

**CSE438: EMBEDDED SYSTEMS PROGRAMMING
EVENT HANDLING & SIGNALING**

REPORT : ASSIGNMENT 4

AUTHORS:

**NITHEESH MUTHURAJ &
ASU ID:1213383988
nitheesh@asu.edu**

**REVANTH RAJSHEKAR
ASU ID:1213380816
rmrajash@asu.edu**

Table of Contents

ASSIGNMENT STATEMENT	3
PART 1: THE THREAD IS RUNNABLE	4
PART 2: THE THREAD IS BLOCKED BY A SEMAPHORE	6
PART3: THE THREAD IS DELAYED	8

Assignment Statement

In vxWorks' Kernel API Reference Manual, it is stated that "If a task is pended (for instance, by waiting for a semaphore to become available) and a signal is sent to the task for which the task has a handler installed, then the handler will run before the semaphore is taken. When the handler returns, the task will go back to being pended (waiting for the semaphore). If there was a timeout used for the pending task, then the original value will be used again when the task returns from the signal handler and goes back to being pended. If the handler alters the execution path, via a call to `longjmp()` for example, and does not return then the task does not go back to being pended."

This description of the signal delivery mechanism is certainly OS dependent. In task 2 of the assignment, you are requested to develop a program or several pieces of program to test what the Linux's signal facility does precisely and to show the time that a signal handler associated to a thread gets executed in the following conditions:

- a. The thread is runnable (but not running, i.e. the running thread has a higher priority).
- b. The thread is blocked by a semaphore (i.e. `sema_wait()` is called).
- c. The thread is delayed (i.e., `nanosleep()` is called).

