Assignment 2

March 2021

1 Introduction

This assignment deals with understanding the scheduling of tasks in Linux with and without Preempt-RT. Preempt-RT is the official patch to make Linux more time predictable.

Learning Objectives: Understanding Linux scheduling.

Expected deliverables.

You have to submit a report of 3 to 4 pages unless there are a lot of figures.

Preparation: In order to prepare for this lab, you have to install the following programs.

1. **KernelShark**: This is a kernal call stack tracing tool.
2. **Preempt-RT**: This is the official patch to make Linux more time predictable.

Following steps are tested on **Ubuntu 18.04 LTS (Long Term Support)**. The bold text represent the commands and the other text is to be followed in between steps. All of the referred figures are at the end of this document.

OR

Use the following virtual machine(VirtualBox)
https://surfdrive.surf.nl/files/index.php/s/2U46gdEh6uDNFHR
The username and password for login is: rts

1.1 Installing Preempt-RT Patch

Download the files **linux-5.4.19.tar.xz** and **patch-5.4.19-rt11.patch.xz** provided on the assignment page and place them in the same directory.

1. Extracting the Linux kernel.
   tar xvf linux-5.4.19.tar.xz

2. Enter the kernel directory.
   cd linux-5.4.19

3. Add patch to the kernel.
   xzcat ../patch-5.4.19-rt11.patch.xz | patch -p1 –verbose

4. Install Dependencies.
   sudo apt-get install libncurses-dev libssl-dev
   sudo apt-get install bison
   sudo apt-get install flex

5. Select Fully Preemptible Kernel (RT).
   make menuconfig
   (i) Navigate using arrow keys to General Setup, then press Enter;
   (ii) Look for Preemption Model, press Enter;
   (iii) Choose Fully Preemptible Kernel (RT) option;
   (iv) Return to the first screen by pressing Esc multiple times;
   (v) Save .config file, then exit.

6. Building the kernel. (Take a cup of coffee and wait for it to accomplish)
   make -j20

7. Install the built modules.
   sudo make modules_install -j20

8. Install the Patched Kernel
   sudo make install -j20
   If it asks to update Resume variable do steps 9 to 18 else jump to step 19.

9. Update the RESUME environment variable
   echo "RESUME=UUID=8e8a8912-d6ee-4d7b-97bc-adab4a28eed9" | sudo tee /etc/initramfs-tools/conf.d/resume
   In place of UUID replace with the UUID you see on your screen.
   sudo update-initramfs -u -k all

10. For Signing the signature of your patch, download the mokconfig.cnf file, and follow the steps below. To get a reference of the following steps click this link: https://askubuntu.com/questions/1081472/vmlinuz-4-18-12-041812-generic-has-invalid-signature.

11. Create the public and private key for signing the kernel
    openssl req -config ./mokconfig.cnf -new -x509 -newkey rsa:2048 -nodes -days 36500 -outform DER -keyout "MOK.priv" -out "MOK.der"

12. Convert the key also to PEM format (mokutil needs DER, sbsign needs PEM)
    openssl x509 -in MOK.der -inform DER -outform PEM -out MOK.pem

13. Enroll the key to your shim installation
    sudo mokutil –import MOK.der
    You will be asked for a password, you will just use it to confirm your key selection in the next step.

14. Restart your system
    Restart your system. You will encounter a blue screen of a tool called MOKManager. Select "Enroll MOK" and then "View key". Make sure it is your key you created. Afterwards continue the process and you must enter the password which you provided in previous step. Continue with booting your system.

15. Verify your key is enrolled via
    sudo mokutil –list-enrolled

16. Sign your installed kernel (it should be at /boot/vmlinuz-5.4.19-rt11)
    sudo sbsign –key MOK.priv –cert MOK.pem /boot/vmlinuz-5.4.19-rt11 –output /boot/vmlinuz-5.4.19-rt11.signed

17. Copy the initram of the unsigned kernel, so we also have an initram for the signed one.
    sudo cp /boot/initrd.img-5.4.19-rt11{,.signed}

18. Update your grub-config
    sudo update-grub

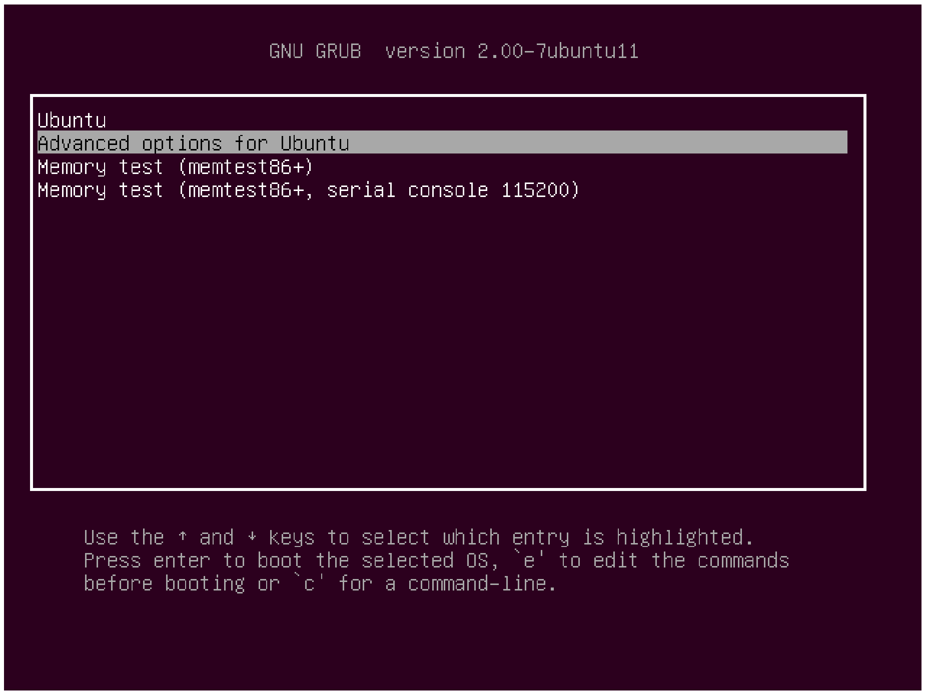
19. Reboot your system and select the signed patched kernel **(5.4.19)** by going in to the Advanced boot options of Ubuntu shown in 1

