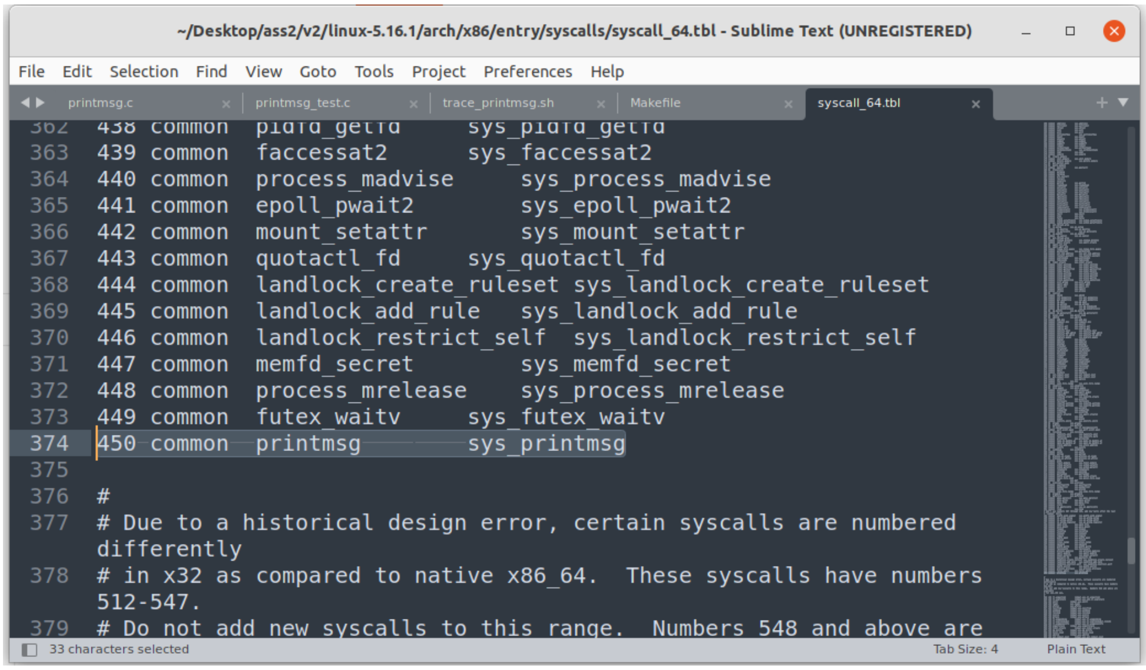
b. What is the full path of the system call table? Show the line you added in the system call table. (2 marks)

solution:

- the full path of the system call table is `linux-5.16.1/arch/x86/entry/syscalls/syscall_64.tbl`