PART1: The thread is runnable

IMAGE

Command: `sudo trace-cmd record -e sched_switch -e signal taskset 0x01 ./task2a`

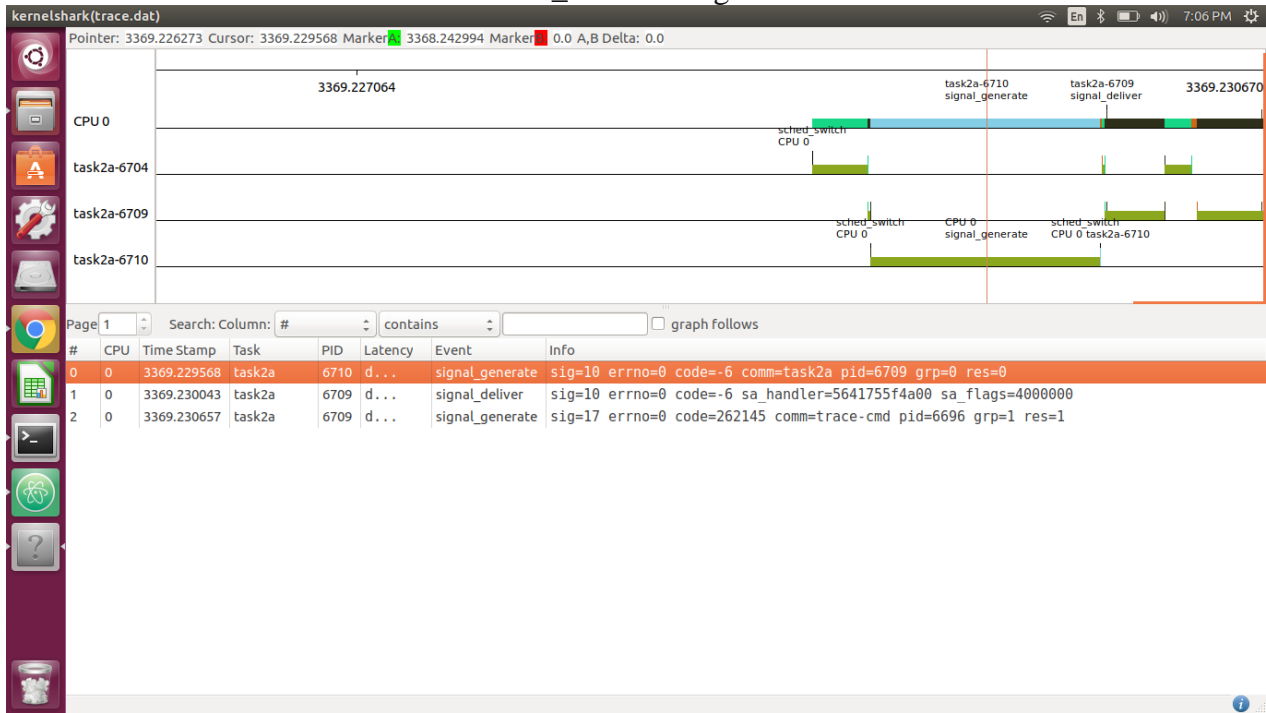


Image 1.1

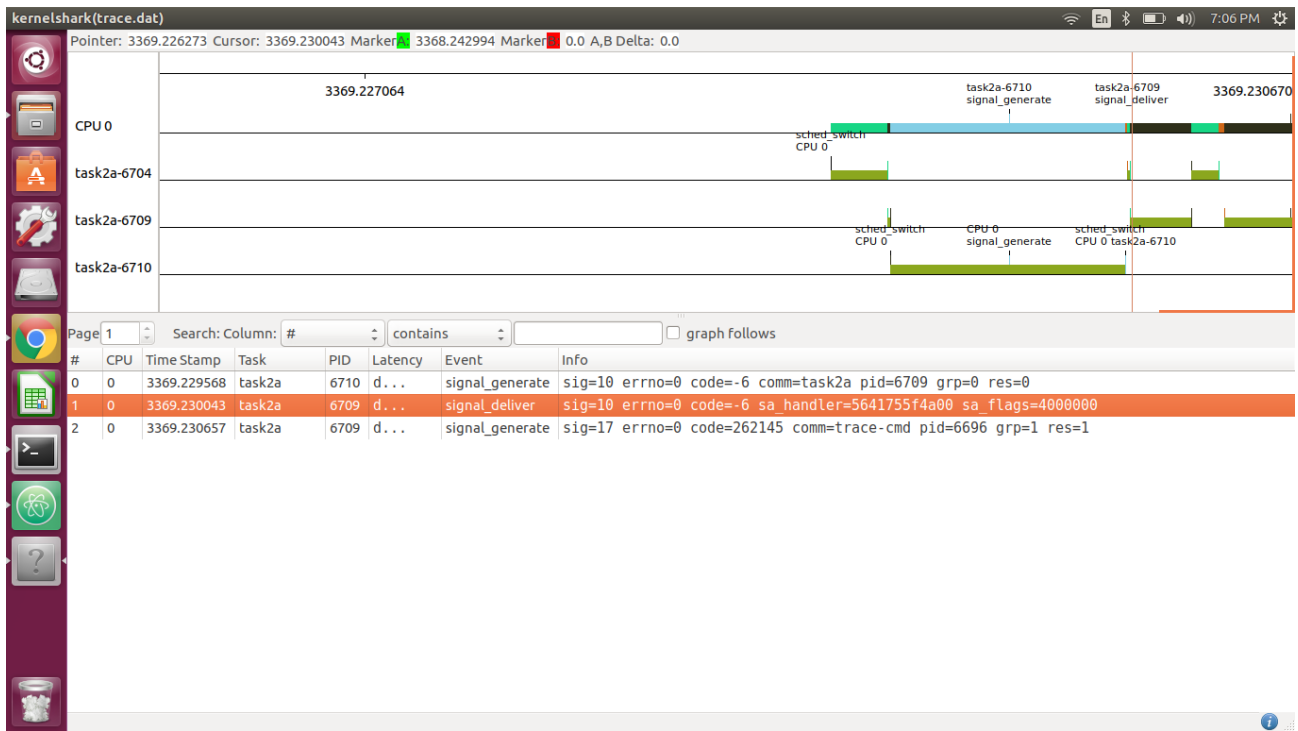


Image 1.2

ANALYSIS

This part of assignment uses 2 threads. A high priority thread & low priority thread. The high priority will be scheduled & is running. The low priority thread is waiting to be scheduled & is runnable. When high priority thread invokes the low priority thread with a signal the higher priority thread will finish execution & then lower priority thread will be scheduled at which time the control enters handler. Its then the lower priority thread will be executed.