20. When the system is up and running, enter the following command in your terminal:

uname -a

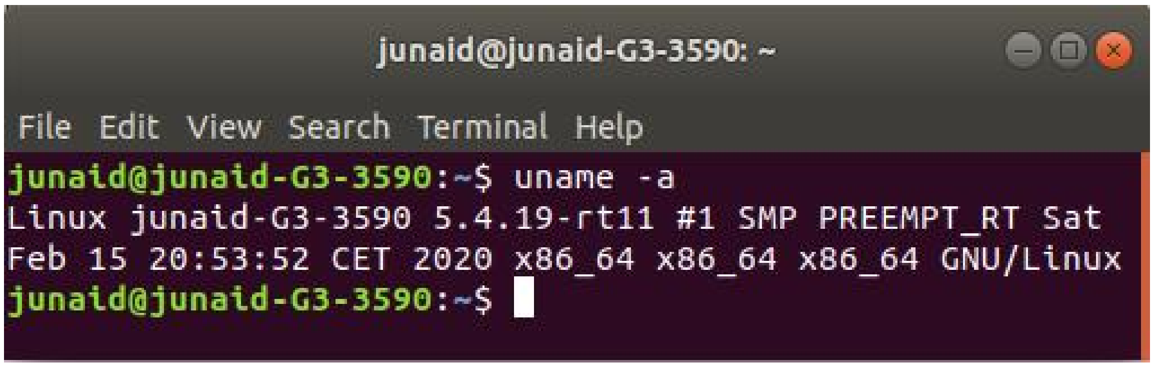
It should show you a message containing PREEMPT_RT as shown in Figure 2.

Perform the following command to reduce the Round Robin scheduling timeslice.
sudo sysctl kernel.sched_rr_timeslice_ms=xx where **xx**=milliseconds. For this assignment, it is best to keep the value of **xx**=4 millisecond. However, if you want to change the values in the experiment and want to explore more on your own, then feel free to increase its value and explain your observations.



```
                 GNU GRUB  version 2.00-7ubuntu11

 Ubuntu
 Advanced options for Ubuntu
 Memory test (memtest86+)
 Memory test (memtest86+, serial console 115200)

     Use the ↑ and ↓ keys to select which entry is highlighted.
     Press enter to boot the selected OS, `e' to edit the commands
     before booting or `c' for a command-line.
```

Figure 1: Advanced Options for Ubuntu



```
junaid@junaid-G3-3590:~$ uname -a
Linux junaid-G3-3590 5.4.19-rt11 #1 SMP PREEMPT_RT Sat
Feb 15 20:53:52 CET 2020 x86_64 x86_64 x86_64 GNU/Linux
junaid@junaid-G3-3590:~$ 
```

Figure 2: Preempt-RT Patch Confirmation

1.2 Installing KernelShark

Download the file **trace-cmd-kernelshark-v1.1.zip** provided on the assignment page.

1. Extract the contents of trace-cmd-kernelshark-v1.1.zip file.
   cd trace-cmd-kernelshark-v1.1

2. Build the source.
   sudo make

3. Build the graphical interface sources.
   sudo make gui

4. Install the built sources.
   sudo make install

5. Install the built graphical interface sources.
   sudo make install_gui

6. If above steps do not make KernelShark appear in showapplications tab, then floow the instruction given on `https://kernelshark.org/build.html`

1.3 How to create a periodic thread?

In order to create periodic real-time tasks using POSIX, follow the tutorial in this link: `http://retis.sssup.it/~lipari/courses/str09/10.rtprogramming-handout.pdf`. For a general tutorial on how to create a POSIX based application, follow the tutorial in this link: `https://computing.llnl.gov/tutorials/pthreads/`.

1.4 Creation and use of traces in KernelShark

This section explains the method to create a trace entry to be shown in KernelShark during the execution. Since KernelShark is only designed to show kernel call stacks, so it is important that any user level function or thread, which is required to be traced, makes the appropriate system calls. In the FtraceExample.c example source file provided with this assignment, in the Assignment_2_Code_Template folder, to get samples of thread creation, trace generation and CPU affinity setting.

The Figure 3 shows you the main of the example. Line 64 shows declaration of thread id variables for two threads. Line 66 opens a file from kernel debug facilities to insert markers in between program executions, which is another method of showing traces in KernelShark. If this method is not used then user has to invoke system calls, which does not mean anything except generation of a trace entry, which becomes cumbersome to manage. With this method user can insert his desired text in the traces along with variables values etc. It should be noticed that the file descriptor variable ***trace_fd*** should be declared global so that all the threads can access it. This is followed by creation of threads, which execute the respective functions, and joining of threads. The function calls ***trace_write("JATW_1")*** and ***trace_write("JATW_2")*** are used to insert marker texts which appear in KernelShark as shown in Figure 4.

Figure 5 shows the code for one of the two threads, in which line number 16 to 25 shows the method to set CPU affinity. In this case, the thread has the affinity to CPU 1 which basically means that ***Thread_1*** will only run on *CPU1*. The next step is the access to the s_mutex for writing a trace message in to the trace_marker file followed by rest of the operations. Figure 6 shows the simple function of formatting and writing a trace text in to the trace_marker. **You can use provided code template for your assignment.**

Command to create traces: In order to show the trace of any program, it is important to first create the trace. Lets build the given FtraceExample.c example to create the trace first. The command to build this simple example is:

gcc -pthread FtraceExample.c -o aout

Once the program is built successfully, use the following command to generate the trace file.

sudo trace-cmd record -p function -F ./aout

With the above command, the trace will be generated by running the program. The *-F ./aout* option runs the program for which the traces will be generated. By default the name of the trace file will be *trace.dat*. If it is not required to observe the system and shared library functions calls in the trace, rather only your very own program traces then use the following command to generate the traces.

sudo trace-cmd record -p nop -F ./aout

In order to run the above command with an input argument to the built binary simply follow this command:

sudo trace-cmd record -p nop -F ./aout FIFO

The above command runs the built binary *./aout* with *SCHED_FIFO* scheduling policy. Similarly, you can run it for *SCHED_RR* and *SCHED_OTHER* scheduling policies by passing *RR and OTHER* as arguments respectively.

