CSE 522: RTES – Assignment 3 Report

- **Author:** Viraj Savaliya
- **ASU ID:** [redacted]
- **Date:** 03/31/2021

The implementation patch for tracing thread events backend in the assignment 3 is included with this submission. The backend for CONFIG_TRACING_THREAD_EVENTS includes hooks which can be used to trace the thread events throughout it's execution. The different hooks present in trace_events.c are:

1. ***sys_trace_thread_switched_in()*** : This hook is called when the thread switches in to run it's task.
2. ***sys_trace_thread_switched_out()*** : hook is called when the thread switches out after completion of task.
3. ***sys_trace_thread_create()*** : This hook is called whenever a new thread is created.
4. ***sys_trace_thread_ready()*** : This hook is called when a thread is ready to be scheduled.
5. ***sys_trace_thread_pend():*** This hook is called when is thread is put into a pending state or wait queue to become ready after some time to run it's task.
6. ***sys_trace_void():*** This hook is used to indicate start of an API call.
7. ***sys_trace_end_call():*** This hook is used to indicate end of an API call.
8. ***sys_trace_isr_enter(), sys_trace_isr_exit(), sys_trace_idle()*** : These are kept as empty prototype hooks to execute nothing when in the backend as they don't resemble any thread event activity.

In all the hooks a timestamp is recorded at the start and the thread name of the thread which called the hooks is found. This data is then stored in an internal buffer is it has space and only when tracing is ON and the thread is a one of the threads to be traced. The buffer then stores the thread name, timestamp and an integer value using a global incremental buffer index indicating what event the recorded data is for. The integer values with the representation is as follows:

0- Thread switch out, 1- Thread switch in, 3- Thread create, 4- Thread ready, 5- Thread pending, 6- Mutex lock, 7- Mutex unlock

In ***sys_trace_void()*** hook, checking is done to find the API call as per the ID argument passed to it. Similarly, in ***sys_trace_end_call()*** checking is done to find end of an API call as per the ID passed as argument. Different Ids used with this hooks in the implementation are SYS_TRACE_ID_MUTEX_LOCK, SYS_TRACE_ID_MUTEX_UNLOCK, TH_TRACE_START, TH_TRACE_END and TH_TRACE_DUMP.

The Mutex lock and unlock events work similar to other hooks to record timestamp, thread name and event indicator integer in the internal buffer. The trace start and trace end call just update the global flag to indicate tracing ON and OFF int the backend. The Dump ID is used to dump all the values stored in the buffer when the hook is called. The new additional IDs of trace start, trace end and trace dump are defined as an enum in the header file. The buffer size if a defined macro in the header file of backend and can be updated as per need.

The application program used to test the tracing thread events backend is present in samples/trace_app directory. This application initializes all the mutexes required by different threads to be ran, then initializes the semaphore for timeout of the and the timer to check for timeout. Then the timer is started and tracing_start() is called upon to indicate the tracing backend to turn ON the tracing for any desired threads calling the hooks. Then the threads are created with defining its stack area , thread struct, API to execute task and priority. Each thread is also assigned some name to identify it with in the kernel. Once the threads are started a semaphore is taken and locked to ensure it's execution till the end of timeout. The threads then run their tasks with periodic manner and till the timeout is hit. Threads perform 3 computations where the second computation is a critical region and perform its using a mutex lock. All the information for each thread for it's name, period, computation iterations, priority and mutex id are given in form of a struct which is statically defined in the header file for each thread. Also, the number of threads, mutexes and timeout value is defined as macros in the header file. Once the timeout value is hit, the backend is informed to turn OFF the tracing and then the threads are aborted. All the recorded values in the buffer till that point is then dumped on to the console and the timeout semaphore is released.

These dumped values are then recorded into a log file generated form the PUTTY session and then converted into a vcd file using the vcd_generator.py script provided in samples/trace_app directory. The different thread events recorded for each executing task thread can be viewed as a waveform using GTKWave.

All the required changes to install the implemented backend in zephyr for tracing thread events is provided a patch file for zephyr version 1.14.1. The changes done to the CMake, Kconfig and necessary tracing files to add the backend can be viewed in the patch file.

The following are the outputs obtained in GTKWave from the implementation run for the CONFIG_TRACING_THREAD_EVENTS using two different priority ceiling values.

