CSE438
Embedded Systems Programming

Project 4 - Real-Time Analysis Report

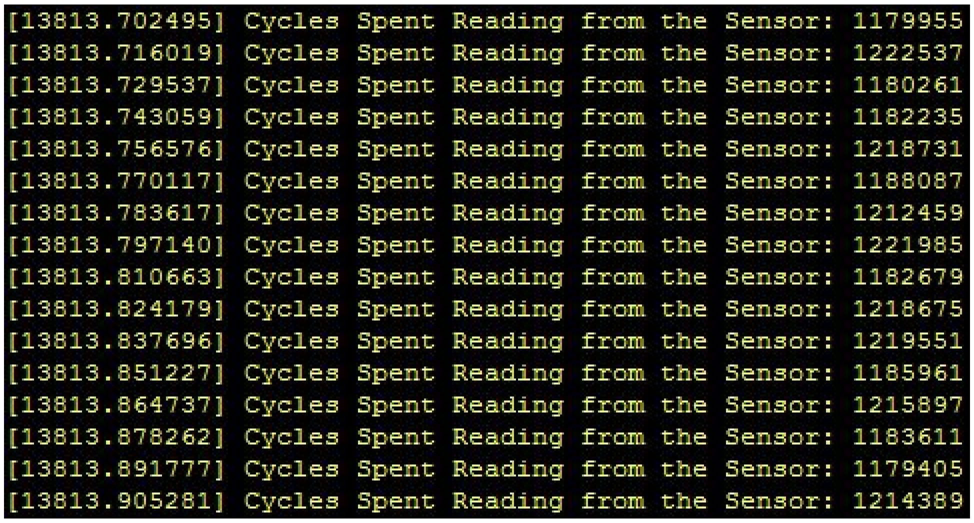
Professor Ryan Meuth

Daniel Pikul
1 December, 2015

Tasks

During program execution, there are 3 primary tasks that run responsible for proper behavior of the game. The first is a kernel level thread named ‘post_data’ responsible for reading data from the MPU6050 sensor module, and posting the gyroscope and accelerometer data as joystick events to the Linux input subsystem. This is called at 200 Hz to ensure new data is always available for the display.
The second is a user-level thread spawned from main which updates the LED display at a refresh rate of 100 Hz.
The third is a user-level thread spawned from main which first reads gyroscope and accelerometer data from the Linux input subsystem, then performs mathematical calculations on the raw data to obtain the roll and pitch angles of the board. These angles are then used to calculate the acceleration of the ball due to gravity, the ball’s velocity, and the ball’s position on the board, from which the display thread will draw the ball on the screen. This is called at 200 Hz, in order to prevent the event file from experiencing an overflow.

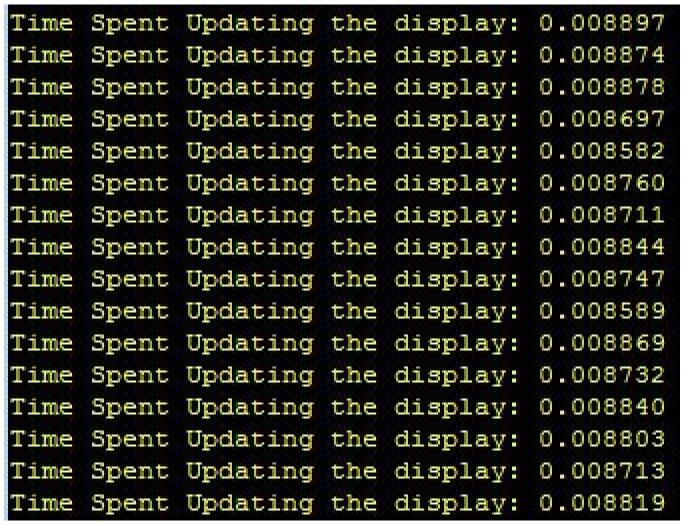
Task 1: Kernel-Level Read (int post_data())



```
[13813.702495] Cycles Spent Reading from the Sensor: 1179955
[13813.716019] Cycles Spent Reading from the Sensor: 1222537
[13813.729537] Cycles Spent Reading from the Sensor: 1180261
[13813.743059] Cycles Spent Reading from the Sensor: 1182235
[13813.756576] Cycles Spent Reading from the Sensor: 1218731
[13813.770117] Cycles Spent Reading from the Sensor: 1188087
[13813.783617] Cycles Spent Reading from the Sensor: 1212459
[13813.797140] Cycles Spent Reading from the Sensor: 1221985
[13813.810663] Cycles Spent Reading from the Sensor: 1182679
[13813.824179] Cycles Spent Reading from the Sensor: 1218675
[13813.837696] Cycles Spent Reading from the Sensor: 1219551
[13813.851227] Cycles Spent Reading from the Sensor: 1185961
[13813.864737] Cycles Spent Reading from the Sensor: 1215897
[13813.878262] Cycles Spent Reading from the Sensor: 1183611
[13813.891777] Cycles Spent Reading from the Sensor: 1179405
[13813.905281] Cycles Spent Reading from the Sensor: 1214389
```