Showing traces in KernelShark: Figure 7 shows an empty view of the KernelShark when it is opened. In this view, go to the top menu and select File → Open and then select the generated trace file which will show the view of Figure 8. From the drop down list of ***Column*** tab, select the option ***Info*** as it is shown in Figure 9. Then enter the text in the trace message search box next to the tab ***contains*** as it is shown in Figure 10 and press next to reach to the trace entry which is highlighted in green color. The timestamps column will provide the time at which that entry was written which will give an indication of the time when you recorded an event or start of a thread etc. Figure 9 also shows the orange and red colored bars against *CPU1* which shows the execution of ***Thread_1*** and ***Thread_2*** since their affinity was set to *CPU1* as it was shown in Figure 5.

Now that everything is set in the system, lets get on to the assignment itself. You have to use the **Assignment_2_Code_Template.c** file which is provided in the *.zip* package.

2 Main questions

Answer the following questions in a digital format and upload it into the BrightSpace. Please do not forget to add your names, student numbers, and the date of submission to the file you upload.

Question 1 (15 points). Explain in detail how SCHED_FIFO, SCHED_RR and SCHED_ DEADLINE work in Linux. You can learn about these policies by reading, for example, the following link: `http://man7.org/linux/man-pages/man7/sched.7.html`. Feel free to explore more on your own.

Question 2 (30 points). In order to understand the differences between the SCHED_FIFO, SCHED_RR and SCHED_OTHER scheduling policies, build a task set (by playing with NUM_THREADS and periods), schedule the task set with each of these policies, report your observations, compare the resulting schedules with each other, and explain the differences.

Note 1. Your answer must include explanation of what you have done, properties of the tasks you have generated (their WCET, period, priorities, utilization, etc.), the schedule visualized via KernelShark, and the KernelShark traces.

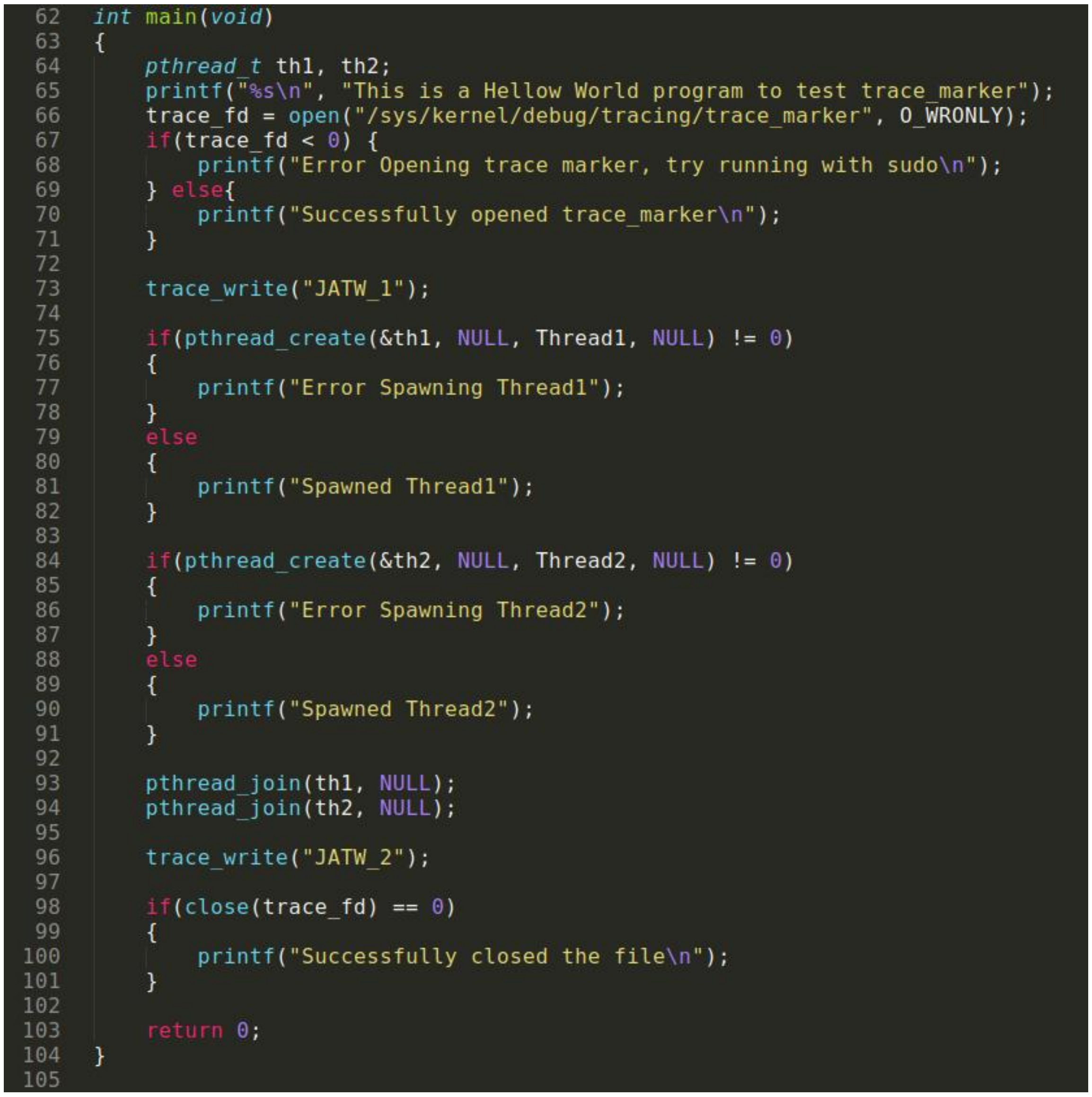
Note 2. To create a periodic task-set you can have a look at section 1.3. You need to build and run the program according to 1.4.

Note 3. Since we are using only a single CPU in this assignment, you must ensure that the total utilization of the task-set is not more than 1. You can use tracing functions to record traces whenever a thread starts or just before going to sleep to get a nice view of activities in KernelShark. Remember that it may incur some overhead, so it is up to you to decide on how you want to measure the response time. However, you must specify your decision in your report. Your report must also explain (briefly) how did you instrument the code for tracing.