As it can be seen from the figure, the signal generate was at time stamp 3369.229568 and the signal was delivered at 3369.230043. Time difference from delivery to generation is 475uS.

PART2:The thread is blocked by a semaphore

IMAGE

Command: `sudo trace-cmd record -e sched_switch -e signal taskset 0x01 ./task2b`

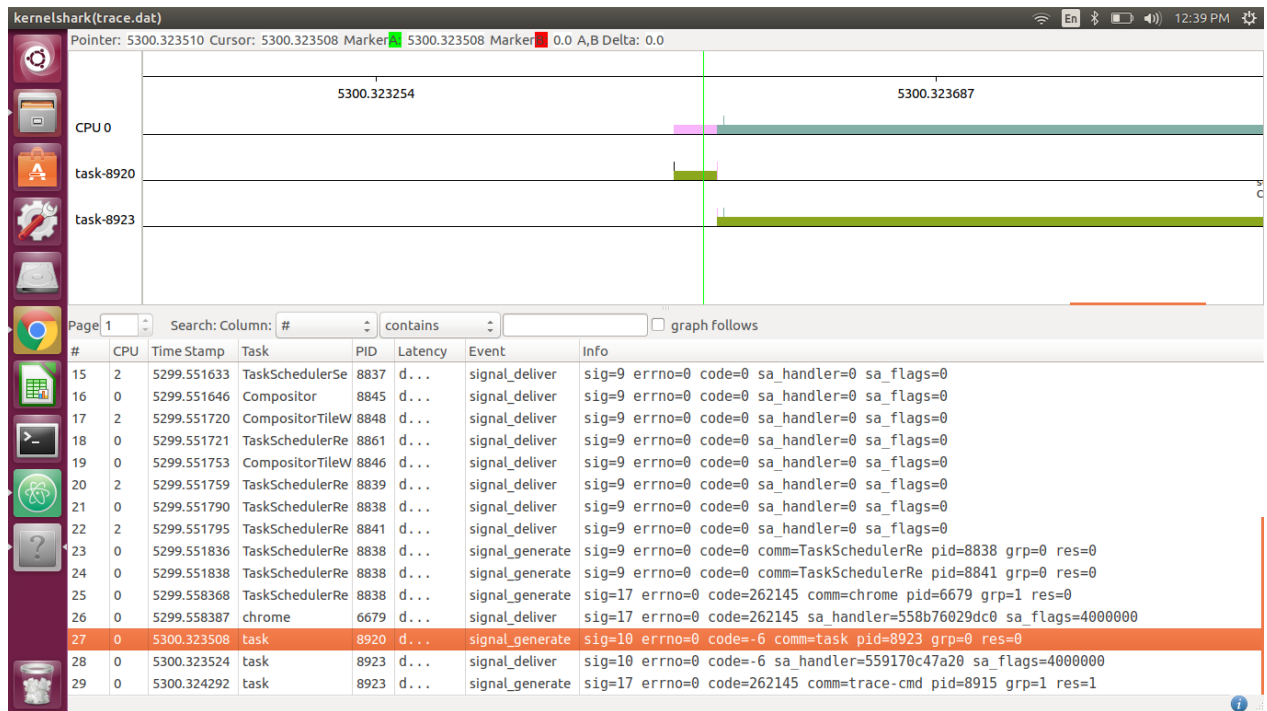


IMAGE 2.1

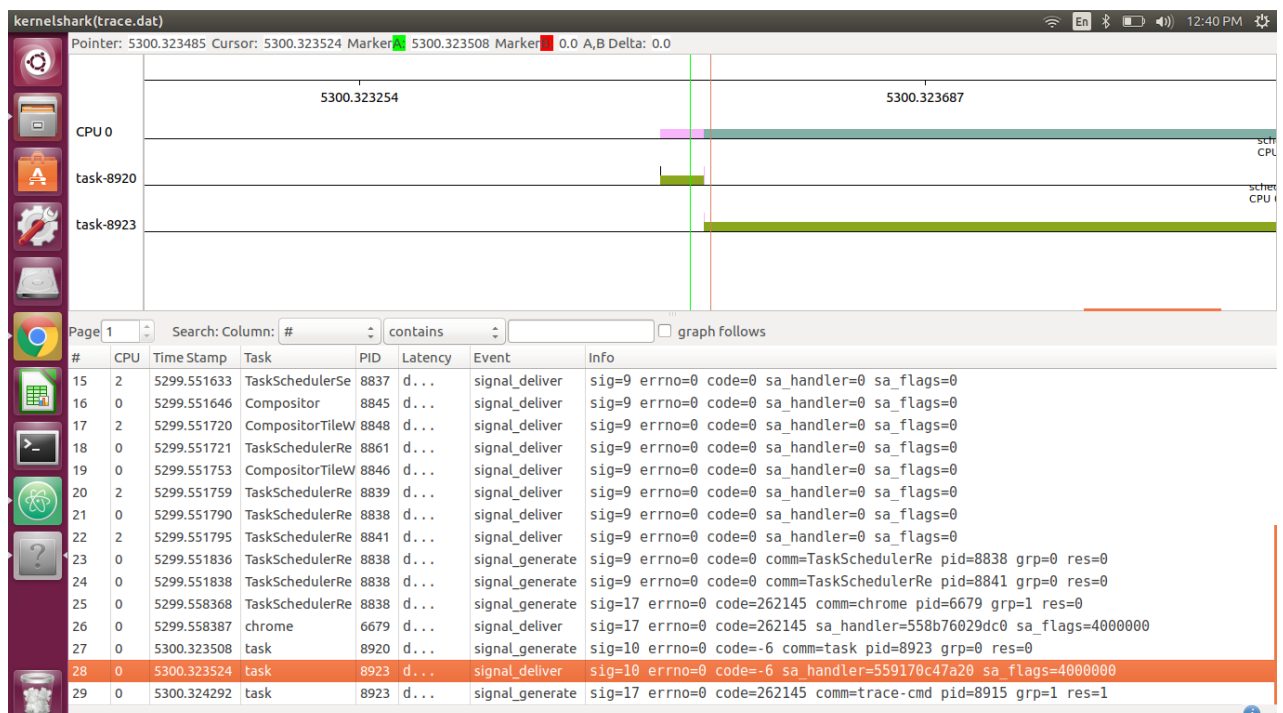


IMAGE 2.2

ANALYSIS:

In this part of the assignment, main thread creates a thread to keep it waiting for semaphore. It attempts to get a semaphore lock, which is not free. Therefore it gets blocked. After 3 seconds of creation the main thread sends a signal to the thread which is waiting to acquire semaphore. After the signal has been delivered to the waiting thread, the thread continues to execute even before acquiring the lock for the same.

Note : the `sem_post` isn't used but the signal generated by main thread releases the waiting semaphore.

As it can be seen from the figure that, The signal was generated at 5300.323508 and was delivered to the waiting thread at 5300.323524. The time between the signal delivered after it was generated is 16uS.

