CSE 522 Assignment 1 Report

Author: Ashish Kumar Rambhatla

ASU ID: 1215350552

Introduction

This report explains the implementation details for the problem statement mentioned in the CSE 522 Assignment 1. The problem statement states to create a set of periodic tasks with the specified parameters such as priority, periodicity, etc. The total program should be executed in 4000 milliseconds as mentioned in the description. The following sections will do a code through, explaining each step on how it helps in achieving the problem description.

Code Walk-through

1. Initialise the thread stacks required for the threads to be spawned using K_THREAD_STACK_ARRAY_DEFINE macro as shown below.

```
/* Initialising the stacks for the specified number of threads */
#define STACKSIZE 1024
K_THREAD_STACK_ARRAY_DEFINE(threadStackGlobal, NUM_THREADS, STACKSIZE);
```

2. Define custom datastructures which are used as cookies in the k_timer to identify the threads. This feature is useful for identifying the threadID of the thread whose deadline expired.

```
/*
 * Defining the cutom datastructures.
 */
struct threadTimerData {
  int64_t timestamp;
  k_tid_t tid;
  int taskNumber;
  atomic_t taskCompleted;
};

struct globalTimerData {
  k_tid_t spawnedTids[NUM_THREADS];
  atomic_t exitFlag;
  const struct shell *shell;
};
```

3. Implemented an activate function as an entry point for the root command shell.

```
int activate(const struct shell *shell, size_t argc, char **argv) {
    ARG_UNUSED(argc);
  ARG_UNUSED(argv);
  gThreadData.shell = shell;
    taskDispatcher();
    return 0;
}
SHELL_CMD_REGISTER(activate, NULL, "Activating all the threads", activate);
```

4. Implemented the deadline handlers which are passed as timer expiry functions to handle the dedline checking and exiting the program.

   i) In the thread deadline expiry handler, I check for the individual tasks completion using a flag. Based on the flag value, I either print the deadline missed message or clear the flag to restart the task execution.

   ii) In the exitExpiryHandler, I set the exitFlag and wakeup all the spawned threads to indicate the program termination using k_wakeup.

```
/* Handler for thread deadline expiry*/
void threadDeadlineHandler(struct k_timer *timer) {
  struct threadTimerData *threadData = (struct threadTimerData*)k_timer_user_data_get(timer);
  int taskNumber = threadData->taskNumber;
  if(atomic_get(&threadData->taskCompleted)) {
    atomic_dec(&threadData->taskCompleted);
  } else {
    printk("Deadline for the task: %d has missed\n", taskNumber);
  }
  return;
}

/* Handler for program exit expiry */
void threadExitHandler(struct k_timer *timer) {
  struct globalTimerData *gThreadData = (struct globalTimerData*)k_timer_user_data_get(timer);
  atomic_inc(&gThreadData->exitFlag);
  for (int i = 0; i< NUM_THREADS; i++) {
      k_wakeup(gThreadData->spawnedTids[i]);
  }
  return;
}
```

5. In the Thread Function, a timer is started with the thread’s period. A compute sequence is executed and then we set the taskCompleted flag. k_timer_status_sync api is used to put the thread to sleep till the timer expires and this loop continues till the exitFlag is set.

```
void threadFunction(struct task_s *taskInfo, int taskNumber, void* dummy) {
  ARG_UNUSED(dummy);
  while(!atomic_get(&gThreadData.exitFlag)) {
    /* Starting the deadline timer */
    k_timer_start(&threadDeadlineTimers[taskNumber], K_MSEC(taskInfo->period), K_MSEC(taskInfo->period));
    updateTimestampInUserData(taskNumber);
    /* Compute sequence */
    compute(taskInfo->loop_iter[0]);
    k_mutex_lock(&mutex[taskInfo->mutex_m], K_FOREVER);
    compute(taskInfo->loop_iter[1]);
    k_mutex_unlock(&mutex[taskInfo->mutex_m]);
    compute(taskInfo->loop_iter[2]);
    /*
     * Setting the taskCompleted flag of the respective thread to help
     * the @threadDeadlineHandler() call identify if a task is completed
```