Note 4. The second part of the slide 7, in this link `http://retis.sssup.it/~lipari/courses/str09/10.rtprogramming-handout.pdf`, provides a method to check the deadline miss of a task. The structure **next** is used to set the release time of the task and the structure **now** in the slide provides the finish time of the task. Similar to the deadline miss mechanism, you can take a difference between the finish time and release time of the task to measure the response time. Keep in mind that, to measure the response time you will have to use the previously set value of the **next** and not the new value which is set by the call **timespec_add_us**. You can add all of your measurements in an array and then dump it in a file, for the sake of analysis, at the end of your simulation.

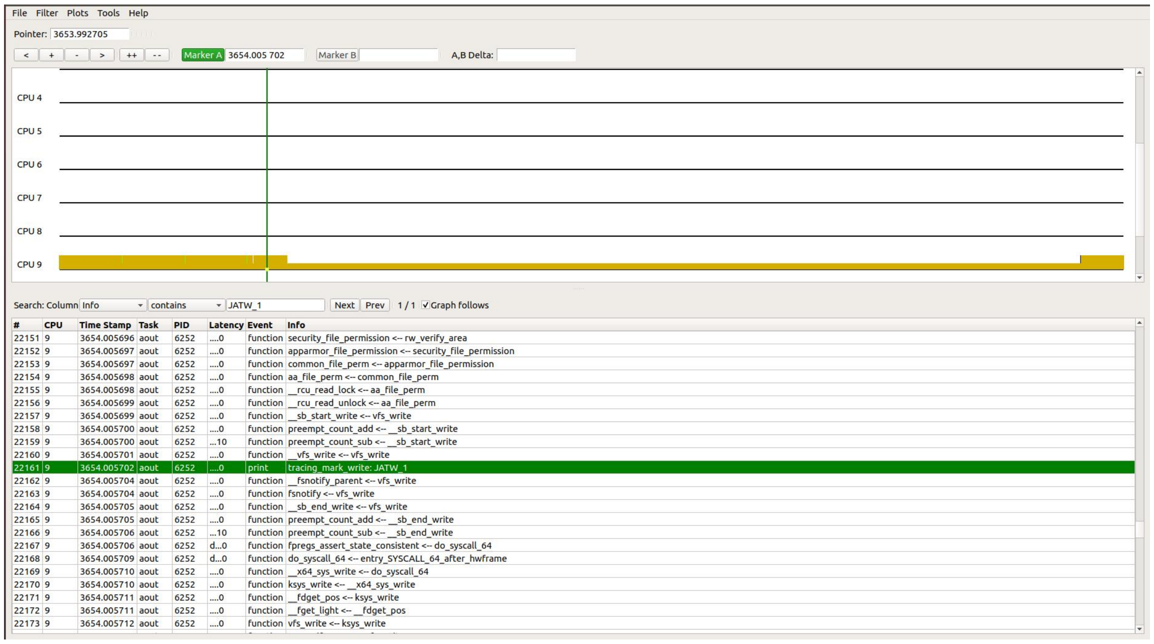
(30 points) The following document explains three ways of building time-triggered tasks (see slide 5 till slide 7 of the `http://retis.sssup.it/~lipari/courses/str09/10.rtprogramming-handout.pdf`). Assume that our goal is to build periodic tasks. In that case, compare these three approaches by building a task set (and scheduling it with rate-monotonic priorities using *SCHED_FIFO*). On that task set, choose a task and then measure the difference between the expected arrival times of the task and the actual release times. Show that unlike the first two methods, the last one results in periodic activations. Justify your answer by adding the traces of KernelShark and visualizing the arrival times of the task

Deadline: The deadline of the assignment is **28 March, 2021 23:59**



```c
int main(void)
{
    pthread_t th1, th2;
    printf("%s\n", "This is a Hellow World program to test trace_marker");
    trace_fd = open("/sys/kernel/debug/tracing/trace_marker", O_WRONLY);
    if(trace_fd < 0) {
        printf("Error Opening trace marker, try running with sudo\n");
    } else{
        printf("Successfully opened trace_marker\n");
    }

    trace_write("JATW_1");

    if(pthread_create(&th1, NULL, Thread1, NULL) != 0)
    {
        printf("Error Spawning Thread1");
    }
    else
    {
        printf("Spawned Thread1");
    }

    if(pthread_create(&th2, NULL, Thread2, NULL) != 0)
    {
        printf("Error Spawning Thread2");
    }
    else
    {
        printf("Spawned Thread2");
    }

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    trace_write("JATW_2");

    if(close(trace_fd) == 0)
    {
        printf("Successfully closed the file\n");
    }

    return 0;
}
```