PART 3: The thread is delayed

IMAGE

Command: `sudo trace-cmd record -e sched_switch -e signal taskset 0x01 ./task2c`

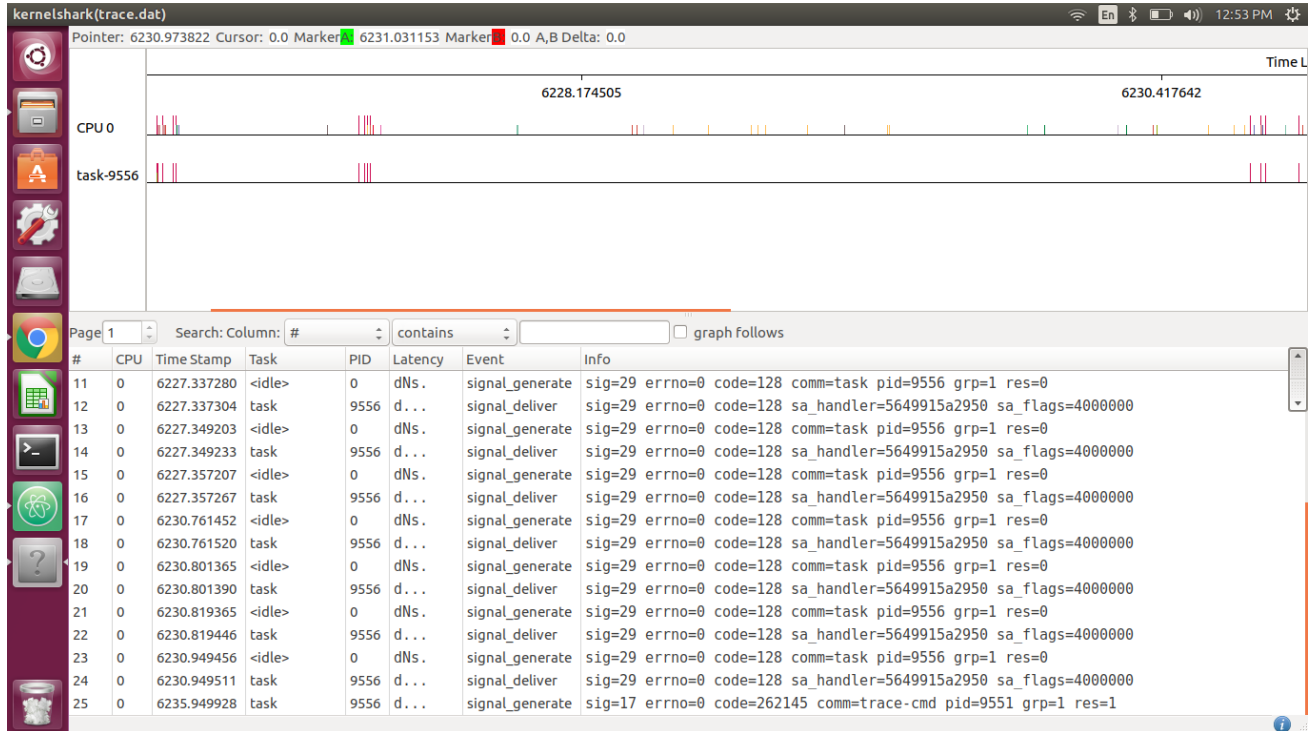