The screenshot above shows a subset of the cycle counts from the run time of the kernel level thread. As can be seen, it takes approximately 1,200,000 clock cycles at 400MHz to execute this function, giving it a total run time of 3 milli-seconds, or 3000 micro-seconds.

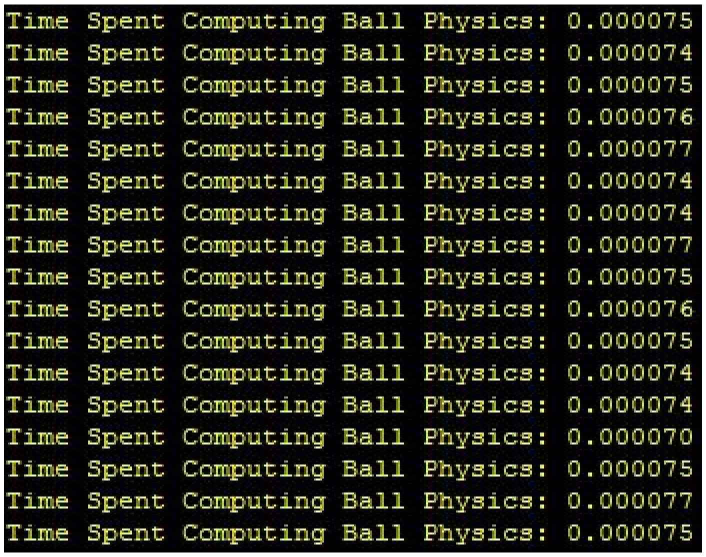
Task 2: Updating the Display (void *max7219_display_thread())



```
Time Spent Updating the display: 0.008897
Time Spent Updating the display: 0.008874
Time Spent Updating the display: 0.008878
Time Spent Updating the display: 0.008697
Time Spent Updating the display: 0.008582
Time Spent Updating the display: 0.008760
Time Spent Updating the display: 0.008711
Time Spent Updating the display: 0.008844
Time Spent Updating the display: 0.008747
Time Spent Updating the display: 0.008589
Time Spent Updating the display: 0.008869
Time Spent Updating the display: 0.008732
Time Spent Updating the display: 0.008840
Time Spent Updating the display: 0.008803
Time Spent Updating the display: 0.008713
Time Spent Updating the display: 0.008819
```

The screenshot above shows a subset of the run time in seconds of the display thread. It takes approximately 8.5 milli-seconds to execute this function, or 8500 micro-seconds.