Figure 3: main of the FtraceExample.c



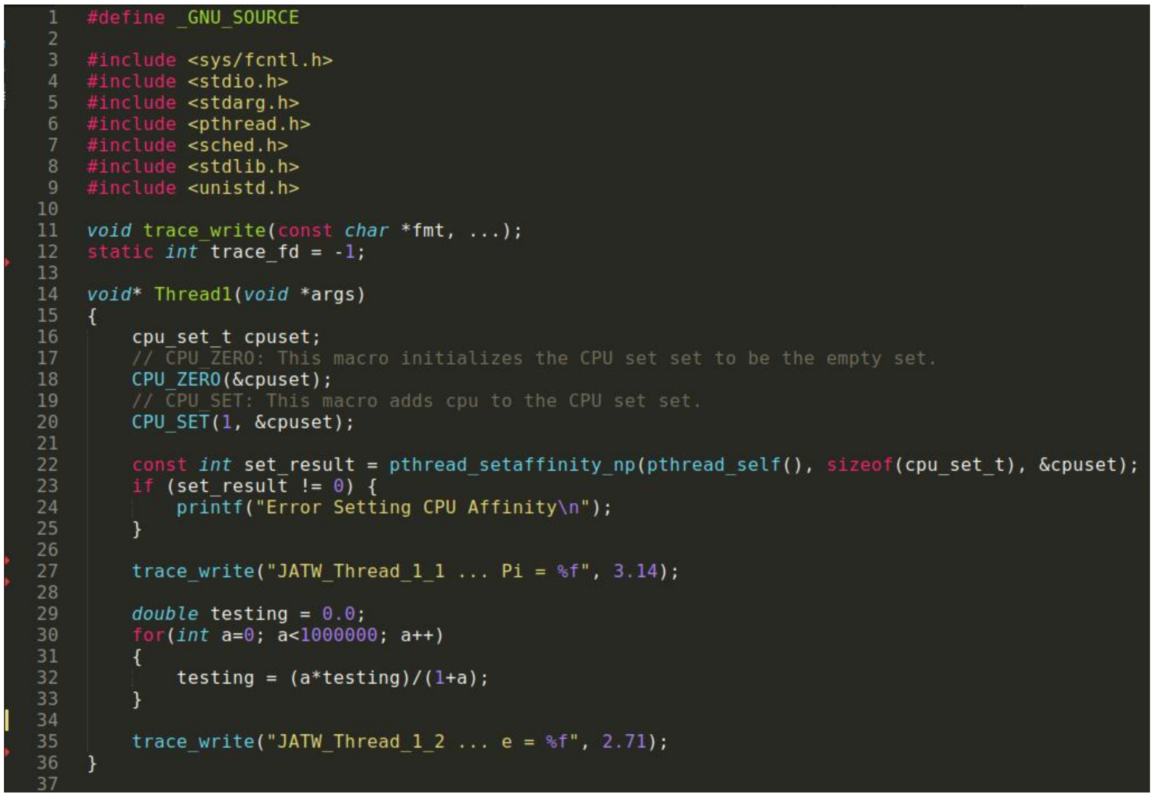
Pointer: 3653.992705

Marker A 3654.005 702 Marker B A,B Delta:

Search: Column Info contains JATW_1 1 / 1 ✓ Graph follows

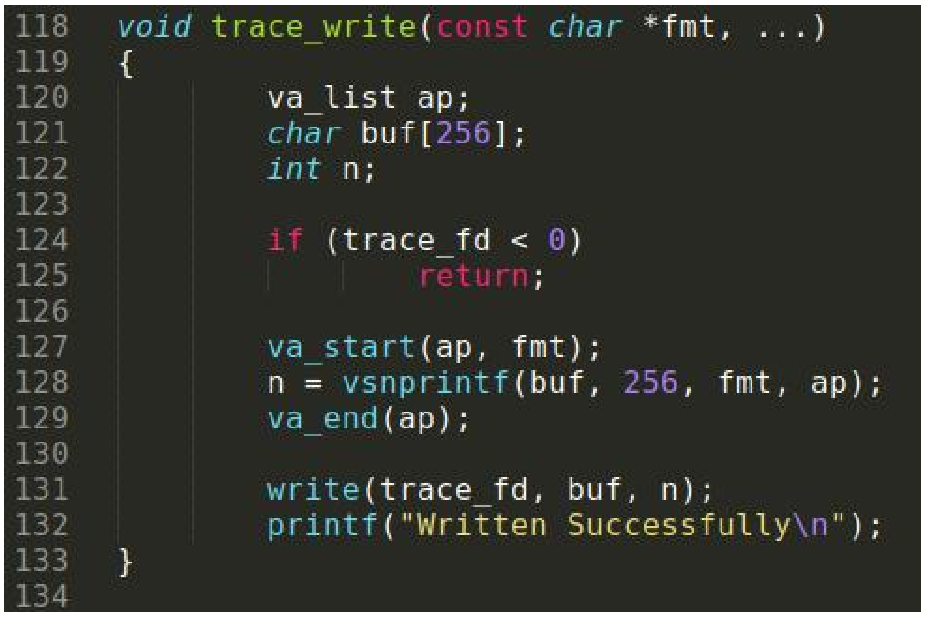
#	CPU	Time Stamp	Task	PID	Latency	Event	Info
22151	9	3654.005696	aout	6252	0	function	security_file_permission <-- rw_verify_area
22152	9	3654.005697	aout	6252	0	function	apparmor_file_permission <-- security_file_permission
22153	9	3654.005697	aout	6252	0	function	common_file_perm <-- apparmor_file_permission
22154	9	3654.005698	aout	6252	0	function	aa_file_perm <-- common_file_perm
22155	9	3654.005698	aout	6252	0	function	__rcu_read_lock <-- aa_file_perm
22156	9	3654.005699	aout	6252	0	function	__rcu_read_unlock <-- aa_file_perm
22157	9	3654.005699	aout	6252	0	function	__sb_start_write <-- vfs_write
22158	9	3654.005700	aout	6252	0	function	preempt_count_add <-- __sb_start_write
22159	9	3654.005700	aout	6252	...10	function	preempt_count_sub <-- __sb_start_write
22160	9	3654.005701	aout	6252	0	function	__vfs_write <-- vfs_write
22161	9	3654.005702	aout	6252	0	print	tracing_mark_write: JATW_1
22162	9	3654.005704	aout	6252	0	function	__fsnotify_parent <-- vfs_write
22163	9	3654.005704	aout	6252	0	function	fsnotify <-- vfs_write
22164	9	3654.005705	aout	6252	0	function	__sb_end_write <-- vfs_write
22165	9	3654.005705	aout	6252	0	function	preempt_count_add <-- __sb_end_write
22166	9	3654.005706	aout	6252	...10	function	preempt_count_sub <-- __sb_end_write
22167	9	3654.005706	aout	6252	d...0	function	fpregs_assert_state_consistent <-- do_syscall_64
22168	9	3654.005709	aout	6252	d...0	function	do_syscall_64 <-- entry_SYSCALL_64_after_hwframe
22169	9	3654.005710	aout	6252	0	function	__x64_sys_write <-- do_syscall_64
22170	9	3654.005710	aout	6252	0	function	ksys_write <-- __x64_sys_write
22171	9	3654.005711	aout	6252	0	function	__fdget_pos <-- ksys_write
22172	9	3654.005711	aout	6252	0	function	__fget_light <-- __fdget_pos
22173	9	3654.005712	aout	6252	0	function	vfs_write <-- ksys_write