Image 3.1 : 12 times the main function is interrupted by sig=29

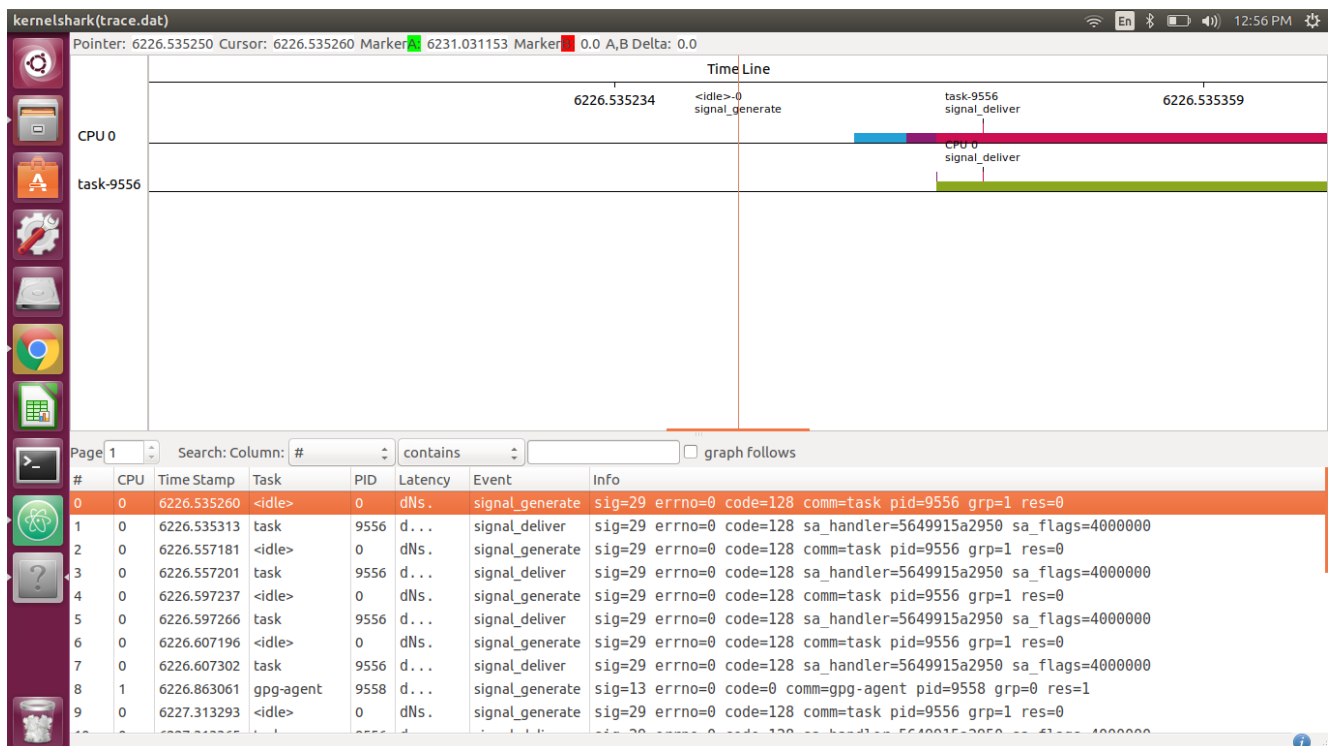


Image 3.2

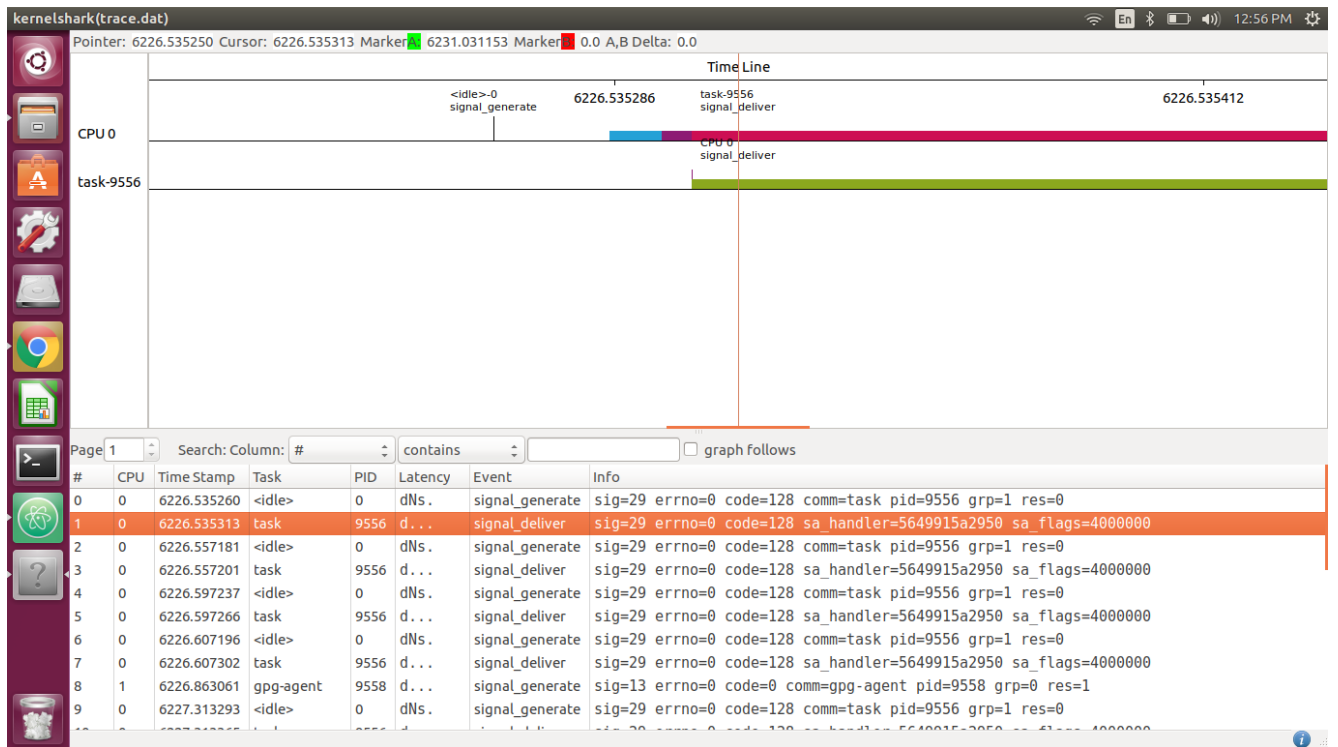


Image 3.3 :