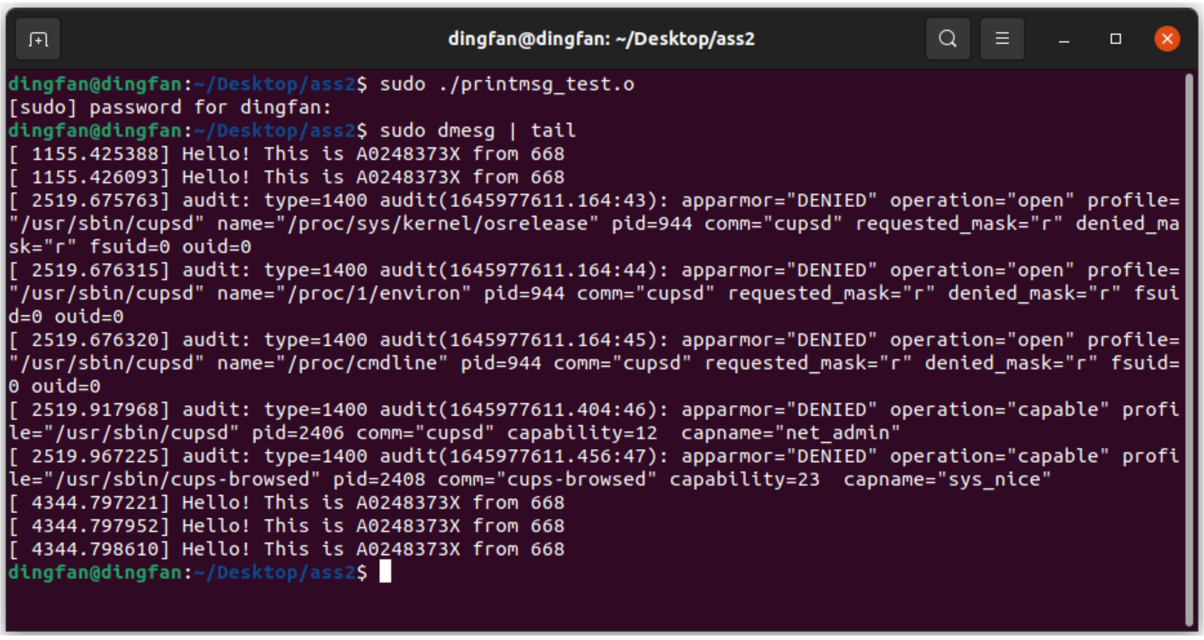


- the new line added in the system call table: `450  common  printmsg  sys_printmsg`



c. Show the screenshot of the “dmesg | tail” containing the print message of your new kernel function. (2 marks)

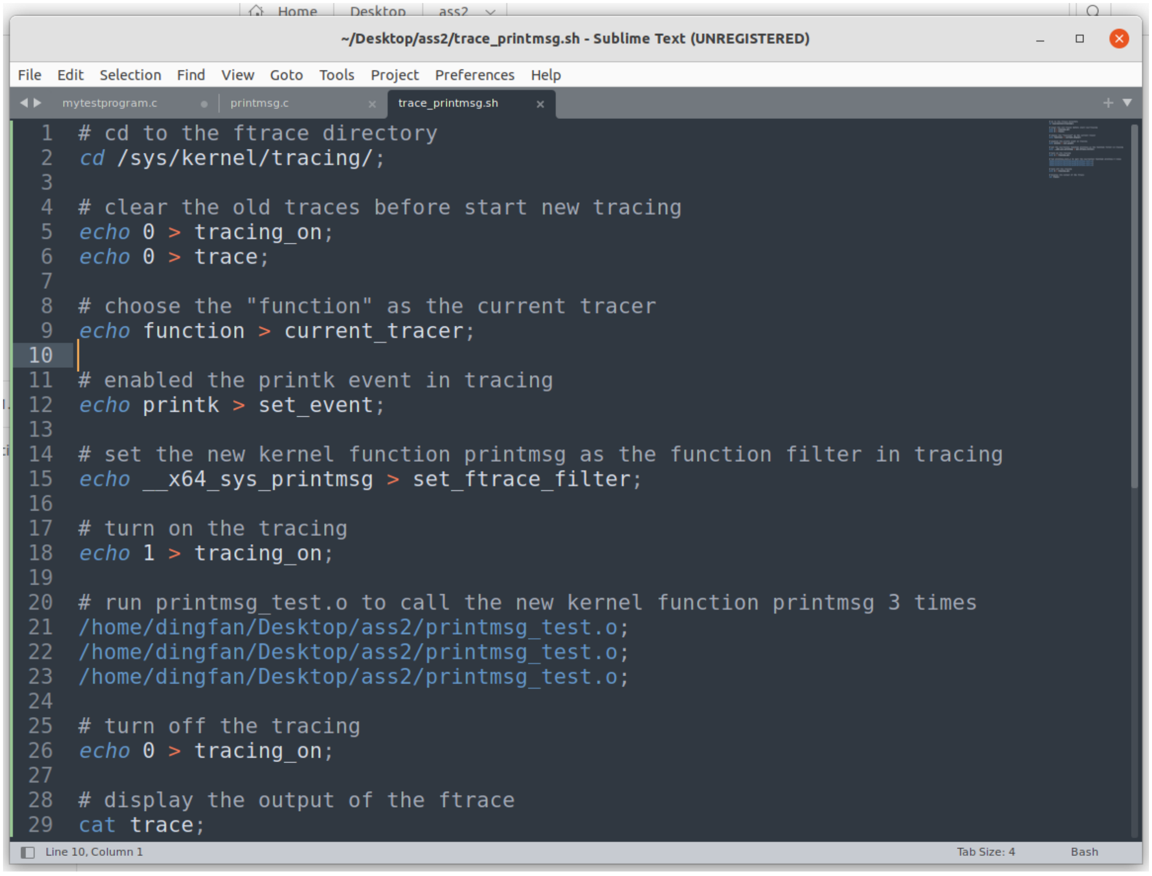
solution:



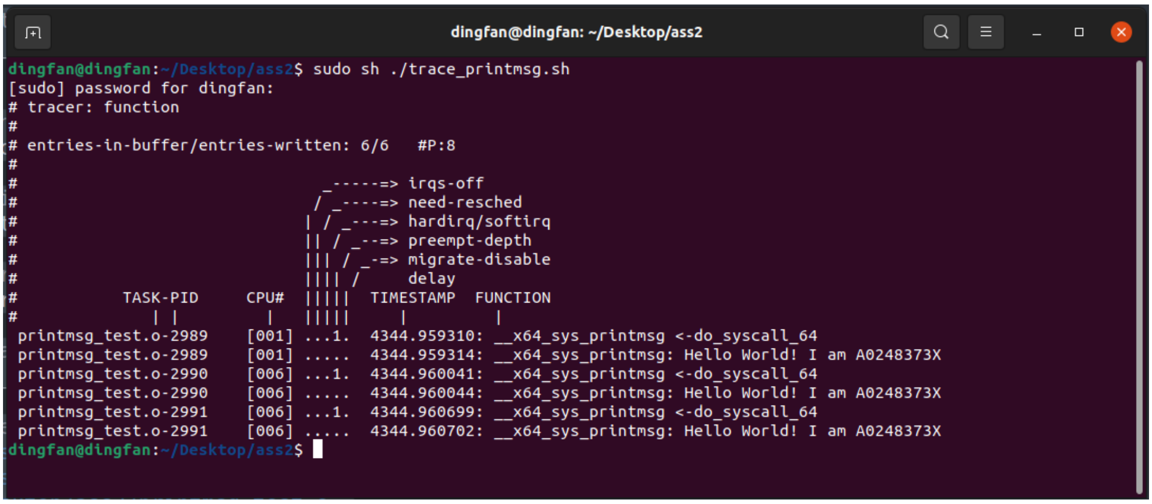
d. Give the first 20 lines of the trace file of ftrace and analyse it. (2 marks)

solution:

- **This question means use ftrace to trace the function that I inserted and called above:**
- Because I have implemented a shell script in Question 1c (which includes all the related commands). That shell script is used for tracing the function(`printmsg`) which I inserted into the new kernel
- Here we just run the shell script to see the first 20 lines of the trace files and analyse it:



```
# cd to the ftrace directory
cd /sys/kernel/tracing/;

# clear the old traces before start new tracing
echo 0 > tracing_on;
echo 0 > trace;

# choose the "function" as the current tracer
echo function > current_tracer;

# enabled the printk event in tracing
echo printk > set_event;

# set the new kernel function printmsg as the function filter in tracing
echo __x64_sys_printmsg > set_ftrace_filter;

# turn on the tracing
echo 1 > tracing_on;

# run printmsg_test.o to call the new kernel function printmsg 3 times
/home/dingfan/Desktop/ass2/printmsg_test.o;
/home/dingfan/Desktop/ass2/printmsg_test.o;
/home/dingfan/Desktop/ass2/printmsg_test.o;

# turn off the tracing
echo 0 > tracing_on;

# display the output of the ftrace
cat trace;
```

```
dingfan@dingfan:~/Desktop/ass2$ sudo sh ./trace_printmsg.sh
[sudo] password for dingfan:
# tracer: function
#
# entries-in-buffer/entries-written: 6/6   #P:8
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| / _-=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
 printmsg_test.o-2989    [001] ...1.  4344.959310: __x64_sys_printmsg <-do_syscall_64
 printmsg_test.o-2989    [001] .....  4344.959314: __x64_sys_printmsg: Hello World! I am A0248373X
 printmsg_test.o-2990    [006] ...1.  4344.960041: __x64_sys_printmsg <-do_syscall_64
 printmsg_test.o-2990    [006] .....  4344.960044: __x64_sys_printmsg: Hello World! I am A0248373X
 printmsg_test.o-2991    [006] ...1.  4344.960699: __x64_sys_printmsg <-do_syscall_64
 printmsg_test.o-2991    [006] .....  4344.960702: __x64_sys_printmsg: Hello World! I am A0248373X
dingfan@dingfan:~/Desktop/ass2$
```