Figure 4: Trace marker JATW_1



```c
#define _GNU_SOURCE

#include <sys/fcntl.h>
#include <stdio.h>
#include <stdarg.h>
#include <pthread.h>
#include <sched.h>
#include <stdlib.h>
#include <unistd.h>

void trace_write(const char *fmt, ...);
static int trace_fd = -1;

void* Thread1(void *args)
{
    cpu_set_t cpuset;
    // CPU_ZERO: This macro initializes the CPU set set to be the empty set.
    CPU_ZERO(&cpuset);
    // CPU_SET: This macro adds cpu to the CPU set set.
    CPU_SET(1, &cpuset);

    const int set_result = pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset);
    if (set_result != 0) {
        printf("Error Setting CPU Affinity\n");
    }

    trace_write("JATW_Thread_1_1 ... Pi = %f", 3.14);

    double testing = 0.0;
    for(int a=0; a<1000000; a++)
    {
        testing = (a*testing)/(1+a);
    }

    trace_write("JATW_Thread_1_2 ... e = %f", 2.71);
}
```

Figure 5: Sample function Thread_1



```
void trace_write(const char *fmt, ...)
{
        va_list ap;
        char buf[256];
        int n;

        if (trace_fd < 0)
                return;

        va_start(ap, fmt);
        n = vsnprintf(buf, 256, fmt, ap);
        va_end(ap);

        write(trace_fd, buf, n);
        printf("Written Successfully\n");
}
```

Figure 6: Sample function Trace_Write

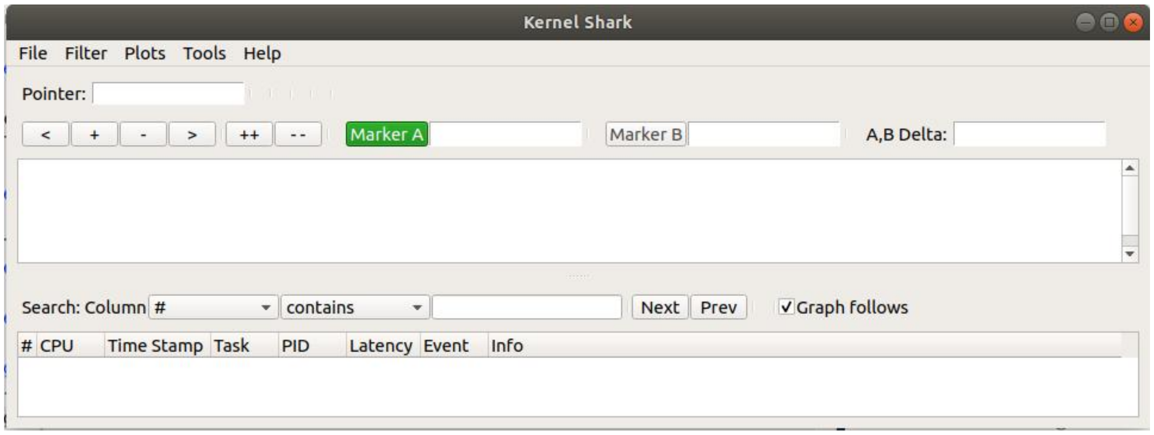
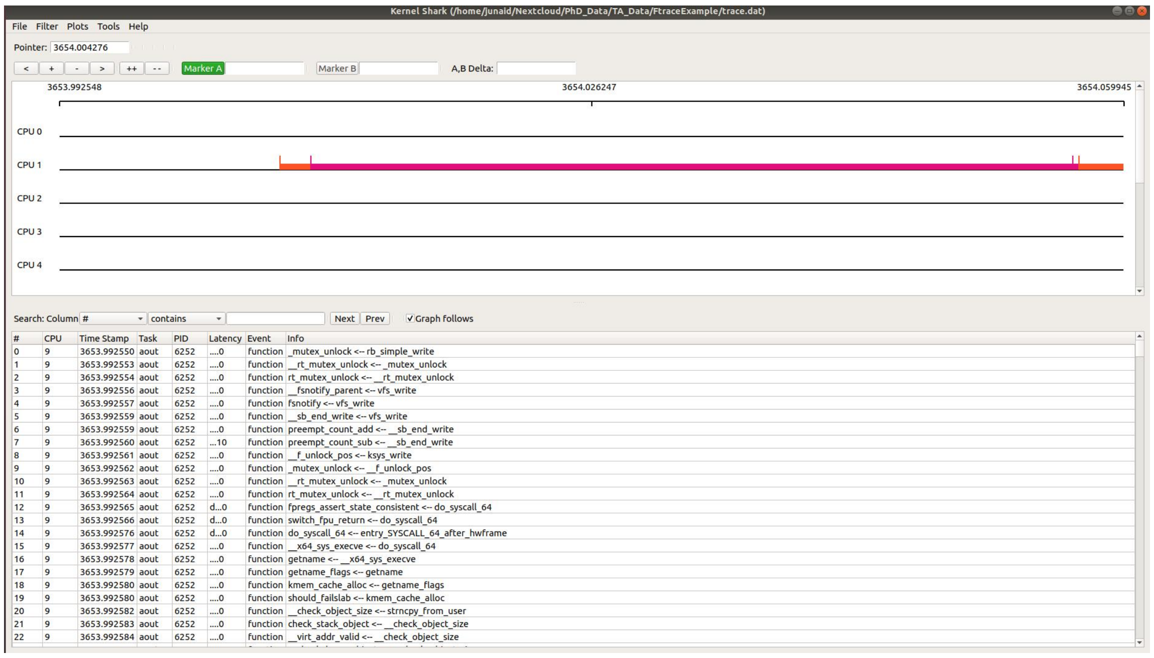