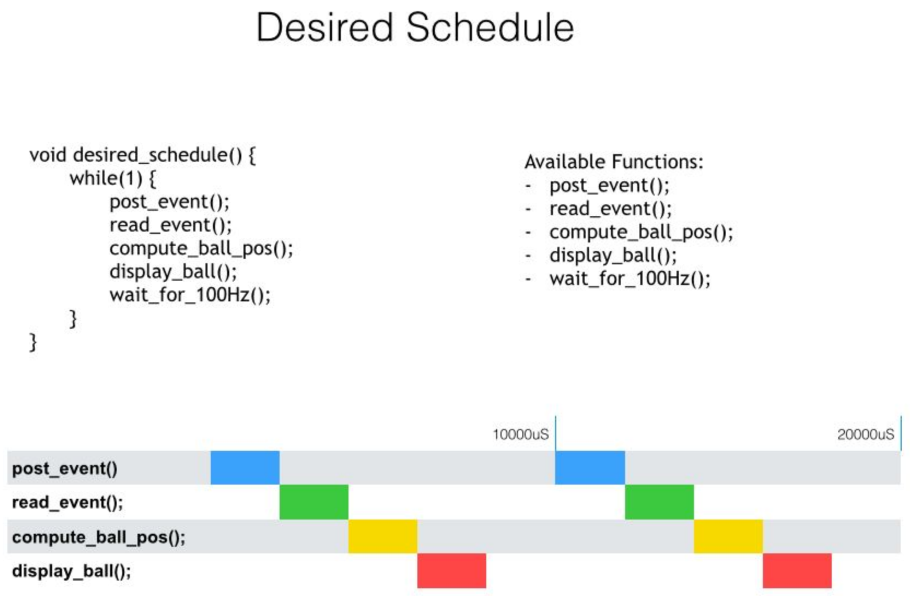
Task 3: Ball Computations (void *compute_thread())



```
Time Spent Computing Ball Physics: 0.000075
Time Spent Computing Ball Physics: 0.000074
Time Spent Computing Ball Physics: 0.000075
Time Spent Computing Ball Physics: 0.000076
Time Spent Computing Ball Physics: 0.000077
Time Spent Computing Ball Physics: 0.000074
Time Spent Computing Ball Physics: 0.000074
Time Spent Computing Ball Physics: 0.000077
Time Spent Computing Ball Physics: 0.000075
Time Spent Computing Ball Physics: 0.000076
Time Spent Computing Ball Physics: 0.000075
Time Spent Computing Ball Physics: 0.000074
Time Spent Computing Ball Physics: 0.000074
Time Spent Computing Ball Physics: 0.000070
Time Spent Computing Ball Physics: 0.000075
Time Spent Computing Ball Physics: 0.000077
Time Spent Computing Ball Physics: 0.000075
```

The screenshot above shows a subset of the run time in seconds of the computation thread. It takes approximately .075 milli-seconds to execute this function, or 75 micro-seconds

.Schedule and Discussion



Desired Schedule

```
void desired_schedule() {
      while(1) {
            post_event();
            read_event();
            compute_ball_pos();
            display_ball();
            wait_for_100Hz();
      }
}
```

Available Functions:
- post_event();
- read_event();
- compute_ball_pos();
- display_ball();
- wait_for_100Hz();

The above schedule is desirable, however did not work for multiple reasons. The first reason is that a 100Hz read rate was found to be too slow for accurate computations. The system was very susceptible to noise, as not only samples were being taken to average it out quickly. Thus, the game responded poorly, and the ball was often rolling in a seemingly random fashion. 200Hz was found to be return much more accurate measurements.

Secondly, the sum of the non-zero run-time of each of the functions was greater than the desired period of 10000 micro-seconds, as can be seen in Table 1.

Task	**Frequency**	**Period (1/Freq)**	**Execution Time**
1. Data Read	200 Hz	5000 uS	3000 uS
2. Display Screen	100 Hz	10000 uS	8500 uS
3. Ball Computation	200 Hz	5000 uS	75 uS

Table 1. Note: In final implementation of application, reading from the event file and computing the read data were concatenated into a single function.

Summing up the total execution times gives a value of 11575 micro-seconds. Thus, the actual frequency that the schedule would run at would be only 50Hz, since 50% of the clock ticks would be completely missed.

In my implementation, the scheduling is chosen at runtime by setting the frequency in hertz of each of the major processes; reading and posting data, reading in user space performing computations on posted data, and updating the display. However, the schedule is dynamic because it is enforced using a mutual exclusive lock that only allows one of the two user space thread to be executed at a time. Also, the computation thread calls a blocking read() immediately after locking the mutex, thereby enforcing that computations are always performed using a new set of data.

The most important aspect of my application is that the value of delta-time used in computations (which is set to the period of the frequency at which data is read) is an accurate representation of the rate at which data is actually being read from the sensor. It is possible in this application for the kernel thread responsible for reading data to be blocked when it is supposed to execute by another thread, thus adding an indeterminate offset to the true value of delta time. However, I found that the complementary filter manages to handle computations fine regardless as long as it is run at a fast enough rate (min 200Hz).