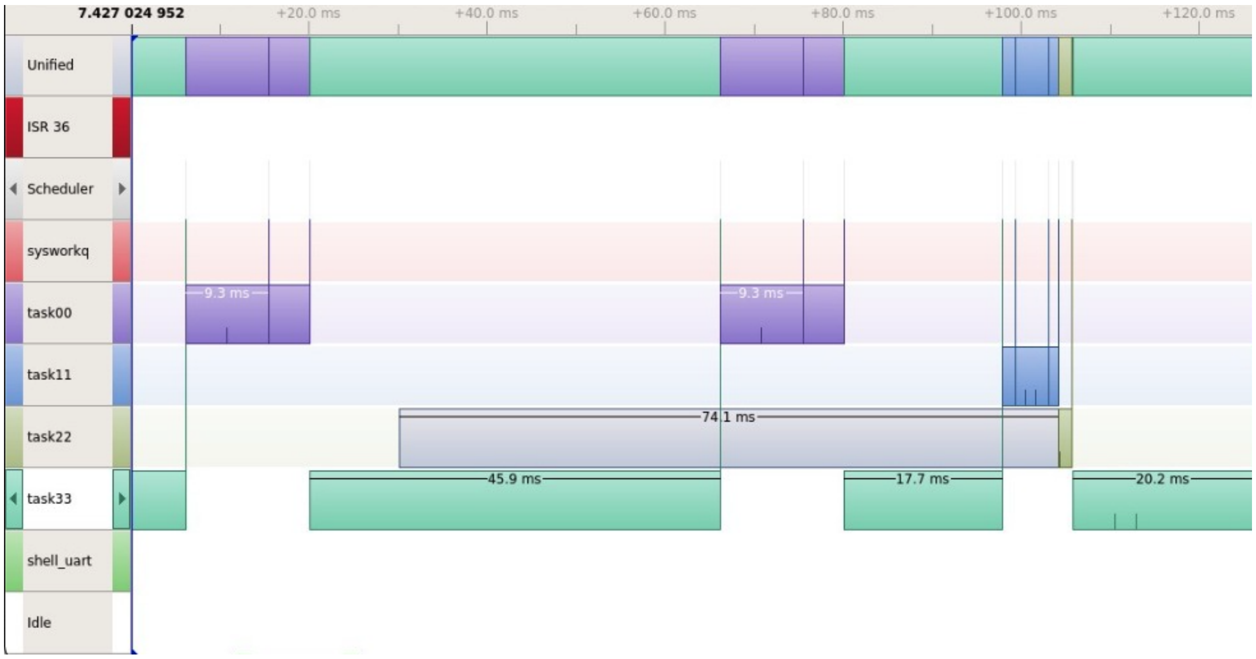
```
       * or not.
       */
      atomic_inc(&threadSpecificData[taskNumber].taskCompleted);
      shell_info(gThreadData.shell, "Completed the compute task and set the flag for task: %d\n", taskNumber);
      /* Putting the thread to sleep till the deadline timer expires */
      k_timer_status_sync(&threadDeadlineTimers[taskNumber]);
    }
  }
```

Systemview analysis

Taskset used for the analysis:

```
#define THREAD0 {"task00", 2, 60, {400000, 400000, 400000}, 0}
#define THREAD1 {"task11", 4, 60, {100000, 120000, 100000}, 1}
#define THREAD2 {"task22", 3, 150, {200000, 12000000, 400000}, 2}
#define THREAD3 {"task33", 5, 50, {10000, 120000, 10000}, 1}
```

a. Priority ceiling disabled.



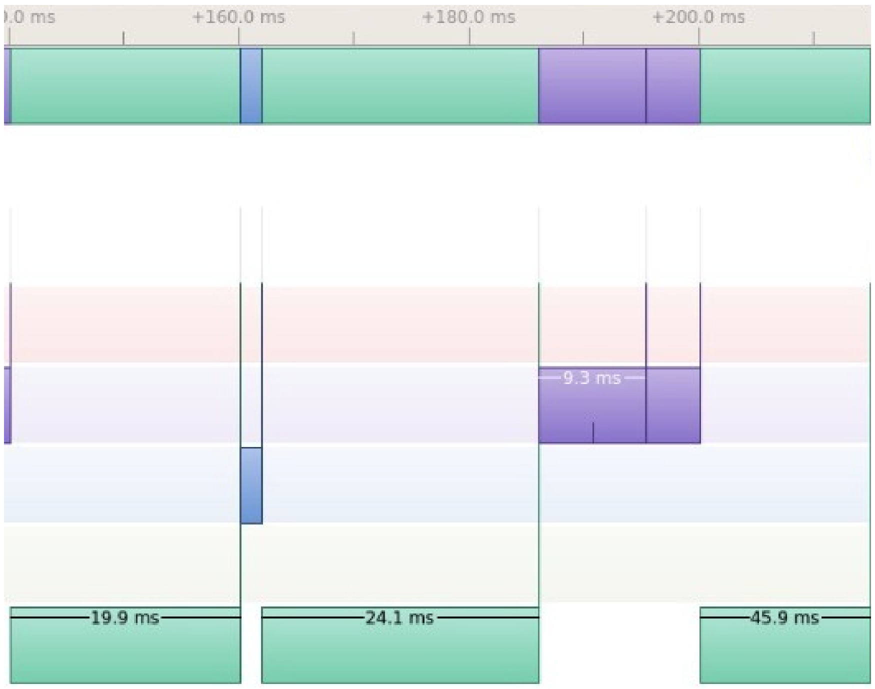
In the above screenshot, we can observe the following as the time taken by the important tasks if not pre-empted.

- task00: 9.3 ms

- task33: 45.9 ms
- task22: 74.1 ms
- task33: 45.9 ms

But at the +80msec, task33 is pre-empted by task11 and then it ran soon after task11 is executed. if we observe the total time (17.7 + 20.2 = 37.9 ms) which is faster than the regular execution time for task33, i.e 45.9ms. Hence we can see the priority ceiling in action.

b. Priority ceiling disabled.



Unlike the above screenshot, here at +200msec the task33 took 45.9 msec which shows the priority ceiling is disabled.