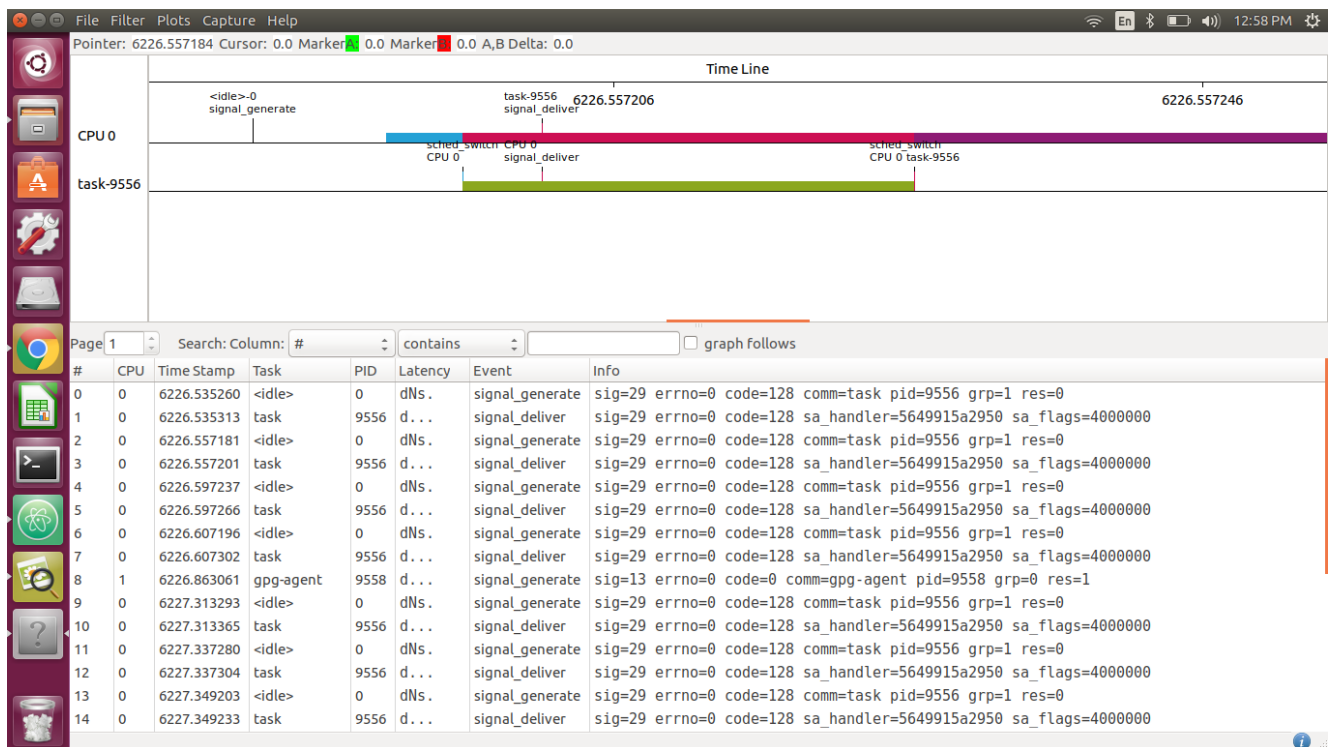
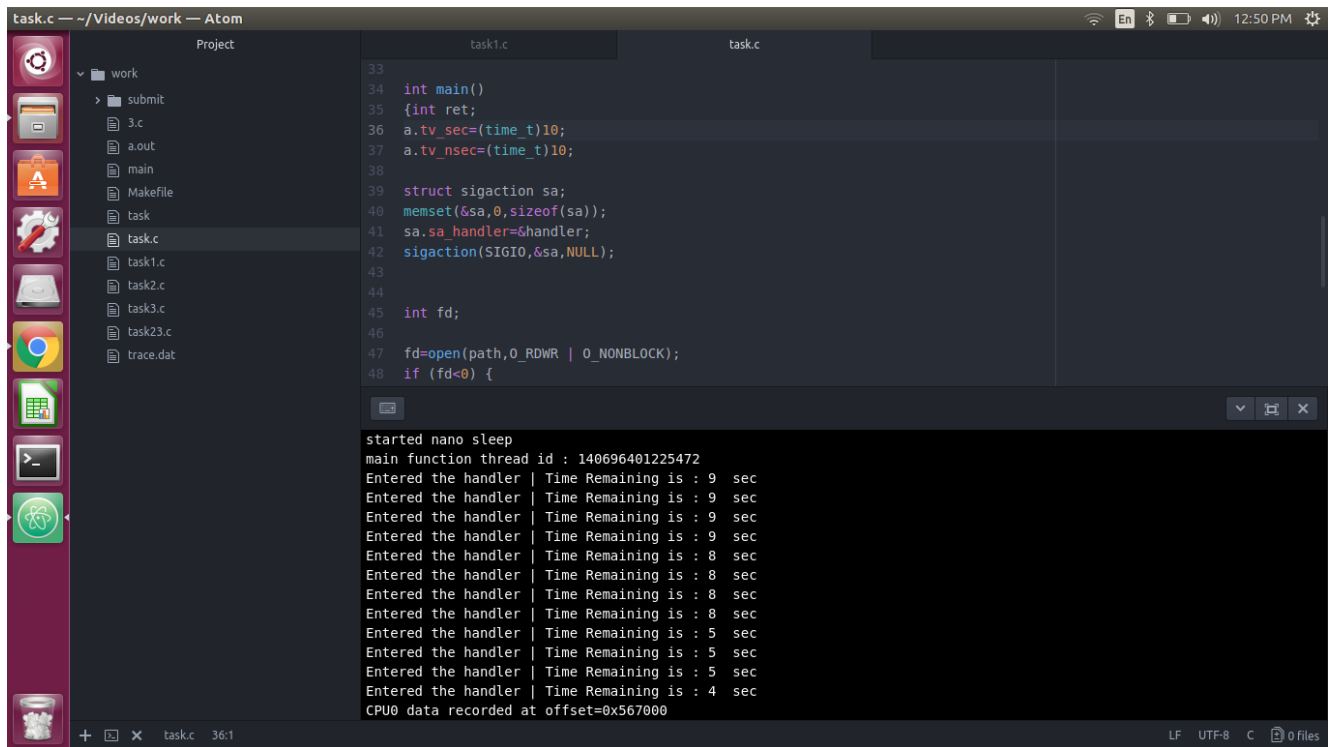