- analyse the result:
  - As I can seen, the trace only has the information related to the new kernel function `printmsg` that I added to the new kernel
  - `tracer: function` : indicate the tracer I used when trace the new kernel function is `function`
  - `entries-in-buffer/entries-written:6/6` : show the total number of events in the buffer(`entries-in-buffer`) which is 6 here; it also show the total number of entries

that were written in buffer (`entries-written:3846005`) which is 6 here;

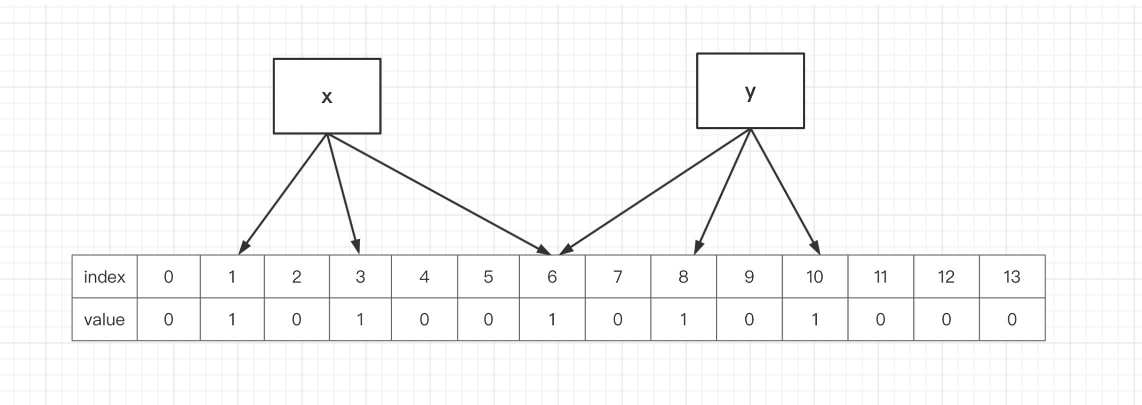
- `TASK-PID` : As the screenshot show, all the 6 trace entries belongs to the same TASK `printmsg_test.o` (which is the user mode program I write to test the new kernel function I mentioned above). The PID of that processes is 2989 2990 and 2991, because I call the `printmsg_test.o` program 3 times to test the new kernel function in my shell script as I mentioned above.
- `Function` : `__x64_sys_printmsg <- do_syscall_64` means the function `__x64_sys_printmsg` is called by `do_syscall_64` ; `__x64_sys_printmsg: Hello World! I am A0248373X` this entry is print by the t`race_printk()` in my new kernel function as I mentioned above.

Additional exercises (5 marks each)

1. Using an example, show why deletion is not supported in the bit-vector form of the Bloom filter. Can you suggest a modification that would allow it?

solution:

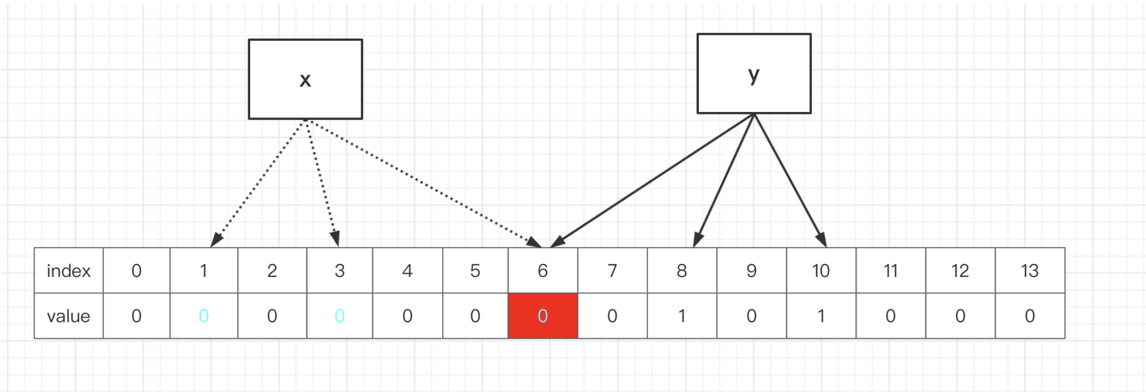
- using the following example to explain why deletion is not supported in bit vector form of the Bloom filter.



