Assignment-3 Report

Raunak-1217240245

1 THREAD EVENT TRACING IMPLEMENTATION

1.1 BACKEND

Created the files thread_events.h and thread_events.c and defined following functions in the thread_events.h,

```
void sys_trace_thread_switched_in(void);
void sys_trace_thread_switched_out(void);
void sys_trace_void(unsigned int id);
void sys_trace_end_call(unsigned int id);

void sys_trace_tracing_dump(void);
void sys_trace_tracing_start(void);
void sys_trace_tracing_end(void);

void sys_trace_thread_create(struct k_thread *thread);
void sys_trace_thread_pend(struct k_thread *thread);
void sys_trace_thread_ready(struct k_thread *thread);

void sys_trace_mutex_lock(void);
void sys_trace_mutex_unlock(void);
```

structure of buffer is defined as,

```
struct trace {
  char threadid[32];
  u32_t count;
  u32_t time;
};
```

thread_switched_in and out logs the thread switching information in the buffer. thread_create pend and ready is used to log the created, pending and ready status of the particular thread in the buffer. mutex_lock and unlock registers a pulse in the buffer whenever a thread takes a mutex lock and unlock. trace_tracing_start and trace_tracing_end uses sys_trace_void and sys_trace_end_call to start and end the tracing[1].

1.2 APPLICATION

In the application part, we create six threads which run in an infinte loop and is killed after a time interval specified for which we want to calculate the tracing data. sys_trace_tracing start is called before the creation of the thread and sys_trace_tracing_end is called after the interval till which we want to do the tracing and in the end sys_trace_tracing_dump is called which dumps the buffer values using printk on the screen.

1.3 CONVERSION TO VCD FILE

Using the logging feature of putty, we save the dumped values in a csv file and we then run our python script on the csv file to convert it into vcd file.

1.4 OUTPUT

To view the results, we use gtkwave software to analyze our thread tracing results using the vcd file created.

2 RESULTS

2.1 WAVEFORMS

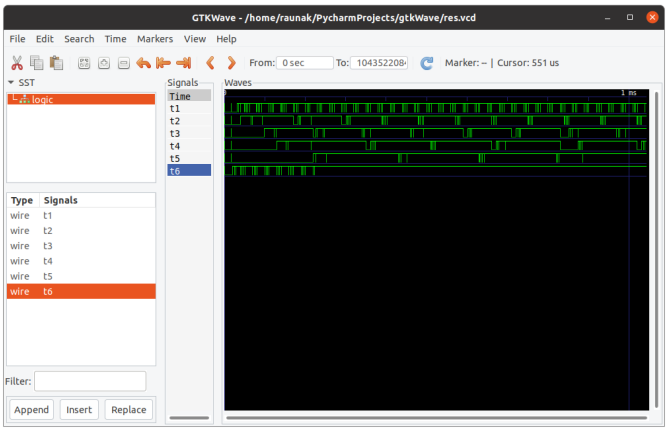

Figure 2.1: CONFIG_PRIORITY_CEILING=0

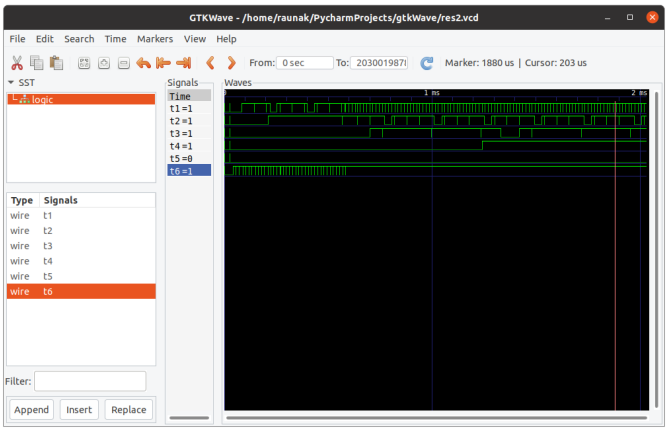

Figure 2.2: CONFIG_PRIORITY_CEILING=10

2.2 Priority Inheritance

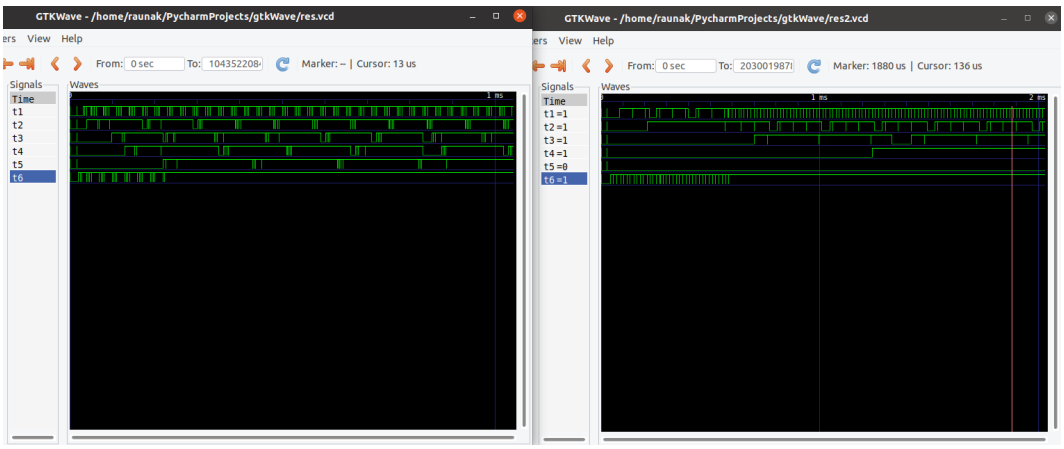

Figure 2.3: Priority inheritance comparison

The right side is run with priority inheritance enabled and we can see the lower priority threads doesn't get switched by the medium priority threads while the higher priority threads are waiting because the lower priority thread inherits the priority of the higher priority thread while the higher prioirty thread is waiting for the critical section to be unlocked by the lower priority threads.

REFERENCES

[1] “device.h - include/device.h - zephyr source code (v1.14.2).” [Online]. Available: https://elixir.bootlin.com/zephyr/v1.14.2/source/include/device.h