Image 3.4 :Another instance of signal delivered & Generated



The screenshot shows the Atom code editor with a project named 'work'. The file explorer on the left lists files: submit, 3.c, a.out, main, Makefile, task, task.c, task1.c, task2.c, task3.c, task23.c, and trace.dat. The main editor window displays the code for 'task1.c' and 'task.c'. The code in 'task1.c' is as follows:

```
33
34 int main()
35 {int ret;
36 a.tv_sec=(time_t)10;
37 a.tv_nsec=(time_t)10;
38
39 struct sigaction sa;
40 memset(&sa,0,sizeof(sa));
41 sa.sa_handler=&handler;
42 sigaction(SIGIO,&sa,NULL);
43
44
45 int fd;
46
47 fd=open(path,O_RDWR | O_NONBLOCK);
48 if (fd<0) {
```

The output window at the bottom shows the execution of the program. It starts with 'started nano sleep' and 'main function thread id : 140696401225472'. The output then shows a series of 'Entered the handler' messages, each followed by 'Time Remaining is : 9 sec' or '8 sec' or '5 sec' or '4 sec'. The output ends with 'CPU0 data recorded at offset=0x567000'.

Image 3.5 : 12 interrupts are generated by mouse which can be seen in Image 3.1

ANALYSIS:

In this part of the assignment, the signal is generated by mouse actions. The main has delayed itself by going into sleep by calling `nanosleep()`. The main thread wakes up as soon as it receives interrupt signal generated by the mouse action. This signal after being delivered to main thread, leaves the main thread in the awake conditions and executed the handler & the threads sleep for the remaining time and then starts to execute the remaining part of code.

From the figure it can be seen that, the signal was generated at 6226.535260 and it was delivered to the main thread at 6226.535313. The delay in signal being generated to being delivered is 53uS. Signal no : 29.