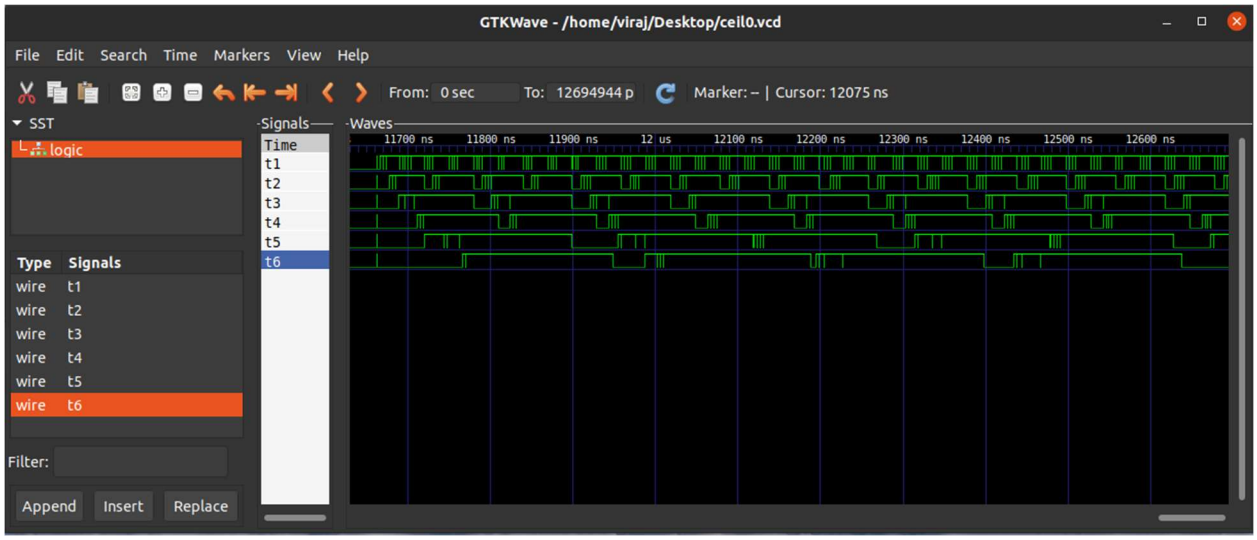

Figure a: Output Traced thread events with CONFIG_PRIORITY_CEILING = 0

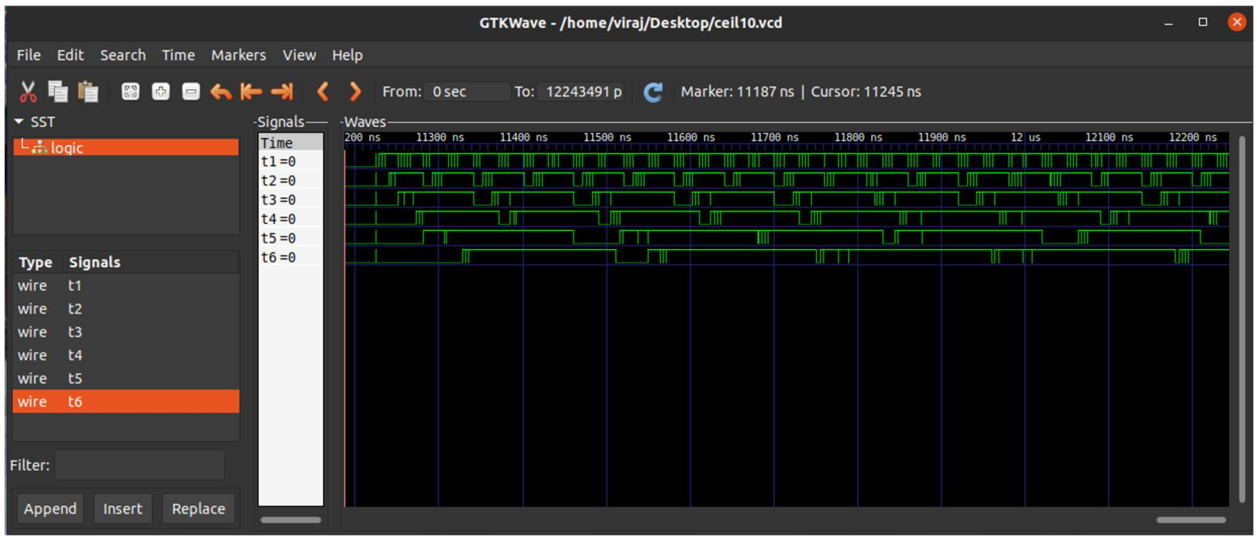

Figure b: Output Traced thread events with CONFIG_PRIORITY_CEILING = 10

The input taskset used for the above two runs is as follows:

```
struct Tasks
{
        char t_name[32];        // task name
        int priority;           // priority of the task
        int period;             // period for periodic task in miliseconds
        int loop_iter[3];       // loop iterations for compute_1, compute_2 and compute_3
        int mutex_m;            // the mutex id to be locked and unlocked by the task
};
```

```
#define THREAD0 {"task 0", 2, 10, {100000, 100000, 100000}, 2}
#define THREAD1 {"task 1", 3, 28, {100000, 100000, 100000}, 0}
#define THREAD2 {"task 2", 4, 64, {100000, 100000, 100000}, 2}
#define THREAD3 {"task 3", 5, 84, {100000, 100000, 100000}, 1}
#define THREAD4 {"task 4", 6, 125, {100000, 100000, 100000}, 0}
#define THREAD5 {"task 5", 7, 165, {100000, 100000, 100000}, 2}
```

The output waveform depict the priority inheritance effect least priority can be seen for task 5 from the waveforms as it takes longer time to execute a task and then switch out. The shortest time to complete a task is for task 0 as it has highest priority. Also the effect of the priority ceiling value can be seen if we observe closely both he zoomed in pictures. The time taken to execute the tasks is more and a little more delayed in CONFIG_PRIORITY_CEILING = 0 for low priority tasks when compared to the same tasks with CONFIG_PRIORITY_CEILING = 10. When the ceiling priority value is higher, the lower priority tasks are not immediately switch to higher ones when a higher priority task is ready but when the priority value is lower it is easy to switch task execution to higher priority ones and the low priority tasks are then delayed more and more. This effect can be seen the output waveforms displayed.