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	0	1	0	1	0	0	1	0	1	0	1	0	0	0

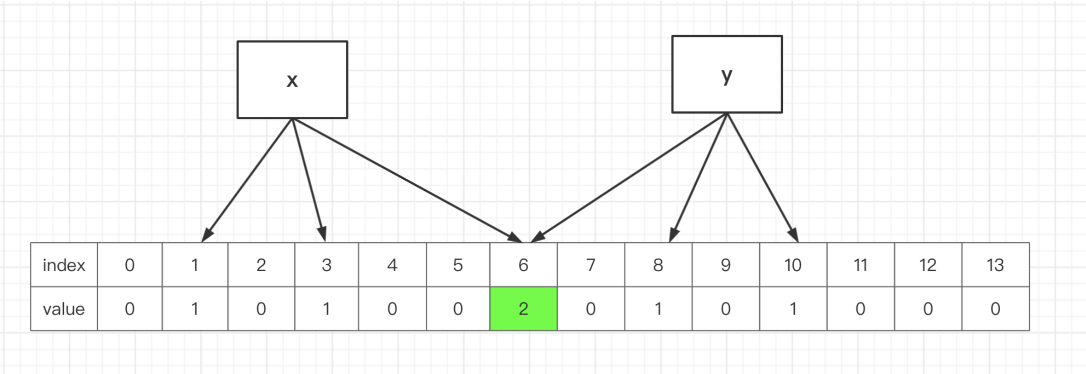
- Assume the element `x` is already in the set; and the hash value of `x` set the index: 1 ,3, 6 of the bit-vector to `1`
- Assume the element `y` is also in the set; and the hash value of `y` set the index:6, 8, 10 of the bit-vector to `1`

- Assume this set currently only have the two element x and y, all other index remain value `0`, only the value of index 1,3,6,8,10 is `1`
- In this condition if we delete element `x`, because the 3 hash index related to `x` is 1, 3, 6, we need set these three index to `0`. Than we get the following graph:



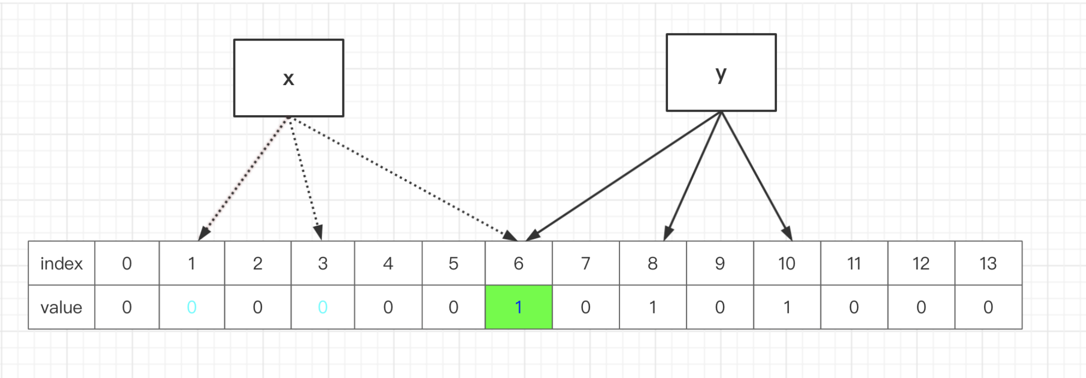
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	0	0	0	0	0	0	0	0	1	0	1	0	0	0