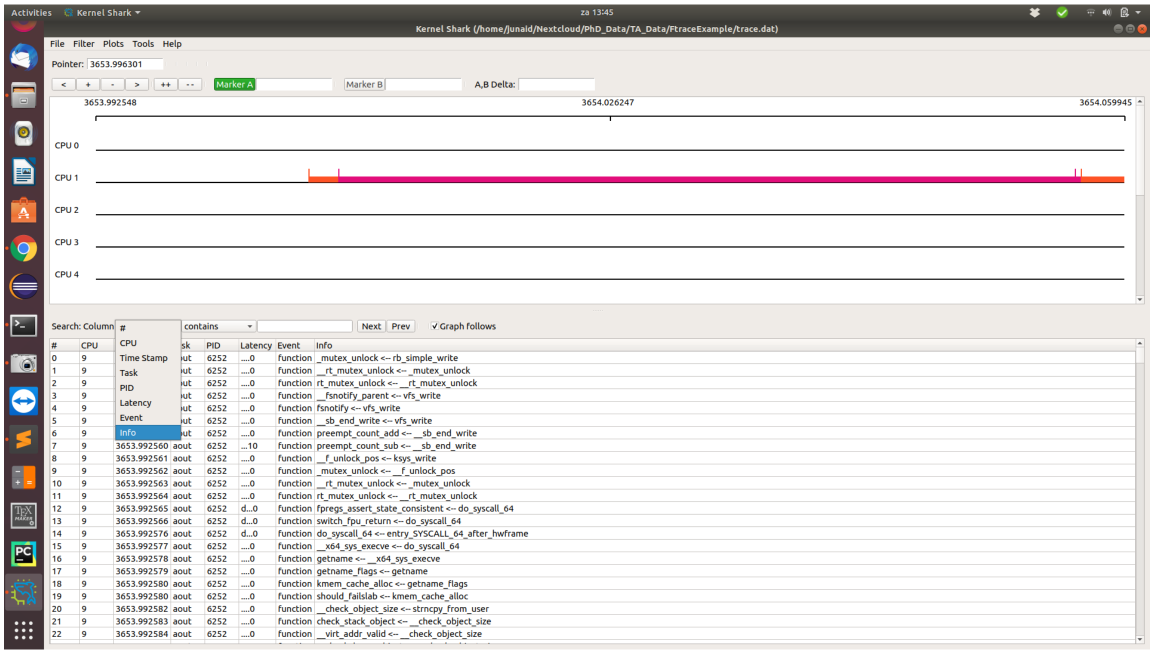
Figure 7: The empty view of KernelShark without any traces


Kernel Shark (/home/junaid/Nextcloud/PhD_Data/TA_Data/FtraceExample/trace.dat)
File Filter Plots Tools Help
< + - > ++ --
Marker A
Marker B
A,B Delta:
3653.992548
3654.026247
3654.059945
CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
Search: Column # contains
Next Prev
Graph follows

Pointer: 3654.004276

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	9	3653.992550	aout	6252	0	function	_mutex_unlock <-- rb_simple_write
1	9	3653.992553	aout	6252	0	function	__rt_mutex_unlock <-- _mutex_unlock
2	9	3653.992554	aout	6252	0	function	rt_mutex_unlock <-- __rt_mutex_unlock
3	9	3653.992556	aout	6252	0	function	__fsnotify_parent <-- vfs_write
4	9	3653.992557	aout	6252	0	function	fsnotify <-- vfs_write
5	9	3653.992559	aout	6252	0	function	__sb_end_write <-- vfs_write
6	9	3653.992559	aout	6252	0	function	preempt_count_add <-- __sb_end_write
7	9	3653.992560	aout	6252	...10	function	preempt_count_sub <-- __sb_end_write
8	9	3653.992561	aout	6252	0	function	__f_unlock_pos <-- ksys_write
9	9	3653.992562	aout	6252	0	function	_mutex_unlock <-- __f_unlock_pos
10	9	3653.992563	aout	6252	0	function	__rt_mutex_unlock <-- _mutex_unlock
11	9	3653.992564	aout	6252	0	function	rt_mutex_unlock <-- __rt_mutex_unlock
12	9	3653.992565	aout	6252	d...0	function	fpregs_assert_state_consistent <-- do_syscall_64
13	9	3653.992566	aout	6252	d...0	function	switch_fpu_return <-- do_syscall_64
14	9	3653.992576	aout	6252	d...0	function	do_syscall_64 <-- entry_SYSCALL_64_after_hwframe
15	9	3653.992577	aout	6252	0	function	__x64_sys_execve <-- do_syscall_64
16	9	3653.992578	aout	6252	0	function	getname <-- __x64_sys_execve
17	9	3653.992579	aout	6252	0	function	getname_flags <-- getname
18	9	3653.992580	aout	6252	0	function	kmem_cache_alloc <-- getname_flags
19	9	3653.992580	aout	6252	0	function	should_failslab <-- kmem_cache_alloc
20	9	3653.992582	aout	6252	0	function	__check_object_size <-- strncpy_from_user
21	9	3653.992583	aout	6252	0	function	check_stack_object <-- __check_object_size
22	9	3653.992584	aout	6252	0	function	__virt_addr_valid <-- __check_object_size

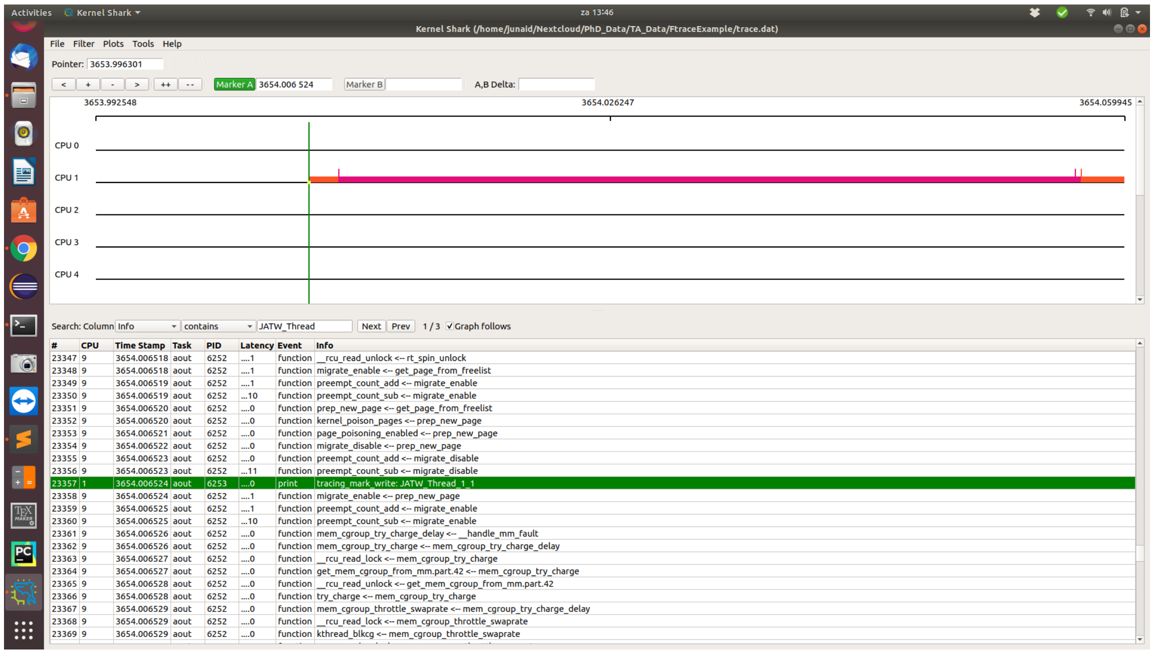
Figure 8: The view of KernelShark with traces



Pointer: 3653.996301

#	CPU		sk	PID	Latency	Event	Info
0	9		ut	6252	0	function	_mutex_unlock <-- rb_simple_write
1	9		ut	6252	0	function	__rt_mutex_unlock <-- _mutex_unlock
2	9		ut	6252	0	function	rt_mutex_unlock <-- __rt_mutex_unlock
3	9		ut	6252	0	function	__fsnotify_parent <-- vfs_write
4	9		ut	6252	0	function	fsnotify <-- vfs_write
5	9		ut	6252	0	function	__sb_end_write <-- vfs_write
6	9		ut	6252	0	function	preempt_count_add <-- __sb_end_write
7	9	3653.992560	aout	6252	...10	function	preempt_count_sub <-- __sb_end_write
8	9	3653.992561	aout	6252	0	function	__f_unlock_pos <-- ksys_write
9	9	3653.992562	aout	6252	0	function	_mutex_unlock <-- __f_unlock_pos
10	9	3653.992563	aout	6252	0	function	__rt_mutex_unlock <-- _mutex_unlock
11	9	3653.992564	aout	6252	0	function	rt_mutex_unlock <-- __rt_mutex_unlock
12	9	3653.992565	aout	6252	d...0	function	fpregs_assert_state_consistent <-- do_syscall_64
13	9	3653.992566	aout	6252	d...0	function	switch_fpu_return <-- do_syscall_64
14	9	3653.992576	aout	6252	d...0	function	do_syscall_64 <-- entry_SYSCALL_64_after_hwframe
15	9	3653.992577	aout	6252	0	function	__x64_sys_execve <-- do_syscall_64
16	9	3653.992578	aout	6252	0	function	getname <-- __x64_sys_execve
17	9	3653.992579	aout	6252	0	function	getname_flags <-- getname
18	9	3653.992580	aout	6252	0	function	kmem_cache_alloc <-- getname_flags
19	9	3653.992580	aout	6252	0	function	should_failslab <-- kmem_cache_alloc
20	9	3653.992582	aout	6252	0	function	__check_object_size <-- strncpy_from_user
21	9	3653.992583	aout	6252	0	function	check_stack_object <-- __check_object_size
22	9	3653.992584	aout	6252	0	function	__virt_addr_valid <-- __check_object_size

Figure 9: Select "Info" from the drop down list of column



Kernel Shark (/home/junaid/Nextcloud/PhD_Data/TA_Data/FtraceExample/trace.dat)

Pointer: 3653.996301

Marker A 3654.006 524

A,B Delta:

Search: Column Info contains JATW_Thread

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
23347	9	3654.006518	aout	6252	1	function	__rcu_read_unlock <-- rt_spin_unlock
23348	9	3654.006518	aout	6252	1	function	migrate_enable <-- get_page_from_freelist
23349	9	3654.006519	aout	6252	1	function	preempt_count_add <-- migrate_enable
23350	9	3654.006519	aout	6252	...10	function	preempt_count_sub <-- migrate_enable
23351	9	3654.006520	aout	6252	0	function	prep_new_page <-- get_page_from_freelist
23352	9	3654.006520	aout	6252	0	function	kernel_poison_pages <-- prep_new_page
23353	9	3654.006521	aout	6252	0	function	page_poisoning_enabled <-- prep_new_page
23354	9	3654.006522	aout	6252	0	function	migrate_disable <-- prep_new_page
23355	9	3654.006523	aout	6252	0	function	preempt_count_add <-- migrate_disable
23356	9	3654.006523	aout	6252	...11	function	preempt_count_sub <-- migrate_disable
23357	1	3654.006524	aout	6253	0	print	tracing_mark_write: JATW_Thread_1_1
23358	9	3654.006524	aout	6252	1	function	migrate_enable <-- prep_new_page
23359	9	3654.006525	aout	6252	1	function	preempt_count_add <-- migrate_enable
23360	9	3654.006525	aout	6252	...10	function	preempt_count_sub <-- migrate_enable
23361	9	3654.006526	aout	6252	0	function	mem_cgroup_try_charge_delay <-- __handle_mm_fault
23362	9	3654.006526	aout	6252	0	function	mem_cgroup_try_charge <-- mem_cgroup_try_charge_delay
23363	9	3654.006527	aout	6252	0	function	__rcu_read_lock <-- mem_cgroup_try_charge
23364	9	3654.006527	aout	6252	0	function	get_mem_cgroup_from_mm.part.42 <-- mem_cgroup_try_charge
23365	9	3654.006528	aout	6252	0	function	__rcu_read_unlock <-- get_mem_cgroup_from_mm.part.42
23366	9	3654.006528	aout	6252	0	function	try_charge <-- mem_cgroup_try_charge
23367	9	3654.006529	aout	6252	0	function	mem_cgroup_throttle_swaprate <-- mem_cgroup_try_charge_delay
23368	9	3654.006529	aout	6252	0	function	__rcu_read_lock <-- mem_cgroup_throttle_swaprate
23369	9	3654.006529	aout	6252	0	function	kthread_blkcg <-- mem_cgroup_throttle_swaprate

Figure 10: Trace text search box