  - After deleting the element `x`, it may arise a problem when we loop up using the bloom filter the element `y` whether in the set.
  - The reason is one of the hash value of `y` has a collision with one of the hash value of x: the index 6, the value of index 6 was set to 0 due to the deletion of element `x`
  - When we using the bloom filter to check whether the element is in the set, the bloom filter will return: "element is not in the set" because the value of index 6 is `0`
  - According to the example we can see that deletion is not supported in bit vector form of the Bloom filter.
- Suggest a possible modification that would allow it:
  - Using the above example and make some modifications
  - Assume firstly add the value `x` to the set; set the value of index of 1,3,6 to `value = value +1` , the value of the index 1,3,6 is set to `1` from `0`
  - Then add the value `y` to the set; set the value of index 6,8,10 to `value = value +1`, the value of the index 8, 10 is set to `1` from `0`
  - I find one of the hash value of element and x has a collision with one of the element y in the index 6, so I set the value of index 6 to `2` from `1`
  - In this condition if we want to check whether an element is in the set using bloom filter, we need to check whether all the 3 hash values of that element is larger than 0 (instead of equal to1 as previous)



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	0	1	0	1	0	0	2	0	1	0	1	0	0	0

- After modification, when we delete the element `x` , we set the 3 hash values of the element x : `value = value -1`



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	0	0	0	0	0	0	1	0	1	0	1	0	0	0

- and lookup whether the element `y` is in the set now
- the bloom filter will return "the element `y` is in the set".
- now deletion is supported in the bit-vector form of the Bloom filter.

2. Consider the following simple C program compiled for a 32 bit x86 Linux machine (64 bit just mean longer addresses with more zeroes to write down):

```
#include <stdio.h>

int globvar = 42;

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

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

And the relocation section of 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 0x4fac0 and foo to address 0x8082ab0.

solution:

- After the first three relocation is applied, the resultant binary of the function “foo” is as followed:
- According to second entries in the relocation section, we need patch the value at offset 0x5 with the address of symbol foo
- Because the address is 0x8082ab0 (big endian), we need to transform into little endian in 32 bit. It is 0x **c0 fa 04 00.**

```
00000000 <foo>:
0: 55
1: 89 e5
3: 8b 15 c0 fa 04 00
9: 8b 45 08
c: 01 d0
e: 5d
f: c3
```

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

solution:

- An executable file usually has a few segments, one of the segment is the read-only one for read-only data
- The name of the 4-th relocation is .rodata, it is for the read only data “%d\n”

(iii). 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.

solution:

- step 1: when the program begin execution initially, before main call the print function:
  - GOT look like as followed:
    - GOT[0]: address of the program’s `.dynamic segment`;

      assume the address of the program’s .dynamic segment is 0x00010000
    - GOT[1]: pointer to a linked list of nodes corresponding to the symbol tables for each shared library linked with the program

      `_dl_runtime_resolve()`

      assume the address of _`dl_runtime_resolve()` is 0x00020000

        - GOT[2]: address of the symbol resolution function within the dynamic linker.

          assume the address of this is 0x00030000

        - **assume the index of printf in GOT is 50**

index	value
0	**0x 00 01 00 00**
1	**0x 00 02 00 00**
2	**0x 00 03 00 00**
...	...
50	address of “pushl 50 of PLT[1]”

    - PLT look like as followed:
        - **assume the index of printf in PLT is 1**

index	value
0	pushl GOT[1] jmpl *(GOT[2])
1	jmpl *(GOT[50]) pushl 50 jmpl PLT[0]

- step 2: when main first call the print function

  assume the allocated address of printf is 0x00040000

    - GOT look like as followed:
        - the `_dl_runtime_resolve()` will find the final allocated address of printf , and using the relocation information patch the GOT[50] as followed:

index	value
0	**0x 00 01 00 00**
1	**0x 00 02 00 00**
2	**0x 00 03 00 00**
...	...
50	0x 00 04 00 00

    - PLT look like as followed:

index	value
0	pushl GOT[1] jmpl *(GOT[2])
1	jmpl *(GOT[50]) pushl 50 jmpl PLT[0]